

תודה רבה על הרכישה של הספר!

עבדתי מאד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוכרי העריכה. יותר מ-2000 אנשים תמכו בספר זהה ואייפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהkindle, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתווך תקווה שהרוכש והותםך לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

beninson@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעה אצבע דיגיטלית - כלומר בתוך דפי הספר נחברים פרטי הרוכש באופן שקויף למשתמש. כדי מאד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העברו לו את הפרטיהם שלכם באתר ומיהקנו את העתק שנמצא ברשותכם.

תודה וקריאה נעימה!

ללמוד ב'אורה סקייפט בונברית

נן בר-זיך

WixEngineering

Outbrain
Engineering

R Really Good

Chegg®

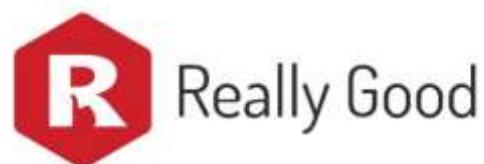
Ono

הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

לימוד ג'אומטריקraft בעברית

רן בר-זיק

מהדורה: 2.1.0



כל הזכויות שמורות © רן בר-זיק, 2019.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלעדי, לא-ייחודי, אישי, בלתי ניתן להעברה (למעט על פי דין), ובלתי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודיים בלבד.

אסור לך להעתיק את הספר, לשכפל אותו, לצור יצירות נגזרות ממנו או לפרסם אותו בכל צורה אחרת.

מותר לך לחתט קטעים קצרים מהספר במסגרת הגנת שימוש הוגן, כלומר פסקה או שתיים, כאשר אתה מפנה למקור ומציר את רן בר-זיק כמחבר הספר.

הדוגמאות המובאות בספר זה הן בבעלות של רן בר-זיק, ואסור לך להשתמש בהן בתוכנות שתפתח. אם אתה רוצה להכניס אותן לפרויקט שלך, שלח מייל ונדבר על זה.

עריכה לשונית: יעל ניר
הגאה: חנן קפלן
עיצוב הספר והכricaה: טל סולומון ורדי (tsv.co.il)

הפקה: כריכה — סוכנות לסופרים
www.kricha.co.il



תוכן העניינים

על הספר	12
על המחבר	13
על העורכים הטכניים	14
דניאל שטרנלייבט	14
gil pinck	14
יגאל סטקלוב	14
תום ביגליין	14
צחי נמבי	15
על חברות התומכות	16
Really Good	16
אואטבריין	16
Chegg	17
Wix.com	17
על ג'אווהסקרייפט	19
אין לומדים	21
ארגנו לעצמכם סביבת עבודה מסודרת	21
קראו את הפרקים לפי הסדר	21
קראו כל פרק פעמיים וכותבו לעצמכם את כל הדוגמאות	21
פתרו את התרגילים לדוגמה בסוף כל פרק	22
התיעצו עם אחרים	22
הגיעו לסדראות ולמייצפים	22
תרגלו, תרגלו, תרגלו	22
התקנת סביבת העבודה ודרכו הלימוד	23
משתנים	32
סקסט	37
מספרים	42
מציאת השארית	44
אופרטורים מקוצרים	46

52	סוגי מידע פרימיטיביים נוספים
52	بولאי
52	 משתנה לא מוגדר
53	רייך
53	Symbol
54	מציאת הסוג של המשתנה
58	הערות
61	בקרת זרימה — משפט-cond
64	אופרטורים השוואתיים נוספים
68	אופרטורים לוגיים
71	אופרטור שלילי
73	אופרטור תנאי
74	אופרטורים המשווים ערכים
85	switch case
94	קבועים
97	בקרת זרימה — פונקציות
101	פונקציה עם ארגומנטים
104	ארגומנטים עם ערכים דיפולטיביים
106	הפונקציה באובייקט
106	Hoisting
108	closure
111	פונקציה אוניבימית ופונקציית חץ
112	פונקציה אוניבימית במשתנה
113	פונקציה אוניבימית שambilodat מהסקופ הגלובלי
122	אובייקטים
128	מחיקת מפתח
128	הכנסת פונקציה כערך
129	קינון ובדיקה מפתח
130	אובייקט קבוע
138	מערכות
143	מערכות ומחרוזות טקסט

146 <i>this-i-new</i>
154 <i>תבנית טקסט</i>
159 <i>לולאות</i>
159 <i>לולאת for</i>
169 <i>לולאה אינסופית</i>
169 <i>לולאת while</i>
171 <i>לולאת do while</i>
171 <i>לולאת forEach</i>
174 <i>לולאת for of</i>
179 <i>לולאת map</i>
182 <i>לולאת filter</i>
186 <i>לולאת sort</i>
186 <i>לולאות על אובייקטים</i>
186 <i>לולאות for in</i>
189 <i>Object.keys</i>
199 <i>ג'אווהסקריפט בסביבת דפדף</i>
199 <i>הסבר כללי על HTML</i>
202 <i>מזהים של תגי HTML</i>
202 <i>גישה אל תגי HTML באמצעות ג'אווהסקריפט</i>
203 <i>אלמנטים של HTML מתורגם לאובייקטים של ג'אווהסקריפט</i>
205 <i>אירועים עם HTML וג'אווהסקריפט</i>
207 <i>הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווהסקריפט</i>
209 <i>שינויי עיצוב</i>
210 <i>סלקטורים של DOM</i>
215 <i>אירועים נוספים</i>
217 <i>פעופע של אירועים</i>
220 <i>הצמדת אירועי ג'אווהסקריפט לאלמנטים ב-HTML</i>
230 <i>דיבאגינג</i>
237 <i>אובייקטים גלובליים ואובייקטים מובנים</i>
240 <i>parseInt</i>
241 <i>eval</i>

241.....	Math
242.....	Date
245.....	JSON
246.....	setTimeout
253.....	בישויים רגולריים
261.....	טיפול בשגיאות
263.....	finally
267.....	מבנה נתונים מסוג Map ו-Set
267.....	Map
268.....	Set
273.....	מבנה אסינכרוני – קולבקים
283.....	Promises
287.....	שרשור הבטחות
290.....	קיבוץ הבטחות
291.....	קיבוץ הבטחות מתקדם
294.....	פונקציית <code>async</code>
301.....	AJAX
303.....	METHODS של HTTP וארגוניים נוספים
307.....	ES6 Classes
314.....	ומה עבשין?
316.....	נספחים: Best Practices
316.....	מה זה Best Practices ולמה כדאי לישם אותם?
317.....	Best Practices שכלי מפתח מKeySpecי צריך להכיר
317.....	בחירה שמות
319.....	KISS
319.....	DRY
320.....	לא להמציא את הגלגל
321.....	Make it work, make it right, make it fast
322.....	תיעוד
322.....	ניהול גרסאות
323.....	לבקש עדנה
323.....	ביקורת עמיתים – Code Review
324.....	Tech Design
325.....	אוטומציה ו-Best Practices
326.....	?linting

327.....	ESLint
328.....	רשימה של Best Practices והחוק הרלוונטי של ESLint
328.....	בלי מספרי קסם
329.....	השוואה קפדיות: ===
329.....	קוד לא נגיש
330.....	חלוקת לחלקים קטנים
331.....	אוריך מינימלי ומקסימלי לשימוש משתנים
331.....	בלי eval
332.....	בלי משתנים שלא הוצאה
333.....	נספח: בדיקות, יציבות ואיכות קוד
333.....	קצת רקע
334.....	מבנה בדיקות
335.....	בדיקות ייחודית
336.....	בדיקות קצרה לקרה (End-to-End)
336.....	בדיקות ממשקי משתמש (UI Tests)
337.....	ספריות ופרימיטו ורקיום מומלצים
338.....	סיכום
339.....	נספח: Velo by Wix
339.....	הקדמה
339.....	מה בונים?
340.....	איך מתחילה?
342.....	הציג נתונים ממסד הנתונים באתר
343.....	Dataset
344.....	חיבור רכיבים למידע מהטבלה
347.....	הוספת משימה חדשה
347.....	Events
349.....	\$w
350.....	wix-data
350.....	wixData.insert() – הוספת רשומה לטבלה
353.....	שינוי סטטוס המשימה
354.....	wixData.get() – שילוף רשומה מהטבלה
355.....	wixData.update() – עדכן רשומה בטבלה
356.....	מספר המשימות שלא הושלמו
356.....	wixDataQuery
359.....	\$w.onReady()
360.....	סינון המשימות לפי סטטוס המשימה
361.....	wixDataFilter
364.....	ניקוי המשימות שהושלמו
365.....	wix-window
366.....	wixWindow.openLightBox()
367.....	wixWindow.lightbox.close()

368.....	מ剔ת רשומה מהטבלה wixData.bulkRemove()
370.....	תוספת – ייצירת מסד נתונים
373.....	Sandbox Live
373.....	Permissions
374.....	סִכּוֹם
375.....	נספח: ג'אוּהַסְקְּרִיפְט מונחה עצמים
375.....	מה זה תכונות מונחה עצמים?
381.....	Prototype based
388.....	נספח: שינויים מהמהדורה הקודמת (מהדורה 2.0.3)

על הספר

הספר "לימוד ג'אווהסקריפט בעברית" מלמד את השפה ג'אווהסקריפט (JavaScript), שפת סקריפט קלה ופושטה ללימוד שהפכה לפופולרית מאוד עם השנים. הספר מיועד ללא-מתכנתים שרצוים להתנסות בשפת תכנות ולמתכנתים בג'אווהסקריפט שרצוים להעמיק את הידע התיאורטי שלהם.

הספר מתחיל בלימוד בסיסי של משתנים ומגיע עד לימוד תכונות אסינכרוני ו-AJAX. בכל פרק בספר יש הסברים לצד דוגמאות רבות הממחישות את העקרונות השונים שהוסבו בו, ובסיומו תמצאו שאלות דוגמה לתרגול עם הסברים מקיפים על הਪתרונות. בנוסף על הספר קיימ אטר המכיל מאות תרגילים ופתרונות אשר יסייעו לכל הלומדים להשיג שליטה בעקרונות השפה ובתחריר שלה. הספר מיועד להביא אדם שלא מכיר את ג'אווהסקריפט לנקודה שבה הוא מכיר היטב את השפה ואת התחריר שלה וידעו איך להשתמש בה כדי לפתור בעיות בסיסיות.

הספר יסייע גם למתכנתים מנוסים יותר שմבקשים לחזק את הידע התיאורטי שלהם. יש בו הסברים על יכולות מתקדמות מאוד של השפה שהוחלו בשנת 2017, כמו מימוש טוב יותר של תכונות אסינכרוני, AJAX באמצעות `fetch` ותכונות נוספות.

על המחבר

REN BAR-ZIK הוא מפתח תוכנה משנת 1996 במגוון שפות ופלטפורמות ועובד כמפתח בכיר במרכז פיתוח של חברות רב-לאומיות, מ-Verizon ועד HPE ועד HPE, שם הוא מפתח בטכניקות מתקדמות הן בצד הלköח, הן בצד השירות, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-CD\AI וכמו כן על אבטחת מידע. בנוסף על מפתחתו כمبرה מלאה, רן הוא עיתונאי ב"הארץ" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטלכולוגיה ולאבטחת מידע וכותב על אינטרנט ורשתות. משנת 2008 מפעיל רן את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרים ווסברים על תכונות בעברית ומתעדכן לפחות פעם בשבוע. רן נשוי ליעל ואב לארבעה ילדים: עומר, כפיר, דניאל ומיכל. רץ למרחקים ארוכים ו חובב טולקין מושבע.

על העורכים הטכניים

דניאל שטרנלייכט

דניאל שטרנלייכט הוא מפתח Frontend מאז גיל 14, המיסיד של המיזמים `Common Ninja` ו-`There is a bot for that`, צרךן כבד של קוד `Open Source` (ומשתדל גם לתרום בהזירה) ונכון לזמן כתיבת הספר מוביל את גילדת `the-Frontend` בחברת `Frontend FEDs Community` שמאגדת מפתחי Frontend מחברות מובילות בארץ ובעולם, ובה מעלים דינמיים, מתייעצים ומשתפים לינקים, חדשות ועדכונים מעולם `the-Frontend`. בעבר כתב בבלוג "עיצוב גרפי וטכנולוגיה", וכיום הוא כותב בלוג טכני על טכנולוגיות `Frontend Node.js`. בשאר הזמן הוא נשוי לרונה ואבא להדר ולאליל המתוקים, מתופף, גולש סקי, משחק כדורים, צופה בסדרות, קורא ספרי פנטזיה ומתח ומדקלם את כל סרטי דיסני בעל פה.

gil fink

gil fink הוא מומחה לפיתוח מערכות ווב, `Google Developer` `Web Technologies` `Microsoft Developer Technologies` `MVP`, `Expert` `sparXys` של חברת `Expert`. כיום הוא מייעץ לחברות ולארגוני שונים, שם הוא מסיע בפיתוח פתרונות מבוססי אינטרנט-`SPAs`. הוא עורך הרצאות וסדנאות ליחידים ולחברות המעוניינים להתחמות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של `Microsoft` `Pro Single Page` (MOC), מחבר משותף של הספר "Application Development" (Apress) ושותף בארגון הכנס הבינלאומי `AngularUP`. [לפרטים נוספים עלgil: http://www.gilfink.net](http://www.gilfink.net)

יגאל סטקלוב

יגאל סטקלוב הוא מפתח Frontend-`Full Stack`, מוביל טכנולוגי ומנהל פיתוח מנוסה בעל ותק של יותר מעשור וחצי בתעשייה. כיום משמש מנכ"ל חברת `Webiya`. בעבר היה מוביל בתחום `Frontend` בחברות `Wix` ו-`Netcraft` והוביל פרויקט פיתוח רבים. יגאל פועל מאדם למען קהילת `the-Frontend` בארץ והוא אחד מהזרים והמארגנים של כנס `the-Frontend` הבינלאומי הראשון בישראל, כנס `YGLF` (`You Gotta Love Frontend`).

תום ביגלאיידן

מפתח `Frontend` ומעצב, נשוי, אב לשתיים וגר בתל אביב.

תומ התחל את דרכו בתחום Frontend בתחילת שנות האלפיים מכיוון פחות צפוי – בפיתוח אפליקציות בג'אווהסקריפט לממירים של IOS בסטארט-אפ קטן. אחרי זה נסגר, עבד עצמאית ובכמה סטארט-אפים, הפרק לאחד המומחים הראשונים בארץ ל-HTML ו-CSS ולא היה בקיא ממנה בBeganim של CSS עם זאת באקספלורר 6. תרם לפרויקטים שונים בקוד פתוח, היה שותף בגין פלטפורמת ניהול התוכן דרupal ובנה את אחד הטמפלטים העבריים הפופולריים תקופה. תומ גם עיצב את האיקונים שעדיין נמצאים בשימוש עד היום בגין הווידיאו הפופולרי בקוד פתוח VLC. לפני קצר יותר משבע שנים נחת ב-XiW, ובהמשך השנים האחרונות עומד בראש צוות קטן ומוכשר שמוביל את תחום Frontend במדיה – וידיאו, תמונות, וקטורים. וanimacions בפלטפורמת בניית האתרים של XiW.

צחי נמני

צחי נמני הוא מומחה בפיתוח, סרבר, אוטומציה וdbName Java, Kotlin, Node.js, C#, Docker, Kubernetes, Git, Terraform, Selenium, Appium, Cypress. צחי הוא עובד בחברת היינק בתחום התיירות. בנוסף, צחי מיעז ומרצה לחברות וליחדים בנושאים של Docker, Kubernetes, CI/CD, Automation Development, Docker and Kubernetes. צחי מאמין גדול בקוד פתוח ותרום לפרויקטים שהוא משתמש בהם.

על החברות התומכות

Really Good

Really Good היא בוטיק פיתוח Front End שעבדה עם סטארטאפים וחברות טכנולוגיות מאז הקמתה ב-2012 על ידי שחר טל ורוני אורבך. אנחנו נחנים לבנות אפליקציות מורכבות עם UX מוקפד ב מגוון טכנולוגיות ללקוחות מעניינים שחוויית המשתמש חשובה להם, ושומרים על איזון בריא בין עבודה לחיים.

אנחנו מגייסים מפתח Front End מנוסים וממש טובים עם תשומת לב לפרטים הקטנים.

<https://reallygood.co.il>

אאוטבריין

אאוטבריין נוסדה בשנת 2006, על ידי זוגbizים ירון גלאי (כימ O) ואורי להב (CTO) הנוכחי ומנהל כללי- אאוטבריין ישראל). החזון המרכזי של החברה הוא לייצר את תוכן האמין והמשמעותי ביותר. מוצר המלצות המרכזי שאאוטבריין פיתחה, מבוסס טכנולוגיית machine learning ומודלים שחוצים את הרגלי'ן. צרכית התוכן של הגולש על בסיס התנהוגיות חזורת. באמצעות קר, האלגוריתם לומד את הגולש ושולח לו המלצות תוכן בזמן אמיתי. היתר, אאוטבריין פיתחה מוצרים טכנולוגיים נוספים, ה Outbrain Amplify ו- Outbrain Smartfeed. טכנולוגיית פרסום הניטיב של אאוטבריין מזיגה את החדשנות והידיעות של אתרי MSN, CNN, BBC, The Washington Post, Sky News The Guardien, Spiegel online, El Pais

מטה חברת אאוטבריין נמצא בניו יורק, משרדיה ממוקמים ב-19 ערים בעולם, מרכז המחקר והפיתוח ברובו יושב בישראל, במשרדים בנתניה.

קישורים:

<https://twitter.com/OutbrainEng>

<https://www.facebook.com/Outbraineng>

<https://www.youtube.com/channel/UCJLORR2uJgIrkM-JIKV-rJA>

<https://www.outbrain.com/techblog>

<https://medium.com/outbrain-engineering>

Chegg

סטודנטים בארה"ב, כמו בכל מקום בעולם, רוצים לרכוש השכלה גבוהה כדי למצוא עבודה טובה, מעניינת וכמו כן רוחנית. רק שלימודים גבוהים עולים כף — והרבה. אז איך מושגים עבודה טובה כשלכתתיליה אין מספיק כף למן תואר ראשון או שני ולהמשיך לחיות תוך כדי? כאן בדיק Chegg נכנסת לתמונה.

Chegg היא חברת אמריקאית שחרתה על דגלה לעזרה לסטודנטים במהלך לימודיהם ומכוונת להפוך את הצורך בהשכלה לאפשרי ומשתלם לכל אחד. את השם היא בחרה בגלל אותה דילמה, והוא למעשה הלחמה של שתי המילים - Chicken- и-Egg. שסמלות את השאלה העתיקה ביותר בעולם הפילוסופיה — מה בא קודם: הביצה או התרנגולת? הלימודים האקדמיים או הניסיון בשוק העבודה?

החברה שמה את הסטודנט ואת הצרכים שלו בראש, ובהתאמתה. "Students First" הוא אחד הערכיים המרכזים לאורם פועלת החברה. Chegg עוזרת לסטודנטים לא רק להיות מצטיינים במהלך הלימודים, אלא גם אחריהם, בתהליך חיפוש העבודה. היא משפרת את התוצאות של הסטודנט האמריקאי בMagnitude שירותים: מהוזלת קניית ספרי לימוד בצורה שטחונית, דרך הצעת שירותים ויישומים לשיפור הזיכרון בלמידה ל מבחנים ומומחי תוכן שימושיים מרוחק, ועד עזרה במימון הלימודים, מציאת המקום להתמחות לציבור ניסיון ואפילו תכנון של קריירה והשלמת מימוןיות עסקיות שחשירות באקדמיה.

החברה האמריקאית הוקמה ב-2007, היא חברת ציבורית בינלאומית ועובדיה בה למעלה מ-1500 אנשים. המשרד הישראלי, שמוקם ברכובות, נוסד בתחילת 2011 והתחליל כפרויקט חיצוני עבר Chegg והיום- המשרד הקטן ברכובות נחשב לחוד החנית של כל תחום הפיתוח של החברה.

Wix.com

Wix.com הינה פלטפורמה גlobלית מובילה לפיתוח נוכחות מקצועית באינטרנט, עם מעלה מ-160 מיליון משתמשים רשומים ברחבי העולם כיום. Wix נסודה על התפיסה כי רשות האינטרנט צריכה להיות נגישה לכל לפתח, לייצר ולשתף תוכן. באמצעות שירותים חינמיים, כמו גם שירותים נוספים למוני פריימים, מאפשרת Wix למילוני עסקים, ארגונים, אישי מקצוע ואנשים פרטיים ליצור ולנהל נוכחות מקצועית ודינמית ברשת. ADI, Wix, ה-אדריטור של Wix, שוק האפליקציות Ascend by Wix ו-X by Wix by Corvid מאפשרים למשתמשים לבנות ולנהל נוכחות>Dינמית>Dיגיטלית. מטה חברת Wix ממוקם בתל-אביב ולחברה סניפים נוספים בארץ שבע, חיפה, ברלין, סן פרנסיסקו, ניו יורק, מיאמי, לוס אנג'לס, סאו פאולו, וילנה, קי"ב, ניירו, דבלין וטוקיו.

במחלקה הפיתוח שלנו, Engineering Wix, שותפים מפתחים מוביילים אשר מעצבים את הארכיטקטורה של השירותים וה מוצר שלנו. הם קובעים את הטון ואת סטנדרט הפיתוח, תור שילוב של היבטי הנדסה, ניהול מוצר, DevOps, ניתוחים וניהול טכני. הם גם מנטורים, וועזרים לمهندסים פחות מנוסים לשפר את כישוריהם, כך שגם הם בסופו של דבר יהיו מסוגלים להיות

מנהלים טכניים בעצמם. אנו כותבים בסקירה, ג'אווה, ג'אווהסקריפט, ריאקט, פיתון, גו ועוד. נוסף על כך, אנו מאמינים מאוד בהעמתת קהילת המפתחים בארץ ובעולם, ולהעניק בחזרה על ידי הצגה בכנסים, כמו גם תרומה לקוד פתוח. המהנדסים שלנו גם תומכים ומשתתפים בכל מיני אירועים טכנולוגיים ללא מטרות רווח.

מוזמנים לנסות בעצמכם לבנות אתר ב-com.Wix וראות עד כמה הפעולה הזאת פשוטה - מה שבעצם מעיד על המורכבות ההנדסית העמוקה מאחורי הקלעים.

על ג'אווהסקריפט

ג'אווהסקריפט החלה את צעדיה הראשונים ב-1993 כשפה שפועלת בסביבת דפדפן ונועדה להעшир דפי HTML. בג'אווהסקריפט יכולנו ליצור אнимציות ופידבק למשתמשים – כל דבר שהיה קשור לאתר אינטרנט דינמיים, למשל דברים שנראים היום מאוד טריוויאליים ופושטים כמו לחיצה על כפתור שעושה פעולה כלשהי בדף. אף על פי שההתחלת שלה הייתה צנואה, ברנדון אייר, ממציא השפה, יצר אותה מלכתחילה כשפה גמישה מאוד. הגמישות הזאת, גם חוסר ההבנה של רבים מהמתכנתים שהשתמשו בה בוגרarily לעקרונות הבסיסיים שלה, גרמו לא מעט מתכנתים בשפות אחרות לזלזל בה. גם השם שלה לא סייע לתדמית. השם ג'אווהסקריפט נקבע מסיבות שיווקיות בלבד – JAVA היא שפת תכנות פופולרית, ואנשי נטסקייפ חשבו שכך יויספו לתדמיתה. בפועל השם זהה לא ממש עזר, ואין כמובן שום קשר בין ג'אווה לג'אווהסקריפט.

על אף ההתחלה הקשה, ג'אווהסקריפט הפכה לפופולרית מאוד. בשנת 1996, חברת נטסקייפ העבירה את השליטה בסטנדרטים של השפה אל ארגון ECMA, ארגון אירופי (היום בינלאומי) המתמחה בתקינה. המהלך הוביל לשחרור הסטנדרט של השפה, שידוע בשם ECMAScript, וג'אווהסקריפט "התיירה" לפי התקינה של ES. משנת 1997, ECMAScript, שנת שחרור ECMAScript, ג'אווהסקריפט, כפי שהיא מישמת בדפדפניים שונים, עוקבת אחר התקינה של ECMAScript, שהיא בעצם "תוכנית המתאר", וג'אווהסקריפט עצמה היא היישום. לכל גרסה יש מספר מילה בצדדים למילים ES (ראשי תיבות של ECMAScript).

מיקרוסופט התנתקה בתקילה ליישום השפה וייסמה שפה מילה בשם Jscript בשם דפדפן אינטרנט אקספלורר, שהייתה בנוייה בדומה לג'אווהסקריפט. למרות היריבות הגדולה בין אנשי מיקרוסופט לאנשי ECMA, שמצרעה כתוצאה מפיתוח שתי שפות שנשענות על שני תקנים מתחרים, העקרונות של ג'אווהסקריפט שלובו גם בגרסה של מיקרוסופט. הפופולריות של השפה עלתה כאשר מקרומדי (יצירת פלאש) שיתפה פעולה עם ארגון ECMA ושייבבה את עקרונות השפה בשפת Actionscript, שימושה את תוכנת פלאש שהייתה פופולרית מאוד אז.

בשנת 2008 נפגשו אנשי מיקרוסופט ו-ECMA באוסלו והחלו בשיחות שלום. בנויגוד לשיחות שלום אחרות שהתקיימו באוסלו, שיחות השלום האלו הסתיימו בהצלחה. תקן ES5, הגרסה הריבועית של ג'אווהסקריפט, שוחרר ויישם בכל הדפדפניים שהיו קיימים אז. מאז, התפתחות השפה והתפוצה שלה הואצז דרמטית. דפדפן כרום, שמריץ ג'אווהסקריפט באופן יצא דופן, נכנס אל השוק בסערה ואפשר למפתחי ג'אווהסקריפט לכתוב סקRYPTים שפועלים על מנגנון 78 העצמתי של כרום ולהריץ ג'אווהסקריפט במהירות מסחררת. השימוש ב-XAJAX תקשורת אסינכרונית עם השרת – נכנס לפעולה, החליף שיטות מישנות כגון Long polling ואפשר לאתרים לספק חוות שימושיות מדיימות למשתמשים. בשנים האחרונות, פרימורוקים וספריות ג'אווהסקריפט אפשרו פונקציונליות מורכבת מאוד וספריות אחרות אפשרו כתיבה של ג'אווהסקריפט גם לטלפונים ניידים ואפילו בטלחות. הראשונות שבספריות האלו נקראו MooTools ו-Query.js והן אפשרו לכל מתכנת לכתוב אפליקציות בטלחות. הספריות האחרונות נקראות ריאקט, אנגולר ו-ASP.NET והן מאפשרות לבנות תוכנות מורכבות מאוד על גבי

הדף (צד הלקוח). ג'אווהסקריפט לא נותרה מוגבלת רק לצד הלקוח, ככלומר לדפדףים ולמכשורי קצה אחרים; המימוש של ג'אווהסקריפט לצד השירות, הידוע בכינוי `js.ode`, הפ לפולרי גם בשרתים. ג'אווהסקריפט מרחיצה כוים אפליקציות מורכבות גם לצד השירות, במיעוד אפליקציות שצרכות לבצע קריאות ולשרת מיליוןית משתמשים. כוים אפשר למצוא ג'אווהסקריפט בכל מקום: באתר אינטרנט, באפליקציות של טלפונים ניידים, באפליקציות המיעודות למחשבים רגילים ומובן בשרתים. הביקוש למתכנת ג'אווהסקריפט נמצא בשיאו ואין זה פלא — אפשר לעשות בשפה זו המון דברים יישומיים כמעט מופע. יש כל כך הרבה ספריות וכלי עזר, עד שבוע יוצאת ספרייה שימושית חדשה. בעזרת ידע מועט אפשר לעשות הרבה מאוד. מה שחשוב הוא ידע בסיסי בשפה.

בשנים האחרונות, תקן ES מתעדכן בכל שנה ומתווספים אליו תכונות ושימושים חדשים. ספר זה מעודכן לגרסה الأخيرة של `JavaScript`. חשוב לציין שאם התקן מתעדכן, אין פירוש הדבר שהעדכון החדש מופיע מיד לדפדףים שמריצים ג'אווהסקריפט או בשרתים שמריצים ג'אווהסקריפט, אלא לוקח זמן עד שהעדכנים החדשים ביוטר עושים את דרכם אל הדפדףים/שרתים שלנו מעתם בהם. אם שמעתם מפתח אינטרנט "מקטרים" על דפדףים ינפים — זו בדיקת הסיבת.

זה המנייע לכתיבת הספר. הבן שלי, כוים מתכנת בזכות עצמו, ניסה ללימוד ג'אווהסקריפט מופע ולא הצליח למצוא ספרים בעברית. החומר שיש כוים בעברית בנוגע לג'אווהסקריפט הוא דל ומיושן. חלק מחוברות העזר הנמצאות בבתי הספר מתיחסות לתקנים שהפסיקו להיות בשימוש בשנת 2007! התיחסות אמיתית לתקנים החדשים ביוטר של השפה, שיצאו בשנת 2017, אין בנמצא בעברית. התחלתי לكتוב הסברים בעברוני, ומפה לשם הבנתי שאני חיב לחת את זה הלאה.

ג'אווהסקריפט היא שפה שקל ללימוד. בניגוד לשפות אחרות, לא נדרש בה סביבת שרת מורכבת או כל פיתוח שעולים כסוף. לא נדרש ידע מוקף במדעי המחשב. כל ש;brיך הוא לפתח `Notepad` במחשב, לפתח דףדף ולהתחליל ללמידה ולפתח. ציריך גם הדריכה נconaה ומשמעות עצמית. אני מקווה שבספר זהה תמצאו לפחות הדריכה נconaה. המשמעות העצמית — עלייכם. אני מאמין שככל שתתקדמי בספר תראו את פירות הלימודים, הניצוץ בעיניכם יתחזק ולא תצטרכו עוד משמעות עצמית — אתם פשוט תתאהבו בג'אווהסקריפט בכלל ובפיתוח לווב בפרט. אל תදלו על הפרק שבו אני מסביר איך למדוד; זהה הפרק החשוב ביותר בספר. אני מאמין לכם הצלחה רבה, בין שאתם מתכנתים בתחילת דרככם לבין שאתם מתכנתים ותיקים שימושיים בספר כדי לשפר את הידע שלכם.

— רן בר-זיק

AIR לומדים

לא למדתי מדעי המחשב באוניברסיטה או במכיליה. למען האמת, עד גיל מאוחר מאוד לא למדתי תכנות בעזרת מדריך. רוב מה שאני יודע למדתי ללא הדרכה, ומכמיה סיבות: הראשונה היא שכאשר עשית את צעדיה הראשוניים בתכנות, בשנת 1996, החומר שהיה זמין באינטרנט היה דל מאוד. על חומר בעברית לא היה מה לדבר, והחומר באנגלית סיפק בדרך כלל רק את הדוקומנטציה. אני לא ממש מתגאה בכך; למידה בלבד ללא הדרכה היא למידה מאוד לא יעילה. הדרכה ממשמעה לא רק מרצה מנוסה, אלא גם אתר אינטרנט שבו יש הסבר מكيف, פורום או קבוצה בראשות חברת זו או אחרת (לא רק פיסבוק), שיש בהם אנשים מנוסים שיכולים לסייע או להפנות לחומר עזר. היא גם מקום שבו אפשר לתרגל ולהתנסות. למרבה המזל, בימים אלו קיימים שלל חומרים, עזרה וסיוע. גם הספר הזה הוא הדרכה. לא תידרשו לצலול לתוך דרך טוביה ללמידה.

כיוון שרוב הזמן למדתי ללא הדרכה, גיבשתי כמה עקרונות למידה שהכנסתי בספר הזה. אני ממליץ לכם לעקוב אחריהם.

ארגנו לעצמכם סביבת עבודה מסודרת

הפרק הראשון עוקק במבנה סביבת העבודה והוא הפרק החשוב בספר. ארגנו את המחשב שלכם וסדרו לעצמכם תיקייה מאורגנת שבה נשמרו את כל התרגולים. אם המחשב שלכם מIFIIZ התראות של עדכוני ג'אווה או משה בסגנון דומה, טפלו בכך. ודאו שגם יכולים להיכנס לאתר הלימודי شاملו את הספר ושהסיסמה שלכם תקינה ופעילה.

קראו את הפרקים לפי הסדר

הפרקים לא פוזרו באקראי אלא תוכנו בסדר מסוים. אין טעם ללמד AJAX לפני שמבינים איך תכנות אסינכרוני עובד. אין טעם ללמד תכנות אסינכרוני לפני שמבינים איך קולבקים עובדים. ואם גם זה נשמע לכם ג'בריש, סימן שאתם צריכים להתחיל מההתחלת. קראו כל פרק לפי הסדר.

קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות

קראו את הפרק פעמיים אחת קרייה שוטפת. לאחר מכן קראו אותו שוב. הפעם קחו את כל הדוגמאות הקוד, העתיקו אותן לסביבת העבודה שלכם וחקקו בהן! נסו לשנות את הערכים, לגרום לשגיאות, לכתוב קוד דומה.

אני מאמין שקוד לומדים דרך הידיים ולא רק דרך הראש. חשוב להבין את התיאוריה ואת הרעיונות מאחורי מה שמנשים לעשות, אך ללא מימוש, הידע הזה יתנוון ויעלם, בדיק כמו בשפת דיבור. אם לא תשתמש ותתרגם, גם הלימוד התיאורטי המעמיק ביותר לא יהיה שווה

הרבבה. הקראיה הראשונה נועדה ללימוד התיאוריה, הקראיה השנייה נועדה לתרגול. לימוד שפה הוא לא מרוץ! קחו את הזמן, כתבו את כל דוגמאות הקוד ונסו לכתוב כאלו משלכם.

פתרו את התרגילים לדוגמה בסוף כל פרק

בסוף כל פרק ישתרגילים לדוגמה. נסו לפתור אותם. אל תתייחסו מהר ואל תרצו אל הפתרון אלא שברואו קצת את הראש. הצלחתם לפתור? נהדר. קראו את הפתרון המוצע ואת ההסבר וראו אם הם דומים לשיכם או שונים במקצת. יש יותר מפתרון אפשרי אחד לכל בעיה....

התיעצו עם אחרים

יש לא מעט קבוצות בעברית (בפייסבוק, אבל לא רק) המזעירות ללימוד ג'אווהסקרייפט ולדיאוינה. מצאו את זו שהכי נוח לכם בה. אל תהססו לשאול שם שאלות. לא הבנתם משהו? דוגמה כלשוי לא הייתה מובנת? הפתרון לתרגיל לא היה ברור מספיק? שאלו שם, ובערבית. אל תהססו לקרוא, למשל דיאוין על MERCHANTABILITY השפה, להשתתף בדיונים או לסייע למשהו שהידע שלו דל משלכם. אל תשחחו להבין את רוח הדברים בקבוצה ולא להציג או להעיק. רוב המשתתפים בקבוצה הם אנשים עובדים שלאו דוקא זמינים להודעות מיידיות.

הגיעו לسدנאות ולミיטאפים

לא מעט חברות מארגנות בחינם סדנאות, מיטאפים ומפגשים שבהם מתכנתים מציגים את ג'אווהסקרייפט ומרצים עליה. כדי מאד להגיע למפגשים האלו, לא רק על מנת לשמעו את התכנים ולהשתתף בסדנאות אלא גם להכיר אנשים אחרים בתעשייה. עם חלוקם אתם תתכתבו בקבוצות הפוייסבוק לג'אווהסקרייפט. קהילת מפתחי הג'אווהסקרייפט בארץ היא קהילה מאד שיתופית וקרובה, ורבים בה מכירים זה את זה.

תרגלו, תרגלו, תרגלו

הספר לא שווה הרבה בלי תרגול מكيف ו שימוש בו. אל תעברו לפרק הבא לפני שתסימנו את כל התרגולים שקשורים לפרק שקראתם. זה לא קרייטי, אלא סופר-קרייטי. באתר hebdevbook.com, המלווה ספר זה, ישנו אתר תרגילים המועד לקוראי הספר.

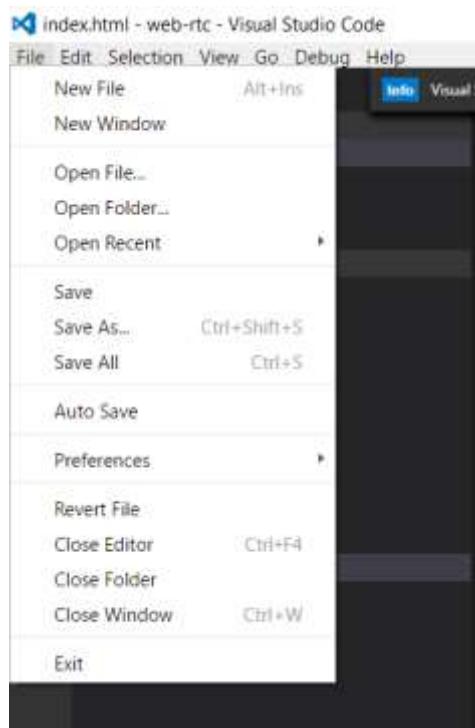
התקנת סביבת העבודה ודרך הלימוד

ג'אווהסקריפט יכולה לזרוץ בסביבת שרת או דרך הדפדפן. איך זה בדוקעובד? ג'אווהסקריפט נמצאת בקובץ טקסט. כן, בדיקן כמו זה שאפשר ליצור באמצעות כתבן הטקסט (Notepad) שיש בכל מערכת חלונות. בדפדפן מותקן כל' שולקח את קובץ הטקסט זהה ומריץ אותו ואת הפקודות שנמצאות בתוכו. אם הדפדפן היה אדם, הוא היה פותח את קובץ הטקסט וקורא את מה שיש בתוכו, למשל: "לך ימינה ופתח את הדלת", ועשה בדיקן מה שכתב. הפעולה הזאת נראית בלשון הפלורית "רינדור", מלשון *render* בlude. הדפדפן לוקח את קובץ הג'אווהסקריפט ומריץ אותו. קובץ הג'אווהסקריפט יכול לעשות כל מיני דברים ולהציג או לא להציג אותם.

איך הדפדפן טוען את קובץ הג'אווהסקריפט? יש כמה דרכי לעשנות זאת, אבל כרגע ארצה ללמד אתכם איך לכתוב קובץ ג'אווהסקריפט ולראות אותו פועל על מנת להבין את כללי השפה ולכתוב משהו באופן ראשון ביוטר. החלק החשוב והקשה ביוטר הוא יצרת סביבת העבודה, הכולמר סביבה ממוחשבת שבה אפשר להקליד ג'אווהסקריפט ולראות אותה עובדת. סביבה זו היא חשובה מאוד כאשר לומדים, כיוון שלימוד של שפת תכנות נעשה ראשית כל "דרך הידים" וחשוב מאוד לא רק לקרוא אלא גם לתרגל ציר סביבה שמאפשרת להקליד פקודות שפה, לשמרן ולראות את הפלט. אני שב ומדגיש: התקנת הסביבה היא החלק הקשה ביותר בתחילת לימוד שפה חדשה וגם החשוב ביותר. לפיכך כדאי להיאזר בסבלנות, לחתת נשימה ארוכה ולזכור שדוקא עכשו מתמודדים עם החלק הקשה ביותר.

הבה נתחיל בעורך טקסט טוב. אמונם אפשר להשתמש בנוטפ, אבל הוא לא מציע צביעת קוד לצורך עזרה בקライות ולא יצרת הוצאות בקלות. יש כמה עורכי טקסט המותאימים במיוחד לכתיבת ג'אווהסקריפט, שאצ'ין כמה מהם. בחרו בעורך טקסט אחד! רובם זהים למדי ומכלים יכולת עריכה בסיסית של HTML, CSS וג'אווהסקריפט.

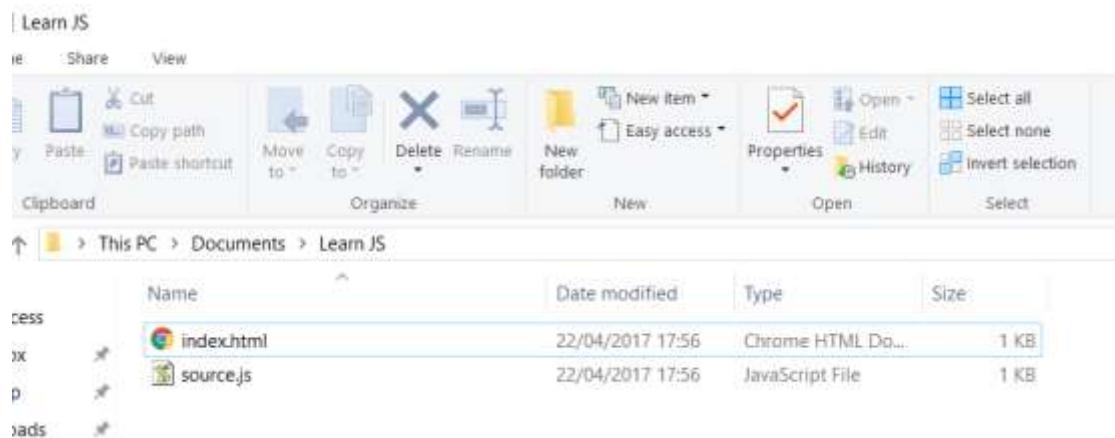
שימוש לב: מתקנים מנוסים לא משתמשים בעורך הטקסט הפשט אל' בעורך טקסט משוכלן. יותר שנקרא IDE או Integrated Development Environment – סביבת פיתוח משולבת. הסביבה הזאת מאפשרת השלמת קוד, מצינית שגיאות בכתיבה וגם יכולה להריץ את הקוד עצמו. IDE מעולה הוא Visual Studio Code. הוא חינמי, מבוסס קוד פתוח, כתוב בג'אווהסקריפט (כן, כן), נתמך על ידי מיקרוסופט ונitin להורדהפה: <https://code.visualstudio.com/>. אחרי התקינה תוכלו לפתוח את התוכנה, לבחור ב-File וatz לפתוח את התקינה שבחרתם ולפתחו או ליצור בה קבצים. אני ממליץ לכם בחום להוריד את עורך הקוד הזה: הוא חינמי לחלוטין ואין מכביד על המחשב.



IDE מוצלח אחר הוא Atom. גם הוא חינמי וxebiooo קוד פתוח וגם הוא... כתוב בג'אוּהַסְקְּרִיפְט. הוא נתמך על ידי גיטהאב וניתן להורדה מה-<https://atom.io/>. הוא ד"ר דומה Visual Studio Code ואחריו ההורדה וההתקנה שלו מאפשר להפעיל אותו בקלות. עורך טקסט נוסף שנחשב לאמין וטוב הוא תוכנה חינמית בקוד פתוח שנקראת `Notepad++`. הוא ניתן להורדה בקישור הבא: <https://notepad-plus-plus.org/download> והוא בסיסי יותר מאשר קודמי.

שימוש ב-Visual Studio Code או ב-Atom הוא מומלץ יותר כיוון שיש בו "השלמה אוטומטית" של פקודות נפוצות בג'אוּהַסְקְּרִיפְט, דבר המקל מאד את הלמידה. כמו כן הוא מציג שגיאות בקוד כבר במהלך הכתיבה, עוד לפני הרצה. קובץ הסביבה הטובה ביותר ביותר ללימוד היא ייצירת קובץ HTML שטוען קובץ ג'אוּהַסְקְּרִיפְט. קובץ HTML הוא קובץ שהדף יודע לפרש ולהציג, והוא יכול לקרוא לקובץ ג'אוּהַסְקְּרִיפְט באופן זה שהדף ירנדר אותו. כאמור, רינדרו הוא הרצת הפקודות שנכתבות בקוד ג'אוּהַסְקְּרִיפְט והציגן על המסך או במקום אחר. רינדר, מילשון `render`, הוא מונח מתחום מדעי המחשב ופירושו הוא "הרצתה". כאשר כתוב "הקוד מרונדר" הפירוש הוא שהקוד רץ ומציג את התוצאות. יש ליצור במחשב תיוקיה – זה יכול להיות ב"המנסכים שלי" או על שולחן העבודה – ובתוכה ליצור קובץ ששמו `source.js` וקובץ בשם `index.html`.

בחלונות ייצרת תיוקיה נעשית באמצעות: כניסה אל סיר הקבצים, בחירת המקום שבו רצים למקם את התיוקיה. לחיצה על המקש הימני של העכבר ואז בחירה ב"חדש" וב"תיוקיה". את הקובץ עצמו כדאי ליצור באמצעות התוכנות Visual Studio Code או Atom או אפילו `Notepad++`. פיתחו את אחת התוכנות האלה (אני ממליץ על Visual Studio Code) ונותנו אל התיוקיה ובחרו ב-File וاز New.



ודאו שמערכת החילונות או המק שלחכם תומכת בתצוגת שם הקובץ המלא, כולל הסיומת .index.html.txt (Extension) של הקובץ. אחרת, קובץ ה-.html יהיה בעצם index.html (Extension) של הקובץ.

בקובץ ה-HTML כתבו את הקוד הבא:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <script src=".//source.js"></script>
</body>

</html>
```

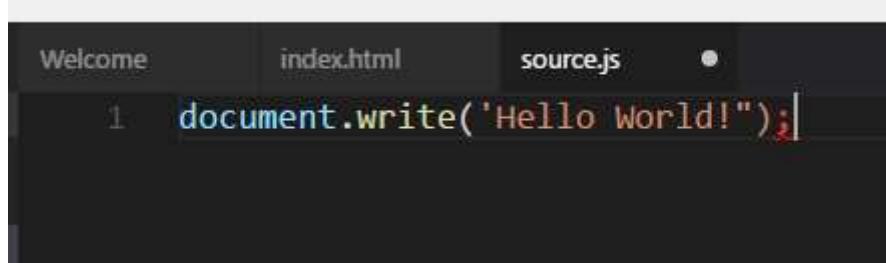
בתוך קובץ ה-.js source כתבו את הטקסט הבא:

```
document.write('Hello World!');
```

אחרי לשמורת את תוכן שני הקבצים, פתחו את הקובץ index.html בדף כרום או בפיירפוקס (לא באdeg'). אם הכל תקין, תראו שכתוב על המסך "Hello World!" – כתבתם את הג'אויסקייפ הראשן שלכם!
שיםו לב: זה השלב המועדף ביותר לפורענות, שעלול להיות מתסכל מאוד, אבל הוא שלב חשוב ביותר ואסור להרים ידיים ולהתאייש. אם פתחתם את הקובץ ודבר לא הופיע, נסו את הדברים הבאים:

בדקו שאנן קראתם לקבצים index.html ו-.js source. הבדיקה צריכה להתבצע באמצעות הכפתור הימני של העכבר בחלונות, כדי למנוע מצב שבו מערכת הפעלה הוסיפה תוספות לשמות index.html ו-.js נשמר בשם index.html.txt.
בדקו שאתם פותחים את הקבצים בדף כרום או בפיירפוקס. בדקו שאין תוספים מיוחדים לדפינים שעולמים לחסום את הרצת הקובץ על ידי הריצה של מצב פרטיות.
בדקו שהקלידתם את הטקסט כשורה source בקובץ js ללא רווחים או תווים מיוחדים.

נסו להשתמש ב-Atom או ב-Visual Studio Code. שימו לב שאין התראות על שגיאות הקלדה. כאן למשל מובאת דוגמה של שגיאת הקלדה שבייצעת. אם קיבלתם התראה כזו, בדקו שוב שלא טעתם בגרש ולא הכנסתם רווחים מיותרים כמו בדוגמה הזו שבה יש שימוש שגוי בגרש יחיד ואז בגרשיים. עדיף להשתמש תמיד בגרש יחיד:

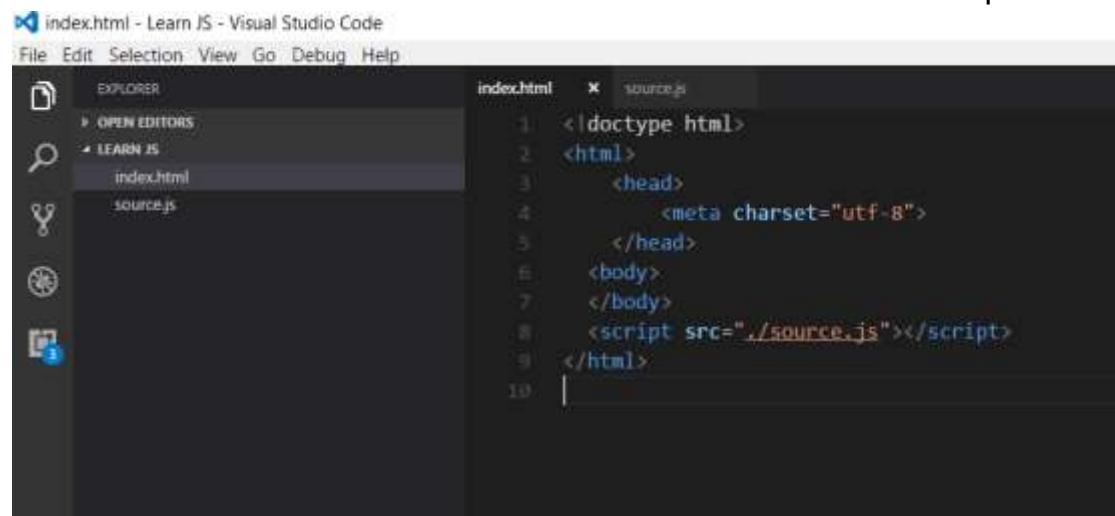


The screenshot shows the Atom code editor with a dark theme. The status bar at the top indicates 'Welcome', 'index.html', and 'source.js'. The 'index.html' tab is active, showing the following code:

```
1  document.write('Hello World!')
```

A red squiggly underline is under the closing parenthesis ')' of the 'write' function call, indicating a syntax error.

אם אתם משתמשים ב-Atom או ב-Visual Studio Code, סביבת כתיבת הקוד שלכם אמורה להיראות כך:



The screenshot shows the Visual Studio Code interface. The title bar says 'index.html - Learn JS - Visual Studio Code'. The menu bar includes File, Edit, Selection, View, Go, Debug, Help. The left sidebar shows the 'EXPLORER' view with 'OPEN EDITORS' and 'LEARN JS' sections, with 'index.html' selected. The main editor area shows the following code:

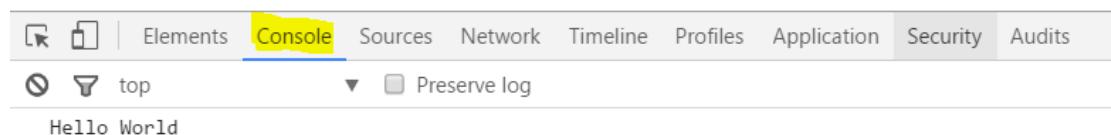
```
1  <!doctype html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5      </head>
6      <body>
7      </body>
8      <script src="./source.js"></script>
9  </html>
```

מצד שמאל תוכלו לראות את כל הקבצים בתיקייה. כדאי ליצור תיקייה מיוחדת בשם learnJavaScript או בשם דומה ולא לשים את כל הקבצים בתיקייה משותפת כמו "המסמכים שלי".

מצד ימין תוכלו לצפות בתוכן הקבצים. בצלום המסך רואים את תוכן הקובץ index.html. אם תקלידו דבר מה, תוכלו לראות שיש השלה אוטומטית של קוד HTML, ואם תקלידו בקובץ source.js שמכיל את הג'אווהסקריפט תראו שיש השלה אוטומטית של פקודות ג'אווהסקריפט. נוסף על כך, תקבלו גם התראה על קוד לא תקין.

כיוון שאתה כבר מפתחי ג'אווהסקריפט מנוסעים, כדאי שתתלמדו להשתמש בקונסולה של הדפדפן. מדובר במשק מיוחד שמאפשר "לדבג" את ג'אווהסקריפט. הפעול "לדבג" כוננו להסתכל על צפונות הרינדור ולראות ממש את הפלט של השפה או את תוכנות הרצה שלה. כלומר מה שכתבנו. נשמע מסובך? יש להבין איך הדפדפן מנסה להריץ את הקוד שלו (כאמור מה שנקרא "רינדור", מילשון הרצה, באנגלית) ולקבל מידע נוסף במקרה שהוא נכשל, כלומר "זורק" שגיאה צבועה באדום בקונסולה ואף מפנה לשורה הביעיתית.

על מנת לראות את הקונסולה, פתחו את כל המפתחים של הדפדפן. בכרום ובפיירפוקס לחצו `i` + `Ctrl` + `Shift` אם יש לכם חלונות או `Cmd` + `Option` + `i` אם יש לכם מק. אפשר לעשות את זה גם דרך התפריט העליון בשני הדפנדנים. לאחר פתיחת כל המפתחים, לחצים על **Console**.

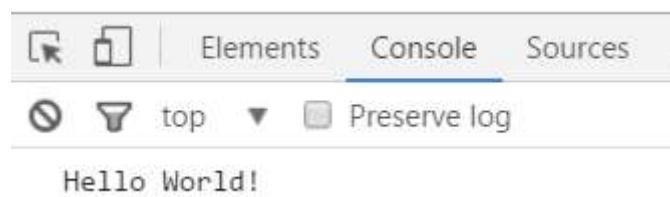


הבה נבדוק מה אפשר לעשות בעזרת הקונסולה. היכנסו אל `js`-`source` והחליפו את הטקסט `://`:

```
console.log('Hello World!');
```

טענו מחדש את הדף באמצעות `F5` + `Ctrl` + `R` בחלונות או `Cmd` + `R` במק. הטעינה מחדש חשובה. הדפדפן לא יודע שהוכנסו שינויים בקובץ הג'אווהסקריפט, ויש לגרום לו לטען מחדש את קובץ הג'אווהסקריפט על מנת להריץ את הפקודות חדשות. אחריו הטעינה מחדש הסתכלו על לשונית ה-`Console`.

אם הכל תקין, המסר יהיה ריק, אך בקונסולה תראו `Hello World!` יש!



חשוב: אל תדלגו על השלב הזה. בכל שלבי הלימוד כדאי להשתמש בקונסולה, שהיא הרבה יותר נוחה להצגה. אם משחו לא עובד, אנא בדקו את הדברים הבאים:

1. אם השלמתם את שלב הקוד? הצלחתם להציג `Hello World` על המסך?
2. אם שמרתם את הקובץ לאחר השינויים?
3. אם טענתם מחדש באמצעות `Ctrl` + `F5` את הדפדפן?

מדובר בסביבה העבודה ביותר ללימוד ג'אווהסקריפט, אך יש סביבות עבודה נוספות. ברשות יש אתרים המאפשרים לכתוב ג'אווהסקריפט ישירות, להריץ את הקוד דרך דרכם ולראות את התוצאות. אתר מפורסם ופופולרי הוא <https://codepen.io/> ואפשר להקליד בו פקודות של ג'אווהסקריפט. פתחו שם חשבון וצרו "pen" חדש. בדקו שיש אפשרות להכניס קודי `HTML`, `CSS` ו-`JS`, שהוא בעצם קיצור של ג'אווהסקריפט. אפשר להכניס את הקוד ולראות את התוצאות על גבי דף מודמה או על גבי הקונסולה של הדפדפן.

מכאן, דרך הלימוד תהיה פשוטה למדי. אני אסביר על תוכנות מסויימות של השפה ואותן דוגמאות. מומלץ בחום רב להעתיק את הדוגמאות אל קובץ `js`-`source` ולהריץ את הג'אווהסקריפט כדי לראות איך זה עובד באמת.

בסוף כל פרק יש תרגילים לתרגול עצמאי ומומלץ מאוד לנסות אותם בסביבת העבודה. אי-אפשר ללמוד שפה על ידי קריאה תיאורטיב בלבד ורצוי לתרגל, לתרגל, לתרגל. את התרגול עושים רק בסביבת עבודה יציבה. לפיכך, אנא אל תלגו אל הפרק הבא לפני שיש לכם סביבת עבודה יציבה. מומלץ מאוד לעבוד ב-`Visual studio code` הכנמית.

תרגיל:

במקום "Hello" גרמו לקונסולה להדפיס את המילים "Ahla Bahla".

פתרון:

היכנסו לקובץ `source.js`, מחקקו את הטקסט שיש שם והדביקו במקומו את:

```
console.log('Ahla Bahla');
```

שמרו את הקובץ, פתחו את הקובץ `index.html`, שטוען את הקובץ `source.js`, ורעננו את הדף. לחזו על `i` + `Ctrl + Shift`, לחזו על הלשונית `Console` וראו את התוצאות.

תרגיל:

גרמו לקובץ ה-HTML לטעון קובץ ג'אווהסקריפט בשם targil.js ושמדיו בקובוסולה: I am a .new file.

פתרונות:

צרו קובץ targil.js באותה תיקייה של הקובץ index.html. יש לוודא שהוא בשם האמתי של הקובץ ושמערך הפעלה לא מźמינה לקובץ עם הסיומת .txt. את זה עושים על ידי בדיקה בהגדירות התצוגה של מערכת הפעלה.פתחו את קובץ ה-HTML בעזרת עורך טקסט (מומלץ להשתמש ב-Visual Studio Code) ושנו את שם הקובץ הג'אווהסקריפט ל-targil.js.

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <script src="./targil.js"></script>
</body>

</html>
```

בקובץ targil.js כתבו את השורה זו:

```
console.log('I am new file');
```

פתחו את הקובץ index.html בדף ולחצו על מנת להציג את כל המפתחים. בחרו את הלשונית Console ובחנו את התוצאה.

פרק 1

משתנים



משתנים

החלק הבסיסי והחשוב ביותר הוא המשתנה. מדובר ברכיב שיכל להכיל בתוכו מידע, ואפשר לשנותו – זו הסיבה שהוא נקרא "משתנה". משתנה בג'אווהסקריפט מוגדר באופן הבא:

```
let variant;
variant = 'Hello World';
```

מה קורה פה? יש כאן הגדרת משתנה בשם `variant`. ההגדירה נעשית באמצעות המילה `shmorah let`. מילה שומרה היא מילה מיוחדת בשפה שהשימוש בה שומר למקורה מסוים. במקרה זה, `let` שומרה אך ורק להגדרת משתנים.

לאחר מכן מכנים ערך למשתנה. הפעולה הזאת נקראת "הציב ערך" או "השמה", והוא נקראת `cr mciyon shahur` מוצב בתחום המשתנה. במקרה הזה מדובר בטקסט הכולל את המילים `Hello` ו- `World`.

שימוש לב: יש סימן ";" בסוף כל שורה. בג'אווהסקריפט מוטב להציב סימן ";" בסוף כל שורה כדי למנוע בעיות וכדי שהסקריפט יעבד. הוא מסמן סוף שורה מעור מנוע הג'אווהסקריפט, שמרננדר את הסקריפט, ממש כמו נקודה בסוף משפט. על אף שלא כל מנוע מקפיד על כך, אני ממליץ לכם: הקפידו תמיד להקליד ";" בסוף כל שורה.
אפשר לזרק ולהציב את הערך מיד בהגדרת המשתנה:

```
let variant = 'Hello World';
```

הבה נבדוק את המשתנה ואת הערך שלו. אפשר להציב את המשתנה הזה בתחום `log`.
כפי שראנו בפרק הקודם:

```
let variant = 'Hello World';
console.log(variant);
```

אם תציגו בקונסולה, תראו שמודפס המשפט "Hello World". מדוע? כיון שהזה מה שיש בתוך המשתנה. מן הסתם, משתנה ניתן לשינוי. נסו את הקוד הזה:

```
let variant = 'Hello World';
variant = 'I am a new version';
console.log(variant);
```

מה לפि דעתכם יוצג בקונסולה? יוצג "I am a new version". למה? כיון שהערך הקודם "נדפס" על ידי הערך החדש. הכנסתם (כלומר הצבתם) ערך חדש לתוך המשתנה, ועכשו מה שיש בתוכו השתנה. כמשמעותו רואים את המשתנה רואים את הערך החדש.

יש הבדל מהותי בין הגדרת משתנה לבין הכנסת ערך לתוכו. הגדרת משתנה היא כמו בניית ארון או קופסה ואפשר בכל פעם להכניס לתוכו ערך אחר.

שימוש לב: אחרי משתנה מסוים הוגדר, אי-אפשר להגדיר אותו מחדש. הקוד הבא:

```
let variant = 'Hello World';
let variant = 'I am a new version';
console.log(variant);
```

גורם לשגיאה הבאה: **Identifier 'variant' has already been declared** שמות המשתנים יכולים להיות מגוונים אך יש להם כמה כללים מחייבים. אפשר להשתמש בכל אות שהיא ובסימנים `_` או `$` בתחילת השם, ובכל האותיות והמספרים ובסימנים `_` או `$` בהמשך השם.

שמות לא תקינים	שמות תקינים
3myVar מתחיל במספר	myVar3
my-var מכיל את התו - שאינו תקין	my_var
#myVar מכיל את התו # שאינו תקין	\$myVar
my var מכיל רווח	myVar

שימוש לב: אפשר להשתמש בעברית בהגדרת המשתנים, אבל מומלץ שלא לעשות את זה גם כיון שהקוד שלכם יהיה פחות קרייא וגם כיון שהזה עלול להוביל להתנהגות מוזרה ולצרות. אתן לכם טיפ: בעולם התכנות אל תחשפו צרות כי יש מספיק מהן גם כך.

שימוש לב 2: בגרסאות קודמות של ג'אווהסקריפט השתמשו במילה השמורה `var` להגדרת משתנה. בתקנים החדשים כבר לא מקובל להשתמש `var` כיון שלשימוש בו יש השלכות

שנדון בהן בהמשך הספר. הוא נשאר איתנו בעיקר בשביל תאימות לאחר עבר קוד שנכתב בשנים עברו.

תרגיל:

צרו משתנה בשם המרכיב לפחות שתי מיללים, הדפיסו אותו בקונסולה וודאו שהדפסה בקונסולה יופיעו המילים "I know JavaScript".

פתרון:

```
let myVar;  
myVar = 'I know JavaScript';  
console.log(myVar);
```

הסבר:

יצירת המשתנה נעשית באמצעות המילה השמורה `let`. השמת הטקסט נעשית באמצעות הסימן `=`. שמו לב שהtekst מוקף בגרשיים. פקודת `console.log` שלמדנו בפרק הקודם משמשת להדפסת הטקסט - ובמקרה זהה המשתנה שהוא מקבלת.

תרגיל:

צרו משתנה בשם `myVar` והכניסו לתוכו את טקסט "me not know JavaScript". דרשו את הטקסט הזה בטקסט "I know JavaScript" והדפיסו אותו באמצעות `console.log`.

פתרון:

```
let myVar = 'me not know JavaScript';
myVar = 'I know JavaScript';
console.log(myVar);
```

הסבר:

יוצרים את המשתנה `myVar` ומכניסים לתוכו את הטקסט "me not know JavaScript" ממש ברגע היצירה. לאחר מכן דורסים את הערך הזה באמצעות השמה נוספת. הדרישה של מה שיש במשתנה נועשית באמצעות `console.log`.

תרגיל:

צרו משתנה בשם `myVar` והכניסו לתוכו את הטקסט "I am myVar". צרו משתנה נוסף בשם `myVar2` והכניסו לתוכו את הטקסט "I am myVar 2". הדפיסו את שניים באמצעות `console.log`:
(בונוס: הדפיסו את שני המשתנים בקונסולה באמצעות פקודת `log` אחת)

פתרון:

```
let myVar = 'I am myVar';
let myVar2 = 'I am myVar 2';
console.log(myVar);
console.log(myVar2);
```

הסבר:

אין מניעה להגיד כמה משתנים באוטו סקריפט. הגדרתי את המשתנה באמצעות המילה השמורה `let`, שם המשתנה (הכולל מספר בסוף) והדרישה שלו ל-`log`.
הסביר לתרגיל הבונוס: נסו לחפש באינטרנט איך להדפיס שני משתנים. הרבה פעמים יהיו דברים שלא תבינו בתרגול. במקרה לקרוא שוב ושוב את ההסביר, נסו לחפש תשובה בראשת. כך עושים מתקנתים מקצועניים: הם מוחפשים תשובות בראשת.

פרק 2

טַהֲרָת



טקסט

טקסט (באנגלית `text string`) הוא סוג מידע חשוב מאוד ובסיסי מאוד במערכות מידע. משתמשים בטקסט בכל מקום ובכל מערכת, ומדובר במקרה בחלק חשוב מאוד בלימידת קוד – יותר ממתמטית ופעולות חשבוניות. בעבר היה קשה לעבוד עם טקסט שאינו באנגלית. אם אתם מבוגרים מספיק אתכם וdae' זוכרים כל מיני אותיות מיוחדות שהופיעו באתרים או בתוכנות. היום קל לעבוד במגוון שפות בתוכנות, באתרים וכמוון בג'אווהסקריפט, בזכות תקן חדש שנקרא "יוניקוד". יוניקוד מאפשר לכתוב בעברית, ביפנית, ברוסית ובכל מערכת כתוב אחרת בקלות ובלי חשש להופעת ג'יבריש והוא מותם במחשב שלכם באופן אוטומטי.

בפרק הקודם למדנו איך ליצור משתנים ולהכניס לתוכם טקסט. בפרק זהה תרחבו את הידע שלכם בנושא הטקסט. כאמור, אפשר להכניס טקסט למשתנים בג'אווהסקריפט באופן הבא:

```
let myVar = 'This is text';
```

הтекסט מוקף בגרש בודד, אבל אפשר להשתמש גם בגרשיים כפולים:

```
let myVar = "This is text";
```

הבחירה ביניהם היא בדרך כלל עניין של טעם. רוב המתכנים נוהגים נcone להיום להשתמש בגרש בודד להגדרת טקסט או בגרש מסווג (`backtick), שעליו נלמד מאוחר יותר. ג'אווהסקריפט מאפשר לחבר בין מחרוזות טקסט די בקלות בעזרת הסימן `+` (כמו בפעולה חיבור):

```
let part1 = 'foo';
let part2 = 'bar';
let myVar = part1 + part2;
console.log(myVar);
```

מה שיודפס על הקונסולה הוא `foobar`. אף על פי שהסימן `+` מוכר מחיבור מספרים, בג'אווהסקריפט אפשר כאמור להשתמש בו לחיבור מחרוזות.

שימוש לב: הרוח בין שמות המשתנים לבין הסימן `+` לא הכרחי להרצתה תקינה אבל נוח מאוד לקרוא. חשוב לציין שהשם התקני של טקסט הוא "מחרוזת טקסט".

שימוש לב: השימוש ב-`foo` וב-`bar` הוא מאוד נפוץ בדוגמאות בשפות תכנות ונדיר למצוא מדריך לשפת תכנות שלא משתמש במילים האלה ובמילה `baz` למשתנים (או לחלקים אחרים בקוד שעוז תלמדו עליהם). לדוגמאות האלו יש שם מפכיד: משתנים מטה-סינטקטיים – אבל לא צריך לפקד משמות מסובכים.

נשאלת השאלה מה קורה אם רוצים להכניס מחרוזת טקסט עם גרש, שהוא בסגנון זהה: `I don't know JavaScript`.

לדוגמה:

```
let myVar = 'I don't know JavaScript';
console.log(myVar);
```

ישizar שגיאה בקונסולה. כיון שכל מחרוזת טקסט חייבת להיות מוקפת בגרשיים, המנווע של ג'אואסקרייפט חושב שכל מה שmag'ע אחרי הגרש הוא משתנה ולא יודע מה לעשות איתו. בדוגמה הבאה, החלק המודגש הוא מחרוזת הטקסט, והחלק שאינו מודגש הוא מה שהמנוע של ג'אואסקרייפט לא יודע מה לעשות איתו:

```
let myVar = 'I don't know JavaScript';
```

יש כמה דרכים לפטור את הבעיה זו. הדרך הטובה ביותר היא לצין לפני המנווע של ג'אווהסקריפט שהגרש שיש במילה 'don' הוא גרש שמהווה חלק ממחזורת הטקסט. איך עושים את זה? באמצעות הסימן \:

```
let myVar = 'I don\'t know JavaScript';
console.log(myVar);
```

ההדפסה תציג את הטקסט כמו שציריך, עם הגירש. הפעולה הזו נקראת **escaping** ובה לוקחים טקסט שהוא שובר את שפת התכונות והופכים אותו לבטוח. לחלופין, אפשר ליצור אותו תווים שלא ניתן לכתוב במקלחת, למשל ירידת שורה. איך כתבים ירידת שורה? בעזרת חא. העתיקו את הדוגמה הזו:

```
let myVar = 'I don\'t \nknow\n JavaScript';
console.log(myVar);
```

ונרא מה קורה. בקונסולה יופיעו שלוש שורות.
יתכן שתיתקלו ב-escaping במקומות נוספים. אם רוצים לכתוב רק \, צריך לעשות לו escaping ולכתוב \\\.

כעת, כשאתם מרגישים בנוח בכל מה שקשרו לטקסט או, נכון יותר, למחוזת טקסט בג'אوهסקריפט, הבה נסביר מעט את העניינים. בג'אוהסקריפט מחוזת טקסט נחשבת לסוג מידע. באנגלית זה נקרא **Data Type**. יש כמה סוג מידע, כמו מספרים למשל, אבל סוג המידע של מחוזת טקסט נקרא **string** והוא עומד בראשות עצמו. כאשר יוצרים משתנה ומכניסים לתוכו טקסט, בעצם אומרים שהמשתנה הוא מסוג מחוזת טקסט. ברגע שהמשתנה מקבל לעצמו מידע והופך למשתנה מסוג מסוים, הוא מקבל גם סט של יכולות מיוחדות, מסוגים穰ת קולר קנט שהופך לסופרמן. מהשניה שנקנס למשתנה טקסט, אפשר לעשות בו כמה פעולות שאפשרות אך ורק לסוג המידע של הטקסט.

כך למשל אפשר לבצע על המשטנה המכיל את טקסט הפעולה שתגרום לכל האותיות שלו להיות ראשיות (כלומר קפיטלים). דוגמה:

```
let myVar = 'Hello World';
let myVarUpper;
```

```
myVarUpper = myVar.toUpperCase();
console.log(myVarUpper);
```

יצרים משתנה עם טקסט. על המשתנה זהה עושים פעולה שאפשרית אך ורק עם משתנה מסווג טקסט. שם הפעולה הוא `toUpperCase()`. התוצאות של הפעולה זהה מושמות למשנה נוספת בשם `myVarUpper` ועודפסות. אם תעתיינו את הטקסט הזה ותריצו בקונסולה, תראו שהתוצאה היא 'HELLO WORLD'.

השימוש בהפעלת פועלות על משתנה עם **Data Type** מסוים אינו בלבד רק למחוזות, ובהמשך הספר תראו שאפשר לבצע פועלות גם על **Data Types** אחרים. בinityים צריך להבין שיש סט רחב מאד של פועלות שאפשר להפעיל על מחוזות, `-toUpperCase()` היא רק אופציה אחת. על מנת להפעיל פועלות שונות משתמשים ב-`(נקודה)` על המשתנה. אם משתמשים ב-`IDE` (עורק טקסט) כמו `VS Code`, כאשר תרשמו ". Achri משתנה, עורך הקוד יראה לכם את שלל הפעולות האפשריות על המשתנה זה.

שים לב שהפעולה `toUpperCase()` לא משנה את המשתנה עצמו! שינוי המשתנה עצמו נקרא "מוציאה", זה לא מה שעושים בה. כדי לקבל את הערך שהשתנה צריך להגדיר משתנה נוספת ולהכניס את הערך למשנה. מבלב? הנה נדגים זאת שוב בפעולה נוספת. הפעם צרו משתנה והכניסו לתוכו טקסט. המשתנה הראשון, מהרגע שיש בתוכו טקסט, יכול לבצע פועלות של טקסט ולהציג את התוצאה אל משתנה אחר שאותו תדפיסו:

```
let firstVar = 'HELLO WORLD';
let secondVar = firstVar.toLowerCase();
console.log(secondVar);
```

יש פועלות נוספות שאפשר לעשות על מחוזות טקסט, אבל כרגע התמקדו בשתי הפעולות האלה גם בתרגול.

תרגיל:

צרו שני משתנים, אחד מהם מכיל את המילה `Hello`, והאחר מכיל את המילה `World`. חבו בינם, הכניסו את התוצאה אל משתנה שלישי והדפיסו אותה.

פתרון:

```
let myVar1 = 'Hello';
let myVar2 = 'World';
let answer = myVar1 + myVar2;
console.log(answer);
```

הסבר:

יצרים ומצביעים ערכיהם במשתנים בשם `myVar1` ו-`myVar2` (יש להקפיד שמחוזות הטקסט יהיו מוקפות בגרשיים בודדים). מגדירים משתנה שלישי בשם `answer` ושים בתוכו את

הסכום של myVar1 ו-myVar2. ההדפסה נעשית כרגיל. שימוש לב שרווח גם נחשב לאות. אם לא מכנים רוח באחת מחרוזות הטקסט בתוך המשתנים, לא יהיה רוח.

תרגום:

צרו משתנה שיכיל את מחרוזת הטקסט:

."The student's and the teacher's motivations were in conflict"

פתרון:

```
let myVar = 'The student\'s and the teacher\'s motivations were in
conflict.';
console.log(myVar);
```

הסבר:

שימוש לב לשימוש ב-**escaping** מה זה escaping? פשוט שימוש בסימן \ על מנת להודיע למנוע של ג'אווהסקריפט שהגרש הוא חלק מחרוזת הטקסט. ההשמה וההדפסה של המשתנה נעשות כרגיל.

תרגום:

הכניסו את הערך JavaScript. המירו אותו לאותיות גדולות והדפיסו את התוצאה. המירו אותו לאותיות קטנות והדפיסו את התוצאה.

פתרון:

```
let myVar;
let answer;
myVar = 'JavaScript';
answer = myVar.toUpperCase();
console.log(answer);
answer = myVar.toLowerCase();
console.log(answer);
```

הסבר:

יצרים שני משתנים ריקים בשם myVar ו-answer. הראשון מכיל את מחרוזת הטקסט JavaScript. אחרי שמכניסים לתוךו את הערך זהה, הוא מסוג טקסט ואפשר להשתמש בפונקציהtoUpperCase(). את תוצאות הפעולה זו מכניסים ל-answer ומדפיסים. אחרי הדרישה משתמשים בפונקציהtoLowerCase() על מה שיש ב-myVar, מכניסים את התוצאה אל answer ומדפיסים.

פרק 3

מספרים



מספרים

בדיקן כמו שאפשר להכניס טקסט לתוכה משתנים, אפשר להכניס אליהם מספרים. למשל:

```
let myVar = 12;
console.log(myVar);
```

המספר 12 הוא ללא גרשים. הוא ממש נכנס כמו שהוא, כיון שהוא מספר. מספרים טהורים יכולים להכנס ללא גרשים. להזכירם, מחוץ לטקסט, עליה למדנו קודם, מוקפת בגרש או בגרשיים. מספרים לא מוקפים בגרש או בגרשיים.

אם תדפיסו את המספר כמו בדוגמה, תראו שיזדפס 12. כדי העין שביניכם ישימו לב שההדפסה בקונסולה נראית מעט שונה מאשר כשמדובר במספר ולא במחרוזת טקסט. ההבדל נוצר בגלל השוני בין סוג המידע של מחרוזת טקסט לבין סוג המידע של מספר.

אם מחברים בין מספרים אזי החיבור יעשה בבדיקה כפי שהייתם מצפים:

```
let foo = 12;
let bar = 10;
let answer = foo + bar;
console.log(answer);
```

אפשר לבצע פעולות אפילו בשלב ההשמה. למשל:

```
let foo = 2 + 4 + 5;
console.log(foo);
```

התשובה שתזדפס תהיה 11. 2 ועוד 4 ועוד 5. בדומה לחיבור, אפשר לעשות פעולות חישור באמצעות הסימן "-". למשל:

```
let foo = 2;
let bar = 8;
let answer = foo - bar;
console.log(answer);
```

שימוש לב שההתוצאה כאן היא שלילית. אין בעיה עם מספרים שליליים גם בהוצאה. למשל:

```
let foo = -12;
let bar = -10;
let answer = foo + bar;
console.log(answer);
```

כאן התוצאה תהיה `-22`. `-10` ועוד `-12`.

ואפשר גם לבצע כפל בעזרת הסימן `**`. למשל:

```
let foo = 2;
let bar = 8;
let answer = foo * bar;
console.log(answer);
```

התוצאה כאן תהיה `16`. `2` כפול `8`.

אפשר גם לבצע חילוק בעזרת הסימן `/`.

```
let foo = 1;
let bar = 2;
let answer = foo / bar;
console.log(answer);
```

התוצאה כאן תהיה `0.5`. ג'אווהסקריפט לא מפחד משברים עשרוניים, כמובן, ואפשר להגיד לו גם שברים עשרוניים.

אפשר לעבוד גם עם חזקות. חזקות מסוימים בג'אווהסקריפט באמצעות `**`:

```
let foo = 10;
let bar = 2;
let answer = foo ** bar;
console.log(answer);
```

התוצאה כאן תהיה `100` כמובן. `10` בחזקת `2`.

בדומה לפעולות המוחדרות של מחרוזת טקסט שראיתם שאפשר להפעיל על משתנים מסווג מחרוזת טקסט, גם למשתנים מסווג מספר ישפעולות מוחדרות. למשל, פעולה המגבילה את המספר לאחר הנקודה העשונית. אם תחלקו 10 ב-3, תקבלו 3.3333333 עד דלא ידע. אפשר להעיף את הנקודות העשוניות. איך? באמצעות פעולה מיוחדת בשם `:Math.round()`:

```
let foo = 10;
let bar = 3;
let answer = foo / bar;
let finalAnswer = Math.round(answer);
console.log(finalAnswer);
```

הבה ננתן את קטע הקוד הזה.
 בשורה הראשונה יוצרים משתנה בשם `foo` ומכוונים לתוכו את המספר 10.
 בשורה השנייה יוצרים משתנה בשם `bar` ומכוונים לתוכו את המספר 3.
 בשורה השלישית יוצרים משתנה בשם `answer`. מה הוא מכך? `foo` חלקי `bar`, שזה בעצם 10 חלקי 3. מה התוצאה של התרגיל הזה? 3.3333333, שנמצאת כרגע במשתנה `answer`.
 בשורה הרביעית מפעילים את הפעולה `Math.round` על `answer` ומכוונים את התוצאה למשתנה `finalAnswer`. הפעולה `round` מעגלת את המספר. למה? ל-3. אם תדפינו את `finalAnswer` תקבלו 3.

הינה טבלה קצרה שמסכמת את כל הפעולות הבסיסיות שאפשר לעשות:

הטו שימושים בו כדי לעשות את הפעולה	סוג פעולה
+	חיבור
-	חיסור
*	כפל
/	חילוק
**	חזקה

מציאת השארית

אם אתם זוכרים מתקופת בית הספר היסודי, יש דבר צזה שנקרא שארית. כמה זה שיש חלקי ארבע? אחת ושארית שתיים. כמובן, החלק מהמספרשלם שהוא קטן מהמחלק. תשע חלקי שמונה זה אחת עם שארית אחת. תאמינו או לא, בתכונות יש עדנה לדברים שלומדים בכיתה

ד'. אפשר לחלק ולמצוא את השארית בעזרת סימן מיוחד – %. הסימן זהה נקרא על ידי המתכנתים "מודולו" (באנגלית modulus). אפשר לקרוא לו שארית:

```
let foo = 6;
let bar = 4;
let answer = foo % bar;
console.log(answer);
```

לוקחים משתנה foo ומצביעים בתוכו את הערך 6. במשתנה bar מצביעים את הערך המספרי 4. שואלים מה השארית של 6 חלקי 4. התשובה כאן היא המספר 2.

הפעולות המתמטיות שנלמדו עד כה – חיסור, חיבור, כפל, חילוק והעלאה בחזקה – נקראות אופרטורים (באנגלית operators). אפשר להציב את תוצאות האופרטורים ישירות בתחום המשתנה. למשל, אם רוצים להציב בתחום משתנה את התוצאה של 4 כפול 2, אין צורך ליצור משתנה שיכיל 4, משתנה שיכיל 2 ומשתנה שיכיל את המכפלה של שניהם, כלומר משווה זהה:

```
let foo = 2;
let bar = 4;
let answer = foo * bar;
console.log(answer);
```

במקרהו, אפשר לכתוב משווה זהה:

```
let answer = 2 * 4;
console.log(answer);
```

זה די מובן אם זוכרים שבסוףו של דבר, מאחורי כל משתנה עומד מידע, בין שמדובר במספר ובין שבטקסט.

אופרטורים מוקצרים

נניח שיש משתנה שרצים להוסיף לו רק ספרה אחת. אפשר להשתמש באופרטור מוקוצר להוספה או להחסרה. האופרטור להוספה הוא **+** והוא נראה כך:

```
let answer = 4;
answer++;
console.log(answer);
```

מה יש פה? מגדירים משתנה בשם answer והוא 4. עם האופרטור **++** מוסיפים לו 1. ערך המשתנה החדש הוא 5. באותו אופן כמו האופרטור **++** קיימ גם האופרטור **--** שמבצע את הפעולה הפוכה של חיסור:

```
let answer = 10;
answer--;
answer--;
answer--;
console.log(answer);
```

כאן למשל יוצרים משתנה ומציבים בתוכו את המספר 10. באמצעות האופרטור **--** מורידים ממנו 1. כיוון שחווזרים על התוצאה שלוש פעמים, המשתנה שווה ל-7. הבעה היא שזה מכוער... יש גם אופרטור קיצור שיכל להחסיר או להוסיף איזה מספר שרצים. כך, למשל, גם הקוד הזה ידפיס 7.

```
let answer = 10;
answer -= 3;
console.log(answer);
```

גם כאן יוצרים משתנה ומציבים בו 10. בעזרת האופרטור של החיסור המוקוצר מורידים ממנו 3. התשובה היא 7.

הינה טבלה חלקית של האופרטורים המקביצים הנפוצים ביותר:

משמעות	אופרטור מקוצר
foo = foo + 1;	foo++
foo = foo - 1;	foo--
foo = foo + 10;	foo+=10;
foo = foo - 10;	foo-=10;

לסיום הפרק יש לציין תכונה חשובה של השפה – ג'אווהסקריפט היא שפה שלחנית מאוד. אם מנסים לחבר בין משתנה מסווג טקסט למשתנה מסווג מספר היא לא תעיף שגיאה אלא תמיר באופן אוטומטי את המספר לטקסט ותבצע חיבור. כך למשל:

```
let foo = 1;
let bar = '1';
let baz = foo + bar;
console.log(baz);
```

תיתן את התשובה המדהימה מחרוזת טקסט של 11. למה? כי baz הוא מסווג טקסט. אם אתם כבר מכירים שפת תכנות, זה השלב שבו אתם מתמלאים בזעם או בתדמה. אבל כאמור ג'אווהסקריפט היא שפה שלחנית ותנסה להגיע לפשרה אם מדובר בחיבור בין טקסט למספר. בחיבור בין מספר לבין סוג המידע האחרים, תתקבל שגיאה של **NaN**, שהוא ראשי תיבות של **Not a Number**.

זה נראה שונה לעין בلتוי מיוםנות, אבל ג'אווהסקריפט היא שפה שטוגי המשתנים בה הם **implicit** – כלומר המנווע של השפה מנסה לנחש אותם ולא ממהר לזרוק שגיאה כמו בשפות אחרות. כאמור, משתמש לא מיומן עלול להתבלבל, ולמתכוונים בשפות אחרות זה נראה כאוטי, אבל ברגע שמתרגלים – מדובר בתוכונה נוחה מאוד.

תרגיל:

צרו שלושה משתנים שיכילו את המספרים 2, 4 ו-6 והציגו את תוצאה החיבור שלהם.

פתרון:

```
let foo = 2;
let bar = 4;
let baz = 6;
let answer = foo + bar + baz;
console.log(answer);
```

הסבר:

יוצרים שלושה משתנים ושמים בהם במספרים באופן מכוון כפי שלמדנו. שלושת המספרים מחוברים והתוצאה נכנסת למשנה `answer`, שבתורו מודפס.

תרגיל:

צרו משתנה, הציבו בתוכו את המספר 12 והדפiso אותו. באותו משתנה הציבו מחרוזת טקסט של 12 והדפiso אותה.

פתרון:

```
let foo = 12;
console.log(foo);
foo = '12';
console.log(foo);
```

הסבר:

כאשר מציבים מספר במשנה, לא משתמשים בגרשיים. כאשר מציבים מחרוזת טקסט משתמשים בגרשיים. ההדפסה נעשית באותו אופן, אך כאמור, ברוב הקונסולות תראו הבדל בין שתי ההדפסות שנועד לסמן שהדפסה אחת היא מספר והאחרת היא טקסט.

חשוב: צריך לשים לב שסוג המידע של המשתנה המשתנה וכדי להימנע מלהציג באותו משתנה סוג מידע שונים בשלבים שונים של ריצת תוכנית. זה מתכוון לטעויות בהמשך.

תרגיל:

צרו משתנה והציבו בתוכו את המספר 100. הדפיסו את השורש של המספר.

פתרונות:

```
let foo = 100;
let bar = 0.5;
let answer = foo ** bar;
console.log(answer);
```

הסבר:

מציבים 100 ו-0.5 בתוך משתנים. שורש, למי שלא זוכר מתמטיקה, הוא חזקה חצי. התשובה היא 100 בחזקת 0.5, ואני מתבביש בעצמי על זה שהכנסתי תחכום במתמטיקה לתרגיל כאן.

תרגיל:

צרו משתנה שיכיל את השארית של 10 חלקו 3.

פתרונות:

```
let foo = 10 % 3;
console.log(foo);
```

הסבר:

על מנת לחסוך במשתנים מציבים בתוך המשתנה foo את תוצאות הביטוי $10 \% 3$. התוצאה היא 1.

תרגיל:

צרו משתנה, הכניסו אליו מספר ובאמצעות אופרטור מקוצר הגדילו את המספר בעוד 1.

פתרונות:

```
let foo = 10;  
foo++;  
console.log(foo);
```

או

```
let foo = 10;  
foo += 1;  
console.log(foo);
```

הסבר:

האופרטורים המקוצרים מאפשרים להוסיף 1 בקלות. יוצרים משתנה ומציבים בו את המספר 10. השתמשתם באופרטור המקוצר ++ להוספה 1 בדילוק למשתנה זהה. אפשר להשתמש באופרטור המקוצר += כדי להוסיף 1.

פרק 4

סוגי מידע פרימיטיביים נוספים



סוגי מידע פרימיטיביים נוספים

בפרקים הקודמים למדנו על מידע מסווג מחרוזת טקסט ומספר. סוג מידע אלו נקראים "סוגי מידע פרימיטיביים". כשאומרים שישו ג'אווה פרימיטיבי לא מתכוונים לכך שהוא לא אוכל בסכין ובמצלג אלא שהוא ייחידת מידע בסיסית ולא מתוחכמת שבאה עם השפה. חוץ ממספר ומחרוזת טקסט יש עוד כמה סוגים מידע כאלה.

بولיאני

סוג מידע בוליאני הוא סוג מידע שמכיל אחד משני ערכי, `true` או `false`, כלומר אמת או שקר. כך מציבים אותו:

```
let foo = true;
console.log(foo);
```

שימוש לב ש-`true`, בדיקן כמו מספר, אינו מוקף בגרשיים. אם הוא היה מוקף בגרשיים הוא היה מידע מסווג מחרוזת טקסט. פעולה מיוחדת שיש לסוג מידע בוליאני היא המרת מחרוזת טקסט באמצעות הפעולה `toString()`:

```
let foo = true;
console.log(foo);
let bar = foo.toString();
console.log(bar);
```

מה קורה פה? קודם כל יוצרים משתנה ומוכניסים לתוכו את הערך `true`. הערך הוא ערך בוליאני. ה-`true` מגיע ללא גרשימים כיון שהוא מילה שומרה. אם תדפיסו אותו, תראו שהקונסולה מראה אותו כ-`true`. מהרגע שהמשתנה זהה קיבל ערך בוליאני, הוא מסווג מידע בוליאני, ועל מידע בוליאני אפשר להשתמש בפעולות מיוחדות. אחת מהן היא `toString()`, שגורמת לערך הבוליאני להפוך למחרוזת טקסט. השורה השלישית מכניסה את הערך של המשתנה הבוליאני אל המשתנה `bar` ואז מדפיסים אותו. חדי העין יבחינו שיש הבדל בקונסולה בין הדרישה הראשונה לשניה. הסיבה היא שבהדרישה הראשונה המשתנה הוא בוליאני ובהדרישה השנייה הוא מחרוזת טקסט.

בהמשך הספר אשוב למשתנים בוליאניים ואדון במצבים שבהם כדאי להשתמש בהם.

משתנה לא מוגדר

"לא מוגדר" (`undefined`) הוא הערך הראשוני של כל משתנה עוד לפני ההגדירה שלו. אם מגדירים משתנה ואז מדפיסים אותו, רואים בקונסולה "`undefined`":

```
let foo;
console.log(foo);
```

אפשר לאפס משתנה בעזרת המילה השומרה `undefined`:

```
let foo = false;
foo = undefined;
console.log(foo);
```

אין פעולות שאפשר להצמיד ל-`undefined`.

ריך

סוג מידע פרימיטיבי נוסף הוא "ריך" (null). כאמור, המשתנה לא מכיל דבר. שימושו לב שלא מדובר ב-0 ולא במחרוזת ריקה, אלא בכלום. ריך. ההגדרה נעשית באמצעות המילה השמורה `null`:

```
let foo = null;
console.log(foo);
```

כאן ההדפסה תראה `null`. אפשר להפוך המשתנה ל-`null` בכל רגע נתון באמצעות המילה השמורה. כך הופכים המשתנה בוליאני ל-`null`:

```
let foo = false;
foo = null;
console.log(foo);
```

שימוש לב: ההבדל בין `undefined` ל-`null` לא נראה כרגע ממשמעותי אך הוא ממשמעותי מאוד. `undefined` ממשמעותו משתנה שהוגדר אך לא אוטח בערך כלשהו. `null` ממשמעותו משתנה שהוגדר ואוטח כ-`null`. כדי לזכור שמדובר בשני סוג מידע שונים לחלווטין. המידע הזה יהיה חשוב בהמשך הדריך. כדי לציין שיש כאן שוחשיים שמדובר בטעות בעיצוב השפה וש-`null` לא אמור להתקיים.

Symbol

מדובר בסוג חדש של מידע פרימיטיבי שנכנס לשפת JavaScript בשנת 2015 (ES2015). **Symbol** מאפשר ליצור משתנה בעל ערך ייחודי לחלווטין. הערך זהה הוא לא מחרוזת טקסט, לא מספר ולא שהוא אחר:

```
let foo = Symbol();
let bar = Symbol();
console.log(foo);
console.log(bar);
```

במקרה הזה, `foo` ו-`bar` הם משתנים שקיבלו לכארה אותו ערך, אבל הם שונים לחלווטין. (`Symbol` נותן למשתנה ערך ייחודי שאפשר להשתמש בו בהמשך. במקרה הזה כל הסיפור נשמע מאד ערטילאי, אבל אני מבטיח שהוא יתבהר בהמשך. מה שכנן — מדובר בסוג מידע פרימיטיבי שאינו שונה במשמעותו מכל סוג מידע שדיברנו עליו עד כה.

שימוש לב: הוא סוג מידע שנכнес רק בנסיבות האחרונות של ג'אווהסקריפט ושימושו כאשר רוצים ליצור מזהה ייחודי. חשוב להכיר אותו.

מציאת הסוג של המשתנה

בכל רגע נתון אפשר למצוא את סוג המשתנה באמצעות האופרטור `typeof`. בדוגמה לאופרטורים המתאימים לסוג המידע מספר, גם `typeof` הוא אופרטור, אבל אופרטור שעבוד על כל סוג המידע ומחזיר מחרוזת טקסט המציג את סוג המידע של המשתנה:

```
let foo = 'Hello World';
let bar = typeof foo;
console.log(bar);
```

בקוד שלעיל נוצר משתנה בשם `foo` והוצבה בו מחרוזת הטקסט "Hello World". מהרגע זהה, `bar` יש סוג מידע טקסט. באמצעות האופרטור `typeof`, המשתנה `bar` מכיל את המידע על סוג המידע של `foo`. אם תדפיסו אותו תראו `string`.
לעתים `typeof` עלול להטעות וудיף להשתמש בו אך ורק לסוגי מידע פרימיטיביים.

תרגיל:

צרו משתנה בוליאני והכניסו לו את הערך `false`. הדפיסו אותו, המירו אותו למחרוזת טקסט והדפיסו אותו שוב.

פתרון:

```
let foo = false;
console.log(foo);
let bar = foo.toString();
console.log(bar);
```

הסבר:

המשתנה הבוליאני `foo` נוצר עם הערך `false` ללא גרשים. אם הייתם יוצרים אותו עם גרשים היה נוצר משתנה של מחרוזת טקסט. המשתנה הבוליאני יכול להפעיל על עצמו פעולה מיוחדת של `toString()` שמירה את הערך שלו למחרוזת טקסט. מחרוזת הטקסט זהו כמובן המשתנה אחר בשם `bar`, שהוא כמובן כבר מחרוזת טקסט לכל דבר ועניין. בהדפסה השנייה הוא מודפס כמחרוזת טקסט.

תרגיל:

צרו משתנה, הכניסו לו ערך בוליאני כלשהו והפכו אותו ללא מוגדר.

פתרון:

```
let foo = true;
```

```
foo = undefined;  
console.log(foo);
```

הסביר:

יצירת הערך הבוליאני נעשית באמצעות המילה השמורה `true`, שיצירת משתנה מסוג בוליאני שלו ערך בוליאני `true`. לאחר מכן, השמה של המילה השמורה `undefined` הופכת את המשתנה ללא מוגדר, ואם תנסה להדפיסו תקבלו `undefined`.

תרגיל:

צרו משתנה מסוג `undefined` והמירו אותו למשתנה שאינו בו ערך.

פתרונות:

```
let foo;  
console.log(foo);  
foo = null;  
console.log(foo);
```

הסבר:

כשיצרים משתנה ולא מוכניסים לו ערך, הוא כבר מסוג `undefined`. כשמוכניסים לו ערך `null` באמצעות המילה השמורה `null` הוא הופך למוגדר אך הוא חסר ערך. שימוש לב שהמילה `null` היא מילה שומרה שאינה מוקפת בגרשיים. אם היא הייתה מוקפת בגרשיים היינו מקבלים מחזרזת טקסט שערכה הוא `null`.

תרגיל:

צרו משתנה `foo` והכניסו לו מחרוזת טקסט. הדפיסו את `typeof` של מחרוזת הטקסט בקונסולה. לאחר מכן הכניסו למשנה מספר, וגם כאן הדפיסו את `typeof` כדי לבדוק את סוג המידע של המשתנה `foo`. עשו זאת עם שישה סוגי מידע שונים.

פתרונות:

```
let foo;
let bar = typeof foo;
console.log(bar);
foo = 1;
bar = typeof foo;
console.log(bar);
foo = '1';
bar = typeof foo;
console.log(bar);
foo = true;
bar = typeof foo;
console.log(bar);
foo = Symbol();
bar = typeof foo;
console.log(bar);
foo = null;
bar = typeof foo;
console.log(bar);
```

הסבר:

בתחילת הקוד יוצרים משתנה `foo` ובו דקיקים אותו באמצעות `typeof` שערך נכנס ל-`bar`. אפשר לראות שבהתחלת `foo` הוא `undefined`. כশמכניםים לתוכו מספר, שונות מהמספר בכר שיש סיבוב המספר `number` וכשמכניםים לתוכו מחרוזת טקסט, שונה מהילה השמורה `true`, סוג הוא גרשים, הוא סוג `string`. כশמכניםים ערך בוליאני, שהוא המילה השמורה `true`, סוג הוא `boolean`, וכשמכניםים `Symbol` הוא הופך לסוג `symbol`. לבסוף מכניםים את הערך `null`. ה-`typeof` שלו הוא `object` מסיבות היסטוריות שלא נכנס אליו פה.

הערות

ברוב המקרים רוצים לכתוב הערות בקוד, טקסט שיסביר כל מיני דברים על הקוד בלי שיירונדר. למה צריך את זה? לפחות כדי לכתוב הסברים לזמן מאוחר יותר, לעתים עבור מתכנים אחרים או לעצמכם ולפעמים לצרכים אחרים.

יש שני סוגי הערות בג'אווהסקריפט. אם רוצים הערה של שורה אחת, אפשר להשתמש בשני התווים `//`:

```
// let a = 5;
console.log(a);
```

כאן למשל התוצאה תהיה `undefined`, כי ההגדירה של `a` נמצאת בהערה. מקובל בהערות לשמר על רוח בין שני תווים הערה `//` להערה עצמה, אך אין חובה לעשות זאת.

לעתים רוצים לכתוב הערה בכמה שורות. לפיכך משתמשים בתווים `/*` ו`*/`. בתחילת גוף הערה כתבים `*` ובסוף `*/`:

```
/*
let a = 10;
console.log(a);
*/
```

במקרה הזה דבר לא יודפס כי גם הגדרת המשתנה וגם ההדפסה שלו נמצאות בהערות.

בהמשך משתמש בהערות על מנת להציג את הערבים השונים. כך למשל אם ארצה לציין שבשלב מסוים משתנה שווה לערך כלשהו, אكتب את זה בהערה במקום לכתוב `console.log`. למשל:

```
let foo = 10;
foo++; // 11
```

כאן נוצר משתנה `foo` והוצב בו הערך המספרי 10. באמצעות האופרטור המקוצר `++`, שכבר למדנו עליון, נוסף 1 למשתנה זהה. על מנת לציין את הערך החדש נווספה הערה.

תרגיל:

בדומה לתרגיל הקודם, צרו משתנה ובאמצעות **typeof** שנו את סוג המידע שלו שש פעמים לשישה סוגים שונים. כתבו בהערות את סוג המידע

פתרונות:

```
let foo;  
let bar = typeof foo; // undefined.  
foo = 1;  
bar = typeof foo; // number.  
foo = '1';  
bar = typeof foo; // string  
foo = true;  
bar = typeof foo; // boolean  
foo = Symbol();  
bar = typeof foo; // Symbol  
foo = null;  
bar = typeof foo; // object
```

הסבר:

כל מה שמיין ל-*//* נחשב להערה. כך קל מאד להבין מה *bar* מכיל.

פרק 5

בחרת זרימה - מושפי תנאי



בקורת זרימה – משפטי תנאי

לפעמים רוצים שחלק מסוים בקוד יפעל בהתאם למשתנה, למשל שהקובנולה תדפיס משהו אם המשתנה בעל ערך מסוים. כך למשל אם יש משתנה שהוא מספר, אפשר לבדוק אם הוא מספר מסוים ולעשות משהו.

למשל, הנה נניח משתנה `foo` מסוים. אם הוא שווה ל-5, יודפס בקונולה "This is five". אם אין עושים את זה?

```
if (foo === 5) {
  console.log('This is five');
}
```

משפט התנאי `if` מאפשר לשאל שאלת, במקרה זה אם `foo` שווה ל-5. סימן השווון הוא `==` והוא בודק אם מדובר במספר המבוקש, במקרה זה 5. אם התנאי נכון כל מה שקרה בסוגרים المسؤولים מתרחש. אם לא, לא קורה כלום.

שימוש לב: `==` נקרא **אופרטור**. כן, בדיק כמו האופרטורים של החיבור, החיסור, הכפל ודומיהם, שלמדנו בפרק המספרים. האופרטור הזה הוא אופרטור של השוואה, ובניגוד לאופרטור נומרי (של מספר) הוא מחזיר ערך בוליאני. כשלמדנו לעילו הוא היה ערטילאי מאד. עכשו הוא קצת יותר הגיוני, אבל רק קצת. בקרוב הוא יחקר לעומק. בדוגמה שלhallן, אם מדובר במספר מסוים אזי בקונולה יודפס "זה 5", ואם לא, בקונולה יודפס "זה לא 5". אם פה זה עושים את זה בנסיבות, באמצעות המילה השמורה `else`:

```
if (foo === 5) {
  console.log('This is five');
} else {
  console.log('This is NOT five');
}
```

באמצעות האופרטור ההשוואתי `==` בודקים אם `foo` שווה ל-5. אם כן, מתרחש מה שכתוב בסוגרים المسؤولים מיד אחריו `-if`. אם לא, מתרחש מה שכתוב בסוגרים المسؤولים אחרי `-else`. כמו שאפשר להשוות מספרים, אפשר להשוות גם מחרוזות טקסט. למשל:

```
let foo = 'five';
if (foo === 'five') {
  console.log('This is five');
} else {
  console.log('This is NOT five');
}
```

באמצעות האופרטור ההשוואתי `==` בודקים אם המשתנה `foo` שווה למחרוזת הטקסט `five`.

אם כן – הקוד שיש בתחום הסוגרים המסתולסים הראשונים יופעל והקונסולה תקבל "This is five". אם לא – רק הקוד שבסוגרים המסתולסים לאחר המילה `else` יופעל והקונסולה תקבל "This is NOT five".

אפשר ליצור משפט תנאי שבודקים כמה תנאים במקביל. למשל, אם המספר הוא 5 תדפיס הקונסולה שהמספר הוא 5. אם המספר הוא 6 תדפיס הקונסולה שהמספר הוא 6, ואם הוא לא 5 ולא 6 יהיה תדפיס שהוא לא זה ולא זה. איך עושים את זה? באמצעות שילוב של `else` ו-`if`:

```
let foo = 5;
if (foo === 5) {
  console.log('This is 5');
} else if (foo === 6) {
  console.log('This is 6');
} else {
  console.log('This is not 5 or 6');
}
```

כרגע, משפט התנאי `if` בודק אם `foo` שווה ל-5. אם כן, קטע הקוד שבתוך הסוגרים המסורסים הראשונים יופעל. אבל כאן יש גם `if` עם עוד תנאי, שבודק אם `foo` שווה ל-6. בסופו של דבר יש תנאי שתופס את הכל. אפשר לשלב כמה `if` שורצים, למשל כמו בקוד הבא:

```
let motherNumber = 1;
if (motherNumber === 1) {
  console.log('Sarah');
} else if (motherNumber === 2) {
  console.log('Leah');
} else if (motherNumber === 3) {
  console.log('Rachel');
} else if (motherNumber === 4) {
  console.log('Rivka');
} else {
  console.log('Not 1-4 number');
}
```

כאן מקבלים מספר מ-1 עד 4 ומדפיסים בהתאם בקונסולה את השם של אחת האימהות מהתנ"ר. שימו לב שיש תנאי `if` מרובים בקוד זהה, שבודקים את המספר שוב ושוב, בכל פעם נגד תנאי אחר. אם המספר הוא לא 1, 2, 3 או 4, יופעל ה-`else` האחרון. חשוב לציין שגם `else` תוסיפו סתם `else` באמצע, לעולם לא תגיעו ל-`if` שטוף אחריו.

עוד משהו חשוב לציין הוא שצריך להיזהר מהשימוש בתוך משפט תנאי. ג'אווהסקריפט היא שלחנית מדי, ואם כתבו `u=x` במקומות `u==x` הקוד שלכם יעבד ולא יזרוק שגיאה – זו הסיבה שבאופרטורים השוואתיים צריך לשימר לב היטב מה כתבים ו לעולם לא לכתוב `=` אחד בלבד.

אופרטורים השוואתיים נוספים

כמו `==` שבודק השוואת טהורה, יש אופרטורים השוואתיים נוספים, למשל `==!` (סימן הקראיה משמאלי לשני סימני השווה) שבודק אי-שווין ויפעל רק אם המשתנה לא שווה. למשל:

```
let foo = 4;
if (foo !== 1) {
  console.log('This will work, foo is not 1')
};
```

האופרטור `==!` בודק אם `foo` אינו שווה ל-1. במקרה זה הוא שווה ל-4, וקטע הקוד בתור הסוגרים المسؤولים יפעל. אושר גודל! הינה עוד דוגמה:

```
let foo = 'not one';
if (foo !== 'one') {
  console.log('This will work, foo is not one');
};
```

כאן בודקים מחרוזת טקסט. האם `foo` שונה מ-`'one'`? במקרה זה כן, ולפיכך הקטע שבתו הסוגרים المسؤولים יופעל.

יש גם אופרטורים שבודקים גדול/קטן מ-. כך למשל אפשר לבדוק אם המשתנה שלנו גדול או קטן ממספר מסוים:

```
let foo = 10;
if (foo > 1) {
  console.log('Great than 1');
} else {
  console.log('Less than 1');
};
```

במשפט התנאי בודקים אם `foo` גדול מ-1 באמצעות אופרטור אי-השוון `>`, ממש כמו בתרגילי חשבון של כיתה א'. כיוון שבמקרה הזה `foo` שווה ל-10 ו-10 גדול מ-1, המשפט שבתו הסוגרים المسؤولים יופעל.

מה לפि דעתכם יודפו בקונסולה אם תרצו את הקוד הבא?

```
let foo = 5;
if (foo > 5) {
    console.log('Greater than 5');
} else {
    console.log('Less than 5');
};
```

התשובה היא "Less than 5". מדוע? משפט התנאי בודק אם foo גדול מ-5 ואם כן מדפיס את "5". אם לא, יודפס "Less than 5". כיוון ש-foo הוגדר כשווה ל-5, 5 אינו גדול מ-5 ולפיכך התנאי לא מתקיים.

air פותרים את העניין? אפשר להוסיף תנאי נוסף בעזרת if else:

```
let foo = 5;
if (foo > 5) {
    console.log('Greater than 5');
} else if (foo === 5) {
    console.log('Equal to 5');
} else {
    console.log('Less than 5');
};
```

אפשר גם להשתמש באופרטור נוסף שמשמעו גדול או שווה:

```
let foo = 5;
if (foo >= 5) {
    console.log('Bigger or equal to 5');
} else {
    console.log('Less than 5');
};
```

can אפשר להשתמש באופרטור \leq שמשמעו גדול מ- או שווה ל-. במקרה זה גדול מ-5 או שווה לו, וכיוון ש-foo שווה ל-5 התנאי מתממש.

כמו שיש אופרטור גדול או שווה, יש גם אופרטור קטן או שווה. הינה דוגמה:

```
let foo = -10;
if (foo <= 5) {
    console.log('Smaller or equal to 5');
} else {
    console.log('Greater than 5');
};
```

משפט התנאי בודק אם `foo` קטן מ-5 או שווה לו. במקרה זה, כיוון `sh-foo` שווה ל-10-, התנאי מתממש, ועל הקונסולה תראו "Smaller or equal to 5" בתגובה.

הבה נבחן את כל האופרטורים ההשוואתיים שהכרתם:

תפקיד	שם האופרטור ההשוואתי
שווין	<code>==</code>
אי-שווין	<code>!=</code>
גדול מ-	<code>></code>
קטן מ-	<code><</code>
גדול מ- או שווה ל-	<code>>=</code>
קטן מ- או שווה ל-	<code><=</code>

הסבר מאחרי הקלעים בקשר לאופרטורים השוואתיים:
האופרטורים ההשוואתיים בעצם מתרגםים את הביטוי שמעבירים להם לערך בוליאני, כלומר אמת או שקר. למשל:

```
let foo = 4;
foo === 4; // true
```

זאת אומרת שכמו שהאופרטור הנטומי מוסיף או משנה מספר, האופרטור השוואתי מוחזיר תמיד `true` או `false`, וזאת בלי קשר למשפט תנאי, אפילו אם סתם זורקים מספרים. **למשל:**

```
2 === 2; // true
2 !== 2; // false
2 > 2; // false
2 <= 2; // true
```

אפשר להחזיר את התוצאות של האופרטור השוואתי אל תוך משתנה, למשל אם כותבים את הקוד הבא:

```
let foo = 4 === 4;
console.log(foo);
```

אפשר לראות בקונסולה שזה `true`. משפט התנאי לא מתענין באופרטורים, במשתנים או בקשר אחר. בסופו של דבר, משפט התנאי מתענין אם מעבירים לו `true` או `false`. כך למשל:

```
if (true) {
  console.log('Will always work');
} else {
  console.log('Will NEVER work');
}
```

בסופו של דבר, כל משפט תנאי בודק אם יש לו `true` או `false`. אפשר ליצור את ה-`true/false` האלו באמצעות משתנה בוליאני או אופרטור השוואתי או אפילו לכתוב `true` כמו בדוגמה לעיל. או `false`. נשמע מוזר, אני יודע, אבל זה חשוב מאוד להבנה של משפט התנאי. בגדול, `if` ו- `else` קובעים את זרימת הקוד באמצעות משתנים בוליאניים.

אופרטורים לוגיים

האופרטורים הלוגיים מרחיבים את האופרטורים ומאפשרים ליצור תנאים מורכבים יותר. למשל, אם תרצה שתנאי מסוים יתמשח אם מספר מסוים יהיה גדול מ-10 או קטן מ-0. אין עושים את זה? באמצעות האופרטור הלוגי "או" שנכתב כ-|||. כך לדוגמה:

```
if (foo > 10 || foo < 0) {
    console.log('This number is larger than 10 OR less than 0')
} else {
    console.log('This number is between 0 to 10.');
}
```

בתוך הסוגרים העגולים יש שני תנאים עם אופרטורים של השוואה, וביניהם האופרטור הלוגי ||| שמשמעותו "או" – או זה או זה.

הבה נדגים זאת שוב במחuzeות טקסט. נניח שתרצה לבדוק אם משתנה מסוים שווה למחuzeות הטקסט "Sunday" או "sunday". איך עושים את זה? ממש בפשטות. צרכיים שאחד משני התנאים האלה יתמשח:

```
foo === 'sunday';
או:
foo === 'Sunday';
```

ممמשים את ה"או" באמצעות אופרטור לוגי של "או", ש כאמור נכתב |||. הינה, ממש ככה:

```
if (foo === 'sunday' || foo === 'Sunday') {
    console.log('Yay! Sunday')
} else {
    console.log('not Sunday');
}
```

אפשר גם לשלב מספר רב של תנאים. לדוגמה:

```
if (foo === 'sunday' || foo === 'Sunday' || foo === 1 || foo === true)
{
    console.log('Yay! Sunday')
} else {
    console.log('not Sunday');
}
```

כאן לדוגמה נראהות שהתנאי יתמשח אם foo יהיה שווה ל-sunday או ל-Sunday או ל-1 או ל-true.

מה קורה מאחורי הקלעים? האופרטור הלוגי יחזיר true אם אחד התנאים מתמשח, תוך שהוא מתעלם מכל השאר. כזכור, if עובד רק עם true או false וזה מה שהאופרטור הלוגי מחזיר לנו.

כדי לשים לב שההשוואה היא משماה לימין. אם התנאי הראשון מתממש, אין בדיקה של התנאי השני.

אופרטור לוגי נוסף הוא האופרטור "גמ", שבו אפשר לקבוע שני תנאים יתמלאו יחד. אם אחד מהם לא מתמלא, האופרטור הלוגי יחזיר `false` ומשפט ה-`if` לא יתקיים. האופרטור הלוגי "גמ" כתוב כך: `&&`. איך כתבים אותו בפועל? הנה נסתכל על הדוגמה הבאה:

```
if (foo === 1 && bar === 1) {
  console.log('foo is 1 and bar is 1');
} else {
  console.log('bar is not one OR foo is not one');
}
```

יש כאן שני משפטים תנאיים עם אופרטורים השוואתיים כמו שאנו מכירים וביניהם, כמו דבק, מחבר האופרטור הלוגי `&&` שמשמעותו "גמ", כלומר גם זה וגם זה. שני התנאים חיברים להתמלא על מנת שמשפט התנאי יחזיר `true` ויתממש. אם רק אחד מהם לא יהיה נכון, האופרטור הלוגי יחזיר `false` והתנאי ייכשל.

אפשר לשלב בין התנאים באמצעות סוגרים, ממש כמו במקרים אחרות מתמטיות שלומדים בבית הספר. כך למשל כתבים משפט שבו בודקים אם `foo` או `bar` שווים ל-1 וגם `baz` שווה ל-1:

```
if ((foo === 1 || bar === 1) && baz === 1) {
  console.log('foo is 1 OR bar is 1 AND baz is one');
} else {
  console.log('bar is not one AND foo is not one OR baz is not one');
}
```

שימוש לב שבתוך הסוגרים העגולים הראשונים יש אופרטור לוגי של "או", שיחזיר `true` אם אחד מהם נכון. בשלב הראשון של הרינדור, המנוע יבצע את הבדיקה של:

```
(foo === 1 || bar === 1)
```

הוא יפרש אותו כ-`true` או כ-`false`, אז ימשיך לשלב השני:

```
(true && baz === 1)
```

אם גם `baz` יהיה שווה ל-1, התנאי יתממש. אפשר להתרעע כמה שיותר עם סוגרים, למשל:

```
if ((foo === 1 || bar === 1) && (baz === 1 || baq === 1)) {
  console.log('foo is 1 OR bar is 1 AND baz is 1 or baq is 1');
} else {
  console.log('bar is not one AND foo is not one OR baz is not one
AND baq is not one');
```

כאן התנאי יתממש רק אם `foo` או `bar` הם 1, וגם אם `baz` או `baq` הם 1.

בדרך כלל, אם יש תנאים מורכבים כאלו, סימן שימושו לא בסדר בקood. קוד אמור להיות קרייא, ואף על פי שמשפט תנאי שיש בו מורכבות כזו הוא אפשרי, לא בטוח שמדובר בנוהג נכון.

אופרטור שלילי

אופרטור לוגי נוסף ושימושו מאד הוא **אופרטור לוגי שלילי**. האופרטור זהה קצת קשה להבנה, אך יש לו חשיבות רבה ושווא להשיקיע זמן בהבנתו. בגדול, האופרטור הלוגי הזה לוקח כל ערך בוליאני והופך אותו. למשל:

```
!true
```

יהיה בעצם `false`. אפשר להשתמש בו במשפט תנאי כדי לומר ההפך. כך למשל אם יש אופרטור השוואתי שבודק אם `foo` שווה ל-1, אפשר להפוך אותו והוא יבדוק אם `foo` שונה מ-1. כך זה קורה:

```
if (!(foo === 1)) {
  console.log('I am NOT 1');
} else {
  console.log('I am 1 actually');
}
```

הוא שימושי כשרוצים להפוך תנאים, ורוב המתכנים משתמשים באופרטור זהה על מנת להפוך תנאים מורכבים. כך למשל אם יש תנאי ובו אופרטור השוואתי שבודק אם משתנה גדול מ-10, אפשר להפוך אותו כדי לבדוק אם המשתנה קטן מ-10 באמצעות האופרטור `!`. הינה דוגמה:

```
let foo = 5;
if (!(foo > 10)) {
  console.log('foo is SMALLER than 10');
} else {
  console.log('foo is greater than 10');
}
```

אמנם יש שפות שימושיות שמשאלות שימוש בתנאי שלילי, אבל הפיכת תנאים נחשבת מנהג פסול מאד ולא הייתה ממליצה להשתמש בה. כדאי לדעת את זה על מנת להבין את פעילות האופרטור `!` שבסופו של דבר ממיר `false` ל-`true`, אבל מומלץ מאוד לכתוב את התנאים כמו שצරיך. למה כן משתמשים באופרטור זהה? הוא שימושי מאד לבדיקה אם משתנה מסוים הוא "ריק" ("empty") ולמנוע `else` מיותר. כשתצברו ניסיון בתכנות, תגלו שהמון פעמים צריך לבדוק אם משתנה מסוים הוא "ריק", וב"ריק" אני מתכוון ל:

```
let foo = '';
let foo = 0;
let foo = null;
let foo;
```

כלומר מחזצת ריקה, 0, `null` או אפילו משתנה שלא הוצב בו ערך. כל הערכים הללו ניתנים לבחינה בבת אחת בעזרת האופרטור `!`:

```
let foo; // can be null, '', 0 or false.
if (!foo) {
  console.log('foo has no value, null, 0, or empty string');
} else {
```

```
  console.log('foo has value');  
}
```

כל הערךים האלה לבסוף מומרים ל-`true` על ידי האופרטור הלוגי ! ואז משפט התנאי עובד. בדרך כלל רואים את האופרטור הלוגי בשימושים כאלה.

לסיכום, יש שלושה אופרטורים לוגיים:

סימן	שם האופרטור
<code> </code>	או
<code>&&</code>	גם
<code>!</code>	אופרטור שלילי

אופרטור תנאי

אופרטור תנאי, "if מוקוצר" או **ternary operator** הוא האופרטור המשמעותי האחרון שתלמדו. בגדול, הוא מאפשר להכניס ערכים למשתנים לפי תנאים מסוימים. כך למשל אם רוצים שהמשתנה `bar` יהיה שווה ל-9 אך ורק אם המשתנה `foo` שווה ל-10. אם המשתנה `foo` לא שווה ל-10, אך המשתנה `bar` יהיה שווה ל-0. אפשר לעשות את זה במשפט תנאי סטנדרטי:

```
if (foo === 10) {
    bar = 9;
} else {
    bar = 0;
}
```

אבל אפשר במקרים האלו, שבهم מבצעים הצבת ערך, לבצע את הפעולה שלעיל בעזרת משפט תנאי מוקוצר:

```
bar = foo === 10 ? 9 : 0;
```

עכשו הסבירו: משפט תנאי מוקוצר מתחילה בשם של המשתנה, סימן שווין ואז תנאי עם סימן שאלה. מיד אחריו סימן השאלה מופיע מה שנכנס לתוכו המשתנה אם התנאי מחזיר `true`, ואז קודתיים ומה שנכנס לתוכו המשתנה אם התנאי מחזיר `false`.

הבה נמשיך לדוגמה נוספת – אם רוצים לקבוע שיוצגו בקונסולה ההודעות "מותר לשתו אלכוהול" או "אסור לשתו אלכוהול" בהתאם לגיל, כך עושים את זה עם משפט תנאי מוקוצר:

```
let age = 18;
message = age >= 18 ? 'Drink!' : 'No drink for you!';
console.log(message); // Drink!
```

יש המשתנה בשם `message` שמקבל ערך באמצעות התנאי המוקוצר. התנאי הוא שהמשתנה `age` יהיה גדול מ-18 או שווה לו. אם כן, התנאי המוקוצר יחזיר ל-`message` את מחזורת `."No drink for you"`. אם לא, הוא יחזיר ל-`message` את מחזורת הטקסט `."Drink"`.

אופרטורים המשווים ערכים

יש סוג נוסף של אופרטורים שימושיים בהם פחות ואני באופן אישן לא משתמש בהם בכלל. עבור עליהם בקצרה כיון שהם מופיעים לא מעט פעמים, אבל השימוש בהם נחשב למנהג פסול.

נניח שיש את שני המשתנים הבאים:

```
let foo = 1;
let bar = '1';
```

האם הם שווים? אם עברתם על סוג מיידע, אז אתם יודעים שלא. המשתנה הראשון, `foo`, שווה למספר 1. המשתנה השני, `bar`, שווה למחרוזת הטקסט 1. על אף שהערך הוא כביכול אותו ערך, יש הבדל משמעותי בין מספר למחרוזת טקסט. למשל:

```
let foo = 1;
let bar = '1';
let answer;
answer = foo + foo; // 2
answer = bar + bar; // '11'
```

המשתנה `foo` שווה למספר 1, המשתנה `bar` שווה למחרוזת הטקסט 1. אם מוחברים שני מספרים שהם 1, התשובה היא 2. אם מוחברים שתי מחרוזות טקסט של 1, ג'אווהסקריפט מ לחבר את מחרוזות הטקסט ואז 1 ועוד 1 שווה ל-11. אבל כיון שאתה מתכווני ג'אווהסקריפט מנוסים, אתם כבר יודעים שהוא נכון רק אם ה-1 הוא מחרוזת טקסט.

כשאתם חמושים בתזכורת זו על מבני נתונים, עליה השאלה אם התנאי הזה יתממש:

```
let foo = 1;
let bar = '1';
if (foo === bar) {
  console.log('foo is equal to bar');
}
```

התשובה היא שההתנאי הזה לא יתמש, אף על פי שבשני המשתנים, `foo` ו-`bar`, יש את הערך 1. משתנה אחד הוא מספר והآخر הוא מחרוזת טקסט, ולפיכך הם שונים. אם רוצים להשוות את הערכים ולא את המספר, אפשר להשתמש באופרטורים השוואתיים שאינם משווים את סוג המידע אלא את הערך בלבד, כמו האופרטור ההשוואתי הוא `==` (שני סימני שווה ולא שלושה) או `!=` (במקום `==!`). כך למשל:

```
let foo = 1;
let bar = '1';
if (foo == bar) {
  console.log('value in foo is equal to value in bar'); // Will work.
}
```

התנאי שליל כן יתממש, כיוון שהאופרטור ההשוואתי **==** (להבדיל מ-**====**) מבצע את המראה של סוג המידע ואז מבצע את ההשוואה – שווי בין ערכים ולא בין סוג מידע.

מדובר בפרקטייה שנחשבת היום מאד לא מקובלת. נהוג להשוות גם בין סוג מידע ולא רק בין ערכים. זה נועד למונע בלבול והתאמות כושלות של תנאים. השתמשו תמיד בהשוואה של סוג. אבל לעיתים אפשר למצוא קודים שימושיים בהשוואה של ערכים בלבד. הטבלה הבאה מציגה את האופרטורים המשווים ערכים בלבד לעומת המשווים גם את סוג המידע:

אופרטור השוואתי של ערכים בלבד	אופרטור השוואתי
==	====
!=	!=

— **Falsy**-**Truthy**. איך בדיק נעשה התרגום זהה? בשביל זה צריך להכיר את המונחים **Falsy** ו-**Truthy**. **Falsy** מואמת ושקרית. כאשר ממירם משתנה לסוג מידע בוליאני, בודקים אם הוא הוא `false`. **Truthy** למעט הערכים הבאים: `0`, `0`, `false`, `""`, `null`, `undefined` ו-`NaN`.

כל מה שהוא **Truthy** מתרגם ל-`true`, כל מה שהוא **Falsy** מתרגם ל-`false`, בהמרה למשתנה בוליאני. וכך, שימושים בהמרות בוליאניות בהשוואה, כמו בדוגמה שלעיל, נתקלים בבעיה.

על מנת להדגים עד כמה זה בעייתי, שימו לב לקוד הבא:

```
console.log(0 == '') // true
```

למה זה ככה? כי מחרוזת ריקה מתרגם ל-`false` ו-`0` גם הוא מתרגם ל-`false`. זוכרים ששניהם **Falsy**? אם כך, שניים `false` ואז נעשית ההשוואה, וזה עלול להיות בעייתי. חשבו על מערכת שמקבלת קלט מספרי ועשה איתו דברים. אם נעשתה טעות ונשלח קלט ריק, ואז משתמשים בהשוואה רגילה, תהיה תקלה במערכת. היא תקבל קלט ריק ותחשב שהוא אפס, וההתנהגות בהתאם.

זו הסיבה שבגללה כדאי לעשות הכל ולהשתמש בהשוואה מדוקית של סוג מידע ולא של ערכים בלבד. תראו הרבה דוגמאות של אופרטור השוואה בין ערכים בראשת (לא בסוף הזה); השתדלו מאוד לכתוב נכון — מהניסיונות שלי זה רק יועיל לכם.

תרגיל:

צרו משפט תנאי שמתרכש אם המספר `foo` שווה ל-10.

פתרונות:

```
if (foo === 10) {
  console.log('This is ten');
}
```

הסביר:

המילה השמורה `if` בודקת את המשפט שיש בתוך הסוגרים העגולים. אם הוא נכון, החלק בתוך הסוגרים הממולאים ירוץ. אם לא אז לא. בתוך הסוגרים יש אופרטור לוגי מסוג שווון שבודק אם המשתנה הוא 10. קטע הקוד הזה, למשל, ידפיס את המשתנה:

```
let foo = 10;
if (foo === 10) {
  console.log('This is ten');
}
```

קטע הקוד הזה, לעומת זאת, לא ידפיס אותו:

```
let foo = 9;
if (foo === 10) {
  console.log('This is ten');
}
```

למה? כי 10 הוא לא 9.

תרגיל:

כתבו קוד שבודק את המשתנה `foo`. אם הוא שווה למחוזת הטקסט "`I am correct`" ואם לא, הקובנולה תדפיס "`He is NOT correct`"

פתרונות:

```
let foo = 'I am correct';
if (foo === 'I am correct') {
  console.log('He is correct');
} else {
  console.log('He is NOT correct');
}
```

הסביר:

מגדירים את המשתנה `foo`, ובאמצעות משפט תנאי והאופרטור ההשוואתי `==` בודקים אם המשתנה שווה ל-`"I am correct"`. אם כן, קטע הקוד בתוך הסוגרים المسؤولים הראשונים, כלומר זה:

```
{
  console.log('He is correct');
}
```

יעול. אם לא, קטע הקוד בתוך הסוגרים المسؤولים מיד לאחר ה-`else`, כלומר זה:

```
{
  console.log('He is NOT correct');
}
```

יעול.

תרגיל:

כתבו קטע קוד שמקבל מספר מ-1 עד 7 ומדפיס את היום בשבוע בקונסולה. אם המספר הוא לא בין 1 ל-7, תודפס בקונסולה הודעה שגיאה.

פתרון:

```
let day = 1;
if (day === 1) {
  console.log('Sunday');
} else if (day === 2) {
  console.log('Monday');
} else if (day === 3) {
  console.log('Tuesday');
} else if (day === 4) {
  console.log('Wednesday');
} else if (day === 5) {
  console.log('Thursday');
} else if (day === 6) {
  console.log('Friday');
} else if (day === 7) {
  console.log('Saturday');
} else {
  console.log('Not 1-7 number')
}
```

הסבר:

יש כאן אוסף של משפטים תנאי שבודקים את המשתנה `day` עם אופרטור השוואה. אם המשתנה `day` הוא 1, כל קטע הקוד שבין הסוגרים המסלולים שבאים אחר כר' יעבד, והקונסולה תדפיס את השם `Sunday`. מייד אחר כר' יש `if` שבודק אם המספר הוא 2, ואז הקונסולה תדפיס את השם `Monday`, וכך הלאה. שימו לב ל-`else if` ולסוגרים שבתוכם יש את התנאי ולסוגרים המסלולים שפועלים אך ורק אם התנאי נכון. בסוף הקוד יש `else` שמתקיים אם כל ה-`if` שהיו לפני לא עבדו, ורק אם הם לא עבדו.

תרגילים:

כתבו קוד שמקבל BMI. אם ה-BMI קטן מ-18 או שווה לו, מודפסת אזהרה שה-BMI נמוך מדי. אם ה-BMI גדול מ-25 או שווה לו, מודפסת אזהרה שה-BMI גבוהה מדי. אם לא, מודפסת הודעה שה-BMI תקין.

פתרונות:

```
let BMI = 20;
if (BMI <= 18) {
    console.log('BMI too low!');
} else if (BMI >= 25) {
    console.log('BMI too high');
} else {
    console.log('BMI OK');
};
```

הסביר:

מגדירים משתנה BMI ובודקים אותו בכמה משפטים תנאי. משפט התנאי הראשון בודק אם ה-BMI קטן מ-18 או שווה לו. משפט התנאי השני בודק אם ה-BMI גדול מ-25 או שווה לו וה-else בסוף תופס את כל מה שלא עונה לתנאים הללו, כלומר כל BMI שהוא בין 18 ל-25 (אך לא שווה להם).

תרגיל:

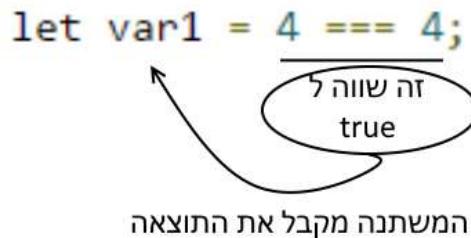
השתמשו בחמישה אופרטורים השוואתיים כדי להכניס `true` או `false` לחמישה משתנים שונים. הדפיכו אותם בקונסולה.

פתרונות:

```
let var1 = 4 === 4; // true
let var2 = 4 !== 4; // false
let var3 = 4 <= 4; // true
let var4 = 4 > 4; // false
let var5 = 4 < 4; // false
console.log(var1);
console.log(var2);
console.log(var3);
console.log(var4);
console.log(var5);
```

הסבר:

כפי שŁמדנו, האופרטורים ההשוואתיים מחזירים ערך בוליאני – `true` או `false`. פשוט יוצרים משתנה ומוכנסים לתוךו את תוצאות פועלות האופרטורים האלו, כיוון שהאופרטור הבוליאני מקבל כל מספר, אין זה משנה אם הוא בתוך משתנה או לא. הדיאגרמה הבאה תבהיר:



תרגיל:

כתבו משפט תנאי שבודק אם המשתנה `foo` שווה ל-`Thursday`, `Thursday!` או ל-`5`. אם כן יודפס בקונסולה המשפט `"Yay! Thursday!"`.

פתרון:

```
if (foo === 'Thursday' || foo === 'thursday' || foo === 5) {
  console.log('Yay! Thursday!')
}
```

הסבר:

כתבבים את התנאים כרגע ומחברים ביניהם באמצעות האופרטור הלוגי `"או"`, שכתוב `-||-`. יש בעצם שרשרת של תנאים שבין כל אחד מהם נמצא האופרטור הלוגי `||`. כך בעצם כתבים `"או"` ג'אווהסקריפט.

תרגיל:

כתבו משפט תנאי שבודק אם המשתנה `foo` שווה ל-`5` וגם אם המשתנה `bar` שווה ל-`Thursday`. אם שני התנאים האלו מתקיימים ביחד, יודפס בקונסולה המשפט `"Yay! Thursday! Thursday!"`.

פתרון:

```
if (bar === 'Thursday' && foo === 5) {
  console.log('Yay! Thursday!')
}
```

הסבר:

יש כאן שני משפטי תנאי עם אופרטורים השוואתיים:

```
bar === 'Thursday'
foo === 5
```

הראשון בודק אם `bar` שווה ל-`Thursday` והשני אם `foo` שווה ל-`5`. ביניהם מחבר האופרטור הלוגי `&&` ומשמעותו `"גם"`. יחד, משמעות הביטוי היא שהמשנה `bar` יהיה שווה ל-`Thursday` וגם שהמשנה `foo` יהיה שווה ל-`5`.

תרגיל:

כתבו משפט תנאי שבודק אם foo אמצעו null או 0 או מופיע בקונסולה "I am not here".

פתרון:

```
if (!foo) {
  console.log('I am not here');
}
```

או:

```
if (foo === 0 || foo === null) {
  console.log('I am not here');
}
```

הסבר:

הפתרון הראשון הוא מה שרוב מתכנתים הג'אווהסקריפט היו בחרים לעשות. באמצעות האופרטור הלוגי ! מקבלים true על foo אם הוא null, 0, מחוץ טקסט ריקה או false. הפתרון השני הוא אינטואיטיבי יותר למי שעושה את צעדיו הראשונים בשפה ומכל שני אופרטורים השוואתיים ואופרטור לוגי || שימושו היא "או": foo שווה ל-0 או foo שווה ל-null. שימושו לב ש-null היא מילה שומרה (לא מוקפת במירכאות). null הוא סוג מידע מיוחד שלמדו עליינו בפרקם הקודמים.

תרגיל:

כתבו משפט תנאי שמשווה בין:

```
let foo = 2;
let bar = '2';
```

ומצליח.

פתרון:

```
if (foo == bar) {
  console.log('equal');
}
```

הסבר:

נעשה שימוש באופרטור ההשוואתי == שמשווה בין הנתונים ולא בין סוג המידע. אף על פי שהסוג המידע של 2 הוא מספר ושל "2" הוא מחוץ טקסט, האופרטור ההשוואתי הזה יחזיר true.

פרק 6

SWITCH CASE



switch case

כמו משפט תנאי, גם משפט **switch case** מאפשר לבצע קוד בהתאם לערך שנמצא במשתנים. נניח שיש משפט תנאי שלוקח משתנה ובודק את המספר שלו. אם המספר הוא 1, הקונסולה מדפיסה "יום ראשון", אם המספר הוא 2, הקונסולה מדפיסה "יום שני" וכך הלאה. אם לא מדובר במספרים 1–7, הקונסולה מדפיסה הודעה שגיאה. איך זה נראה? ככה:

```
let day = 1;
if (day === 1) {
    console.log('Sunday');
} else if (day === 2) {
    console.log('Monday');
} else if (day === 3) {
    console.log('Tuesday');
} else if (day === 4) {
    console.log('Wednesday');
} else if (day === 5) {
    console.log('Thursday');
} else if (day === 6) {
    console.log('Friday');
} else if (day === 7) {
    console.log('Saturday');
} else {
    console.log('Not 1-7 number')
}
```

יש דרך אלגנטית יותר לכתוב משהו זהה. **switch case** מאפשר לבצע השוואה של משתנה מסוים לערך כלשהו (מספר, טקסט, ערך בוליאני וכו') ואז להריץ קוד בהתאם לערך ולספיק גם פעולות בריית מחדל.

נשמע מסביר? בכלל לא:

```
let foo = 1;
switch (foo) {
  case 1:
    console.log('Sunday');
    break;
  case 2:
    console.log('Monday');
    break;
  case 3:
    console.log('Tuesday');
    break;
  case 4:
    console.log('Wednesday');
    break;
  case 5:
    console.log('Thursday');
    break;
  case 6:
    console.log('Friday');
    break;
  case 7:
    console.log('Saturday');
    break;
  default:
    console.log('Not 1-7 number');
    break;
}
```

הקוד הזה הוא בדיקן כמו הקוד הקודם. פותחים בהצהרת **switch** שהיא מילה שומרה. **ב-*switch*** שמיים משתנה שיכל להכיל כל דבר ואחריו סוגרים מסוללים. בתוכם יש הצהרות **case**:

1 case לדוגמה שבמקרה שהערך הוא 1 – שימושו לב שמדובר בה מספר ולא במחרוזת טקסט – אם הערך עונה על התנאי הזה, כל הקוד שיש בין הנקודותים ל-*break* מתקיים. אם שום תנאי לא מתקיים, כל מה שקרה אחרי default מתקיים עד ה-*break*. לא חיברים את ה-*break* האחרון, אבל הציבו אותו בשביל הסדר הטוב.

break היא מילה שומרה שמשתמשים בה כדי לשבור את ה-*switch* ולצאת ממנו. חשוב לציין שה-*default* הוא חיוני תמיד, אפילו אם אתם חשבים שלא תגינו לשם. הקפה על default תמיד יכולה למנוע בעיות וצרות אחרות. הבה נדגים זאת בדוגמה אחרת – קוד שמחזיר שם של חודש לפי מספר. עם else if היה קשה מאוד לעשות את זה, אבל עם switch זה ממש קל:

```
let monthNumber = 5;
let monthName;
switch (monthNumber) {
```

```

case 1:
    monthName = 'January';
    break;
case 2:
    monthName = 'February';
    break;
case 3:
    monthName = 'March';
    break;
case 4:
    monthName = 'April';
    break;
case 5:
    monthName = 'May';
    break;
case 6:
    monthName = 'June';
    break;
case 7:
    monthName = 'July';
    break;
case 8:
    monthName = 'August';
    break;
case 9:
    monthName = 'September';
    break;
case 10:
    monthName = 'October';
    break;
case 11:
    monthName = 'November';
    break;
case 12:
    monthName = 'December';
    break;
default:
    monthName = 'n/a';
    break;
}
console.log(monthName);

```

אפשר לכתוב יותר משורה אחת. כך למשל אפשר גם להכניס ערך למשתנה "עונה" כדי לקבוע אם החודש הוא בקיץ או בחורף:

```

let monthNumber = 5;
let monthName;
let season;
switch (monthNumber) {
    case 1:

```

```
monthName = 'January';
season = 'Winter';
break;
case 2:
    monthName = 'February';
    season = 'Winter';
    break;
case 3:
    monthName = 'March';
    season = 'Winter';
    break;
case 4:
    monthName = 'April';
    season = 'Winter';
    break;
case 5:
    monthName = 'May';
    season = 'Summer';
    break;
case 6:
    monthName = 'June';
    season = 'Summer';
    break;
case 7:
    monthName = 'July';
    season = 'Summer';
    break;
case 8:
    monthName = 'August';
    season = 'Summer';
    break;
case 9:
    monthName = 'September';
    season = 'Summer';
    break;
case 10:
    monthName = 'October';
    season = 'Winter';
    break;
case 11:
    monthName = 'November';
    season = 'Winter';
    break;
case 12:
    monthName = 'December';
    season = 'Winter';
    break;
default:
```

```

        monthName = 'n/a';
        break;
    }
console.log(monthName); // May
console.log(season); // Summer

```

גם פה, כמו בקטעי הקוד הקודמים, לוקחים משתנה מסוימת ושמים אותה בתוך ה-**switch**. **switch** היא כאמור מילה שומרה שמודיעה למנוע של ג'אווהסקריפט שיש פה תנאי **case**. מיד אחרי ה-**switch** והמשתנה שנמצא בסוגרים העגולים יש סוגרים מסוימים, המכילים תנאים שנקבעים **case**. כל **case** מכיל ערך שהמשתנה נבחן מולו. אם המשתנה זהה לערך זהה, כל הקוד מהנקודות ועד המילה השומרה **break** עובד.

אפשר לקבץ יחד כמה תנאים. כך, למשל, אם רוצים להכניס את ערך העונה לפי מספר החודש, במקום לעשות משהו זהה (שבהחלט יעבדו):

```

let monthNumber = 5;
let season;
switch (monthNumber) {
    case 1:
        season = 'Winter';
        break;
    case 2:
        season = 'Winter';
        break;
    case 3:
        season = 'Winter';
        break;
    case 4:
        season = 'Winter';
        break;
    case 5:
        season = 'Summer';
        break;
    case 6:
        season = 'Summer';
        break;
    case 7:
        season = 'Summer';
        break;
    case 8:
        season = 'Summer';
        break;
    case 9:
        season = 'Summer';
        break;
    case 10:
        season = 'Winter';

```

```

        break;
    case 11:
        season = 'Winter';
        break;
    case 12:
        season = 'Winter';
        break;
    default:
        monthName = 'n/a';
        break;
}
console.log(season); // Summer

```

אפשר לעשות משהו זהה:

```

let monthNumber = 5;
let season;
switch (monthNumber) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 10:
    case 11:
    case 12:
        season = 'Winter';
        break;
    case 5:
    case 6:
    case 7:
    case 8:
    case 9:
        season = 'Summer';
        break;
    default:
        monthName = 'n/a';
        break;
}
console.log(season); // Summer

```

זה קורה כי אם לא משתמש במילה השמורה `break`, הקוד לא יעצור ואפשר להכנס כמה תנאים שרצויים. נשאלת השאלה, למה להשתמש בזה? אחרי הכל, בעזרה `else if` אפשר לכתוב את הקוד שלייל בלי ללמידה `switch case` מעיך. זו טענה ולידית, אבל כאן נכנס הנימוק של קוד קרייא ואלגנט. המטרה שלכם כמתכנתים היא לכתוב קוד קרייא שקל לכל אחד (גם לכם) להבין. שימוש מרובה ב-`else if` הוא מבלב וקשה לקריירה ולתיקון. בלי מעת מקרים, `switch case` יכול להקל את הקריירה ומשתמשים בו הרבה. לפיכך כדאי וצריך להכיר אותו ולהשתמש בו בכל פעם שיש יותר משלושה `else if`. זה לא כלל ברחל, אלא רק במקרה של

תנאי לא מסובך מדי – אתם לא רוצים להתחיל להכין משפטי תנאי והצבות בתוך ה- `switch` `case` `שלכם`.

תרגיל:

באמצעות switch, כתבו קוד ש לוקח שם של יום ומדפיס בקונסולה את מספר היום.

פתרונות:

```
let foo = 'Sunday';
switch (foo) {
  case 'Sunday':
    console.log(1);
    break;
  case 'Monday':
    console.log(2);
    break;
  case 'Tuesday':
    console.log(3);
    break;
  case 'Wednesday':
    console.log(4);
    break;
  case 'Thursday':
    console.log(5);
    break;
  case 'Friday':
    console.log(6);
    break;
  case 'Saturday':
    console.log(7);
    break;
  default:
    console.log('Not valid name');
    break;
}
```

הסביר:

cotבבים את המילה השמורה switch, שמקבלת משתנה בסוגרים העגולים. המשתנה זהה "יבחן" כנגד ערכים שונים. פורטט הבדיקה הוא זהה:

case value:

המילה השמורה case ומיד אחרייה מגע הערך. אם הערך זהה תואם את הערך שיש במשתנה שנמצא בסוגרים העגולים ליד ה-switch, הקוד שמופיע מיד אחרי הנקודותים ועד ה-break יירוץ. במקרה זהה מדובר בהדפסה בקונסולה. שימו לב ששמתי ערך דיפולטיבי שמודפס אם המשתנה יש ערך שלא צינתי בምורש ב מבחנים. זה נעשה באמצעות התנאי default.

פרק 7

הבושים

קבועים

עד כה למדנו על משתנים, כלומר על אבני בניין בסיסיות שיכולים לקבל ערכים מסוימים שונים (מספרים, מחרוזת טקסט וכו'). מגדירים אותם פעם אחת ויכולים לשנות אותם בהמשך:

```
let myVar = '';
myVar = 10;
myVar++;
myVar = undefined;
```

כأن למשל מוגדר משתנה בשם `myVar` וmoוצבת בתוכו מחרוזת טקסט ריקה. בהמשך מוצב בתוכו המספר 10. מעלים את המספר ב-1 באמצעות האופרטור `++` ולבסוף הופכים את המשתנה `to undefined`, סוג מידע של מדנו עליי קודם לכך.

כפי שאפשר ליצור משתנים, אפשר גם ליצור קבועים. בנויגוד למשתנים, קבועים כשם כן הם – **קבועים** (באנגלית `Constants`) ואי-אפשר לשנותם אם הם מסווג מידע פרימיטיבי (מספר, מחרוזת טקסט, `null`, `undefined`, `symbol`). את הקבוע מגדירים באמצעות המילה השמורה `const` באופן הבא:

```
const MY_CONSTANT = 10;
```

קונבנצייה אחת שהיא משתמשים כדי להגיד אוטיות גדולות עם קו תחתון שופריד בין מילים, אבל ראוי לציין שימוש לא חובה לעשות את זה. אפשר להגיד קבוע בדיקן כמו כל משתנה אחר. אם מנסים לשנות את הקבוע, מקבלים שגיאה בזמן הרצת הקוד.

```
const MY_CONSTANT = 10;
MY_CONSTANT = 11;
```

נשאלת השאלה, למה להגביל את עצמנו ולהשתמש בקבועים? קבועים עילים יותר מבחינה דיברנו מנסיבות שלא אכנו אליהן כאן. ואם זה נשמע לכם איזוטרי, אפשר לבוא ולומר ששימוש מושכל-ב-`const` מעלה את ערככם בעיני כל מי שיקרא את הקוד שלכם. לפיכך כדאי וצריך להזכיר אותו. בשלב זהה של הלימוד קשה לדון על מערכות מורכבות, אך במערכות מורכבות זה קרייטי לחת ערכים שאף אחד לא יכול לשנות או להשפיע עליהם. זו הסיבה שאנו לומדים ומשתמשים בקבועים.

תרגיל:

הגדירו קבוע בשם `foo` והכנסו לתוכו מחרוזת טקסט. הדפיסו את הקבוע.

פתרון:

```
const FOO = 'Hello World!';
console.log(FOO);
```

הסבר:

הגדרת הקבוע נעשית באמצעות המילה השמורה `const`. לאחר מכן אפשר להדפיס את הקבוע בדיק כmo משתנה.

תרגיל:

הגדירו קבוע בשם `foo`, הכנסו מספר, שנו את המספר ובחנו את התקלה שהקונסולה מציגה.

פתרון:

```
const FOO = 10;
FOO++; // Uncaught TypeError: Assignment to constant variable.
```

הסבר:

מגדירים קבוע בשם `foo` באמצעות המילה השמורה `const` ומציבים בתוכו את המספר `10`. אחר כך מנסים לשנות את המספר באמצעות אופרטור ההוספה המקוצר `++` שמוסיף למספר `1`. בקונסולה מופיעה מיד השגיאה "Uncaught TypeError: Assignment to constant variable". הטקסט המדוקע עשוי להשתנות בהתאם לסוג הדף או לגרסתו, אבל שגיאה תהיה שם.

פרק 8

בחרת זרימה - פונקציית



בקרת זרימה – פונקציות

פונקציה היא סוג של רכיב בג'אווהסקריפט שמבצע פעולה ויכול להחזיר תוצאה. אפשר להשתמש בפונקציה宬ן לבניית דברים מסובכים והיא החלק החשוב ביותר בשפה.

פונקציה מוגדרת באמצעות המילה השמורה `function`, שם הפונקציה, סוגרים ריקים וסוגרים מסולסים שבתוכם המילה השמורה `return`, שמחזירה ערך שהוא התוצאה של הפונקציה. הערך זהה יכול להיכנס לכל משתנה שהוא.

מבלבל? הנה נסתכל על הדוגמה הבאה:

```
function myFunc() {
  return 1;
}
```

ראשית הוגדרה פונקציה בשם `myFunc`. ההגדירה נעשית באמצעות המילה השמורה `function` ושם הפונקציה. במקרה זה נבחר השם `myFunc`. אחרי השם יש סוגרים ריקים ואז סוגרים מסולסים. בתוך הסוגרים המסולסים מופיעה הפונקציה עצמה, ייחידה ואוטונומית של קוד שלא תרוץ אם לא נקרא לפונקציה. במקרה זה אין מידע-מה קוד, רק המילה השמורה `return -1`. כמובן מי שיקרא לפונקציה יקבל 1 בתגובה.
AIR מרים פונקציה? ככה:

```
function myFunc() {
  return 1;
}
let foo = myFunc();
console.log(foo); // 1
```

כאן מגדירים את הפונקציה וקוראים לה. את תוצאות הפונקציה מכניסים למשנה. מה לפה? דעיכם יקבל המשתנה? את מה שהפונקציה מחזירה. במקרה זה 1.

הבה ננסה ליצור עוד פונקציה:

```
function anotherFunction() {
  return 'Hello World!';
}
let foo = anotherFunction();
console.log(foo); // "Hello World"
```

גם כאן מגדירים פונקציה וקוראים לה בשם. איך ההגדירה מתבצעת? באמצעות המילה השמורה `function` ואז שם הפונקציה וסוגרים עגולים. הסוגרים המסוללים כוללים את הפונקציה. כרגע אין בה הרבה – היא מחזירה באמצעות המילה השמורה `return` לכל מי שקורא לה את מחוזת הטקסט `"Hello World"`. אם קוראים לה ומוכנסים את התוצאות שלה למשתנה `foo`, הוא יקבל את מה שהפונקציה מחזירה, במקרה זה `"Hello World"`.

אפשר לחלק את עניין ההגדירה וההפעלה של פונקציה לשולש חלקים. החלק הראשון הוא הגדרת הפונקציה באמצעות המילה השמורה `function`. החלק השני הוא ההחזרה – מה הפונקציה מחזירה. החלק השלישי הוא הקריאה, והוא נעשה מחוץ לפונקציה. הקריאה תמיד תחזיר את מה שמחזר על ידי ה-`return`.

```
function anotherFunction() { 1
  return 'Hello World'; 2
}
let foo = anotherFunction(); 3
console.log(foo); // "Hello World"
```

ובן שהפונקציה יכולה לעשות עוד דברים ולא רק להחזיר מחוזת טקסט או מספר. הבה נסתכל על הפונקציה הבאה:

```
function calculateMe() {
  const myVar1 = 10;
  const myVar2 = 20;
  const result = myVar1 + myVar2;
  return result;
}
let foo = calculateMe();
console.log(foo); // 30
```

מגדירים את הפונקציה באמצעות המילה השמורה `function` כרגע ומקפידים לשים סוגרים עגולים. בתוך הסוגרים המסוללים מתרחש האksen: מגדירים משתנים, עושים חיבורים ומקבלים תוצאה אותה מוחזרים עם `return`.

כל הדבר הנדר זהה שהוא הפונקציה לא יעבוד אם לא תקראו לו. קוראים לו באמצעות קראיה בשם הפונקציה והסוגרים העגולים והכנסת מה שהוא מוחזירה בתוך המשתנה `foo`. במקרה

זהו הוא יקבל את מה שהפונקציה מחזירה, שזה 30. החגיגה שמתרכשת בתחום הפונקציה – הכולמר הגדרת קבועים ופעולות מתמטיות – מתרכשת אך ורק בתחום. היא לא זולגת החוצה. זה מה שיפה בפונקציה – אפשר לעשות מה שרצים בתחום זהה לא יזלוג החוצה. בדיקן כמו לאו וגאס – מה שקרה בפונקציה נשאר בפונקציה, והדבר היחיד שזולג ממנה הוא ה-`return`. את היכולת המופלאה זו אדגים באמצעות הדוגמה הבאה:

```
function calculateMe() {
  const foo = 20;
  const bar = 30;
  const result = foo + bar;
  return result;
}
let foo = calculateMe();
console.log(foo); // 50
```

מה יש כאן? פונקציה בשם `calculateMe` שבתוכה מגדרים שני קבועים: `foo` ו-`bar`, וכן את `result` שלתוכו מכניסים את הסכימה של `foo` ו-`bar` ומוחזירים את `result`.

כאמור, מה שקרה בפונקציה נשאר בפונקציה, והפונקציה לא תופעל אם לא תקרו לה ולא תכניסו את מה שהוא מוחזירה (כולומר מה שיש מיד אחרי ה-`return`) למשתנה כלשהו. אך חזיז ורעם! שם המשתנה הוא `foo`! איך זה יכול להיות? הרוי לנו בפרק על קבועים שאפשר להגדיר קבוע רק פעם אחת! קוד זה:

```
const foo = 50;
let foo = 50;
console.log(foo); // Uncaught SyntaxError: Identifier 'foo' has already
been declared
```

יקפוץ שגיאה ולא יroz. מנוע הג'אווהסקריפט לא יכול להריץ אותו כי אחד מכללי השפה הבסיסיים הוא שא-אפשר להגדיר משתנה וקובע בעלי אותו שם. אך איך זה קרה פה? התשובה היא כאמור שם שנמצא בתחום הסוגרים המסתוללים, הכולמר בתחום הפונקציה, מתקיים במד אחר ולא זולג החוצה. הממד الآخر זה נקרא "לקסיקל סקופ" (Lexical Scope), ולפונקציה יש סקופ משלה, הכולמר ממד שבו אפשר להגדיר משתנים שהיו מבודדים לחלוtin מהסקופ של מי שקורא לפונקציה. נושא הסקופ יחזור בהמשך כנושא שבסוג בפונקציות מתקדים יותר.

```

function calculateMe() {
  const foo = 20;
  const bar = 30;           → scope 1
  const result = foo + bar;
  return result;
}
let foo = calculateMe();
console.log(foo); // 50 → scope 2

```

אפשר כמובן לקרוא לפונקציה כמה פעמים שונים. גם הקוד הזה, למשל, יעבוד:

```

function calculateMe() {
  const foo = 20;
  const bar = 30;
  const result = foo + bar;
  return result;
}
let foo = calculateMe();
let bar = calculateMe();
let answer = calculateMe();
console.log(foo); // 50
console.log(bar); // 50
console.log(answer); // 50

```

מה שקרה פה הוא שמחוץ לפונקציה השתמשתם בדיק באותם שמות שבhem קראתם קבועים בתוך הפונקציה, והכל עובד. למה? כי מה שקרה בתוך הסקופ של הפונקציה לא רלוונטי ולא זולג לסקופים אחרים. הסקופ שבו עובדים כרגע נקרא "הסקופ הגלובלי" ואם קוראים לפונקציה מתחוץ, דבר לא יזלוג החוצה אליו. בפונקציה אפשר לעשות כל מה שרוצים ולקרא למשתנים כרצונכם. אמשיך לכתוב על סקופים בהמשך.

שים לב: בדוגמה קראתי לפונקציה מספר רב של פעמים, ובכל פעם הפונקציה רצתה כאילו זו הפעם הראשונה. הפונקציה היא ייחידה אוטונומית ועצמאית ויכולת להיקרא מספר רב של פעמים. בכל פעם היא תחזיר למי שקורא לה את מה שיש מיד אחרי ה-`return`.

ברגע שיש `return`, הפונקציה מוחזירה את הערך שנכתב. אבל מה קורה אם לא נכתב דבר? מההו בסוגנון זהה:

```

function myFunc() {
  return;
}
let foo = myFunc();
console.log(foo); // undefined

```

מה שיקרה הוא שהפונקציה תחזיר `undefined`. זוכרים אותו? למדנו עליו בפרק על סוגים מיידע פרימיטיביים נוספים. זהו מבנה נתונים בסיסי והוא חוזר כאשר כותבים סתם `return` או כאשר אין `return` כלל. למשל פה:

```
function myFunc() {  
}  
let foo = myFunc();  
console.log(foo); // undefined
```

אם יש כמה `return`, ה-`return` הראשון הוא שקובע והקוד שמתבצע לאחריו לא ירצו לעולם. בדוגמה זו למשל:

```
function myFunc() {  
    return;  
    const myVar1 = 'result';  
    return myVar1;  
}  
let foo = myFunc();  
console.log(foo); // undefined
```

הקטע שמייד אחרי ה-`return` בתוך הפונקציה לא ירצו לעולם. שימו לב: אפשר להגיד כמה פונקציות שרוצים ולקראא להן איך שרצו. בכל הדוגמאות הוגדרה פונקציה אחת, אבל אין שום בעיה להגיד כמה פונקציות שרוצים.

פונקציה עם ארגומנטים

אפשר להעביר לפונקציה ערכים מבוחץ ולקבל אותם בתוך הפונקציה. זה נעשה בשני אופנים. ראשית בהגדרת הפונקציה. זוכרים את הסוגרים העגולים הריקים? עכשו הם לא יהיו ריקים – אני אכנס לתוכם משתנה. המשטנה זהה מוגדר בתוך הפונקציה כאילו הגדרתי אותו עם `let`. אפשר להכפיל אותו או ללקחת אותו ולהציב אותו איפה שרוצים, בדיק כמה משתנה רגיל. השם המקביל למשטנה זהה הוא ארגומנט.

air קובעים אותו? אפשר להכניס אותו בקריאה עצמה. הנה, הביטו בדוגמה:

```
function multiply(arg1) {  
    const answer = arg1 * 2;  
    return answer;  
}  
let foo = multiply(10);  
console.log(foo); // 20
```

הפונקציה `multiply` מקבלת ארגומנט בשם `arg1`. כאמור, ברגע שמכרים עליו בסוגרים אפשר להשתמש בו בפונקציה עצמה. במקרה זהה לוקחים אותו, מכפילים אותו ומכניסים את התוצאה למשטנה `answer`.

איך מכנים אותו בקריאה? פשוט מאוד, מעתירים את הארגומנט בסוגרים. מה שמעבירים נחשב ל-`arg1`.

אפשר לקרוא כמה פעמים אותה פונקציה ובכל פעם להשתמש בארגומנט אחר, והערך שהפונקציה תחזיר ישתנה:

```
function multiply(arg1) {
  const answer = arg1 * 2;
  return answer;
}
let foo = multiply(10); // 20
let bar = multiply(20); // 40
let baz = multiply(30); // 60
```

הפונקציה היא אותה פונקציה, אבל התוצאה שלה משתנה בהתאם לארגומנט שמעבירים. כShockoraim לפונקציה כר:

`multiply(10);`
از `arg1` בתוך הפונקציה מקבל את הערך 10 והתשובה המוחזרת היא 20, `arg1` כפול 2 שנכנס ל-`answer` ומוחזר עם `return`.
כShockoraim לפונקציה כר:

`multiply(20);`

از `arg1` בתוך הפונקציה מקבל את הערך 20 והתשובה המוחזרת היא 40.
כלומר, הפונקציה נותרת כפי שהיא, רק הקריאה שלה משתנה.

שימוש לב: בדרך כלל משתמשים בפונקציות כאשר כתבים ג'אווהסקריפט כי זו הדרך הטובה ביותר לפרק את הקוד לחלקים קטנים ולהימנע מחרזרות על קוד. חזרות על קוד הן דבר שмотיב להימנע ממנו בקוד כיוון שאם רוצים לשנות אותו, נדרשת עבודה רבה.
אפשר להשתמש בכמה ארגומנטים שרוצים. כך למשל נראית פונקציה עם שלושה ארגומנטים:

```
function multiply(arg1, arg2, arg3) {
  const answer = arg1 * arg2 * arg3;
  return answer;
}
let foo = multiply(1, 2, 3); // 1*2*3 = 6
let bar = multiply(4, 5, 6); // 4*5*6 = 120
let baz = multiply(10, 20, 0); // 10*20*0 = 0
```

מכנים את כל הארגומנטים שרוצים ומפרידים ביניהם באמצעות פסיק בהגדרת הפונקציה. כך למשל מגדירים פונקציה בעלת שלושה ארגומנטים:

`function multiply(arg1, arg2, arg3);`

אפשר להשתמש בארגומנטים האלו כמשתנים מן המניין בתוך הסוגרים המסולסלים של הפונקציה. מה שחשוב הוא שהפונקציה תעשה `return`. כשתקראו לפונקציה, תכניותו שלושה ארגומנטים בקריאה:

```
multiply(1, 2, 3);
```

כל ארגומנט יופרד בפסיק.

ארגוניים עם ערכים דיפולטיביים

נשאלת השאלה, מה יקרה אם יש פונקציה שמקבלת ארגומנטים אבל לא מעבירים לה ארגומנט? למשל:

```
function multiply(arg1) {
  const answer = arg1 * 2;
  return answer;
}
let foo = multiply();
console.log(foo);
```

יש פונקציית **multiply** שמקבלת ארגומנט (הוגדר בפונקציה `c-1`arg1), אבל בקריאה של הפונקציה:

```
let foo = multiply();
```

לא הועבר שום ארגומנט. איזה ערך יקבל arg1? התשובה היא `undefined`, ואז הקוד שלעיל יקבל שגיאה, כי אי-אפשר להכפיל `undefined` ב-2. התוצאה תהיה `NaN` (כאמור, ראשית התיבות של `Not a Number` – לא מספר).

אפשר להכניס ערך בירית מוחלט (דיפולטיבי) לארגוניים שיוכנס אליהם אם מי שקרה לפונקציה לא העביר לה ארגומנטים. עושים זאת באופן פשוט למדי – כר:

```
function multiply(arg1 = 1) {
  const answer = arg1 * 2;
  return answer;
}
let foo = multiply();
console.log(foo);
```

אם בקריאה מעבירים ערך, כל ערך, 1 arg1 יקבל אותו. אבל אם לא מעבירים ערך, arg1 יקבל במקרה הזה את הערך 1 וזה התשובה תהיה 2. יש לזכור שהשמה של ערך נעשית על ידי הסמן `=`.

אם לפונקציה יש כמה ארגומנטים, בג'אווהסקריפט אפשר להכניס לכלם ערך בירית מוחלט.

הסתכלו על הדוגמה זו:

```
function multiply(arg1 = 0, arg2 = 0, arg3 = 0) {  
  const answer = arg1 * arg2 * arg3  
  return answer;  
}  
let foo = multiply(1); // 1*0*0 = 0  
let bar = multiply(4, 5); // 4*5*0 = 0  
let baz = multiply(undefined, 5, 6); // 0*5*6 = 0
```

כאן לכל הארגומנטים יש ברירת מחדל של המספר 0. אם לא מעבירים ארגומנט כלשהו, הוא מקבל 0 (ואז המכפלת של שלושת הארגומנטים תהיה 0).

שימוש לב לדוגמה الأخيرة: הארגומנט הראשון שמעבירים הוא `undefined`, שהוא מילה שומרה שהוזכרה בפרק על סוגי מידע. אני בעצם מצהיר שהargonment הראשון `arg1` הוא `undefined` ומפעיל את ברירת המחדל. שימוש לב שמדובר ב-`undefined` אמיתי ולא בערך ריק כמו מחזנת ריקה או אפילו `null`.

שימוש לב: זה נחשב רע להכניס יותר מחמשה ארגומנטים לפונקציה. אם בפונקציה שלכם יש יותר מחמשה ארגומנטים סימן שציר לפצל אותה לשתי פונקציות או יותר.

הפונקציה כאובייקט

פונקציה ניתנת להגדירה גם כך:

```
let multiply = function (arg1 = 0) {
  const answer = arg1 * 2;
  return answer;
}

let foo = multiply(1);
console.log(foo)
```

זה בדיק אותך כמו להגיד פונקציה בדרך שלנו. שימו לב שהגדירת הפונקציה היא כמו הגדרת סוג מידע פרימיטיבי מסווג מחרוזת טקסט, מספר או `Symbol`. אם תעשו `typeof` על משתנה שהוגדר כפונקציה, תגלו שהסוג של המשתנה הוא `function`:

```
let multiply = function (arg1 = 0) {
  const answer = arg1 * 2;
  return answer;
}
let foo = typeof multiply;
console.log(foo); // "function"
```

מדוע? כי פונקציה בג'אווהסקריפט היא מבנה נתונים חדש. בניגוד לערבים הפרימיטיבים שהכרתם עד כה, פונקציה היא ערך לא פרימיטיבי ומורכב יותר. צריך לזכור שפונקציה בסופו של דבר היא מבנה נתונים שחייבים להכניס למשתנה לבדוק כמו מספר, טקסט, ערך בוליאני וחבריהם. מסיבות ההיסטוריות, אפשר להגיד פונקציה בשתי הדריכים, או בשם:

```
function foo() {}
```

או במשתנה:

```
let foo = function () {}
```

אבל התוצאה היא אותה תוצאה, המשתנה `foo` שיש בתוכו פונקציה. זה הכל. כמעט הבדל אחד קטן ומשמעותי שנקרא Hoisting – העמסה.

Hoisting

כאשר מגדירים פונקציות, המונע של ג'אווהסקריפט מזיז את כלן למעלה. לשם הדגמה, מה לפ' דעתכם תהיה תוצאה הקוד הזה?

```
console.log(foo()); // 5
function foo() { return 5; }
```

משתמשים בפונקציה לפני שהיא מוגדרת, והקוד הזה עובד. למה בדיק הוא עובד? כי המנווע של ג'אווהסקריפט, לפני שהוא מרייך את הקוד – מזיז את כל הפונקציות למעלה. כלומר, הקוד מזרץ כר בפועל, אפילו שלא נכתב כר:

```
function foo() { return 5; }
console.log(foo()); // 5
```

אבל פונקציות שהוגדרו כמשתנים לא עוברות Hoisting. אז מהهو זהה:

```
console.log(foo()); // ERROR! foo wasn't loaded yet
let foo = function () { return 5; }
```

יקבל שגיאה, כי כאמור קוראים לפונקציה לפני שמגדירים אותה. יש לזרה משמעותית כאשר עובדים בקוד מתקדם יותר ולא בהתחלה, כמוובן.

closure

הזכרתי קודם סקופ והראיתי איך לפונקציה יש "מרחב בטוח" משל עצמה. המשתנים שמוגדרים בפונקציה נוטרים בתוכה בשלוחה ולא כל הפרעה. אבל זה לא תמיד מדויק. בעוד המשתנים שוגדרים בפונקציה נוטרים בתוכה, המשתנים המוגדרים בחוץ, בסקופ הגלובלי יותר, כן חשופים לפונקציה, ואני אדגים:

```
function myFunction() {
  console.log(foo);
}
let foo = 'Hello';
myFunction(); // Hello
```

מה קורה פה? הפונקציה `myFunction` היא פשוטה למדי ומדפיסה משתנה בשם `foo`. אבל שימושו לב ש-`foo` מוגדר מחוץ לפונקציה! ועודין, כשהמניסים את הקוד הזה ומריצים אותו, רואים שהפונקציה יודעת את ערך המשתנה הזה!
הינה דוגמה נוספת:

```
function myFunction() {
  const answer = foo + bar;
  return answer;
}
let foo = 1;
let bar = 2;
let baz = myFunction();
console.log(baz); // 3
```

פה הפונקציה תדע מה הערך של `foo` ושל `bar` אף על פי שהם לא מוגדרים בה אלא מחוץ לה. כמובן, `foo` ו-`bar` לא מוגדרים בסקופ של הפונקציה אלא בסקופ הגלובלי ועודין שלא מועברים לפונקציה כארגומנטים, ועודין הפונקציה מכירה אותם.

הדיagramה זו אמורה להבהיר את העניין:

```

function myFunction() {
  const answer = foo + bar;
  return answer;
}

let foo = 1;
let bar = 2;
let baz = myFunction();
console.log(baz); // 3

```

התוכונה זו נקראת בג'אוوهסקרייפט **closure**. המשמעות שלה היא שימושים שהוגדרו בסקופ האב (אם קיימ) של הפונקציה יהיו מוכרים בסקופ של הפונקציה (סקופ הבן). זו תוכונה חשובה מאוד בג'אוوهסקרייפט והיא מאפשרת גמישות, אך היא גם מסוכנת מאוד.

ובן שאם דורסים את המשתנים בסקופ של הפונקציה, המשתנים בסקופ הגלובלי לא נדרסים, אבל מהרגע שהגדרתם משתנה עם אותו שם, ה-closure במשתנה זה לא יתבצע יותר. **למשל:**

```

function myFunction() {
  let foo = 2;
  const answer = foo + bar;
  return answer;
}
let foo = 1000;
let bar = 2;
let baz = myFunction();
console.log(baz); // 4

```

כאן אפשר לראות שבתוך הסקופ של הפונקציה דורסים את foo. foo שמוגדר מחוץ לפונקציה כמובן לא נפגע, אבל מה שמרתחש בתוך הפונקציה הוא ש-foo הופך לעצמאי לחלוטין. מובן שה-closure הוא דו-כיוני ואפשר להשפיע על המשתנים בסקופ האב דרך סקופ הבן.

למשל:

```
function myFunction() {  
    foo = 'Hello!';  
}  
let foo = 'Bye!';  
myFunction();  
console.log(foo); // Hello
```

שימוש לב שבתוך הפונקציה לא הוגדר משתנה foo באמצעות let. מהרגע שעושים את זה משנים לחלוטין את המשתנה שנמצא בסΚופ האב.

כלומר, כל משתנה שוגדר מבוצע חשוף לפונקציה הפנימית. זהCLI חזק שעלול ליצור כאו משמעותי. בגרסאות קודמות של ג'אווהסקרייפט (ES5 ומטה) הcano היה גדול אף יותר כיון שהגדרת המשתנה, שנעשתה באמצעות var, אפשרה ל-var להיכנס תמיד לסקופ הגלובלי. ה-let מוגבל יותר.

שימוש לב: כרגע זה נראה תיאורטי למדי ואףלו מטופש, אבל יש חשיבות עליונה להבנת ה-closure. בלולאות ובפונקציות רקורסיביות, ובטח ובטח באפליקציות מורכבות יותר, יש הישענות רבה על התוכנה הזאת. לפיקך כדי להשיקע זמן בלימוד שלא עוד אחזור לנושא בהמשך.

פונקציה אונומית ופונקציית חץ

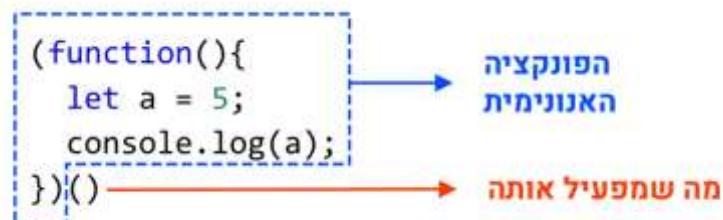
פונקציה אונומית היא פונקציה ללא שם. אחד השימושים המוכרים יותר הוא פונקציה שמריצה את עצמה. איך זה נראה בדרך כלל? ככה:

```
(function () {
  let a = 5;
  console.log(a);
})
```

שים לב שיש כאן הגדרת פונקציה ללא שם, באופן דומה למה שהכרתם: המילה השמורה `function`, סוגרים פותחים וסוגרים והסוגרים המסמלים שבתוכם הפונקציה עצמה. בדוגמה זו הפונקציה מגדרה משתנה `a` ומדפיסה אותו בקונסולה. אבל הדבר הזה לא ירצו כי אף אחד לא קורא לפונקציה. איך קוראים לה? מוסיפים סוגרים פותחים וסוגרים מיד בסוף.

```
(function () {
  let a = 5;
  console.log(a);
})()
```

אם לא תתעכלו ותריצו את קטע הקוד שלעיל, תראו שהוא אכן רץ. למה? כי הפעלתם את הפונקציה האונומית מיד לאחר היצירה שלה.



פונקציות אונומיות, ככלmor כאלו ללא שם, משמשות בהמון מקרים ומקומות בג'אווהסקריפט, בין שמריצים אותן מיד לאחר הפעלה ובין שלא.

בג'אווהסקריפט מודרנית, אפשר לכתוב פונקציה אונומית ללא שימוש במילה השמורה: **Lambda** function

```
(() => {
  let a = 5;
  console.log(a);
})()
```

זה בדוק אותו דבר, אבל מבחינת הסקוופ יש לפונקציות ח' כמה יתרונות גדולים וצדאי להשתמש בהן. נשאלת השאלה, למה להשתמש בכלל בפונקציה אונומית? ויש לה כמה תשובות.

פונקציה אונומית כ משתנה

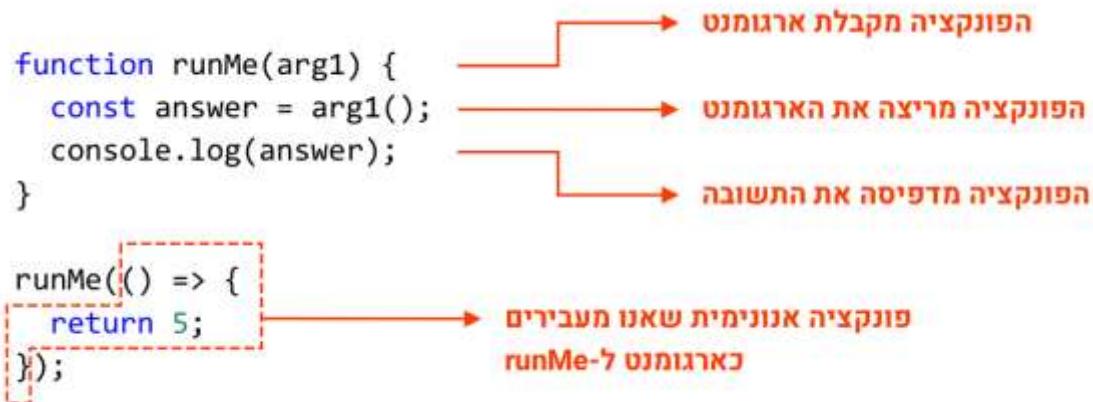
יש פונקציות שמקבלות משתנים שאמורים להיות פונקציות.
מה זאת אומרת? למשל הפונקציה הזו:

```
function runMe(arg1) {
  const answer = arg1();
  console.log(answer);
}
function returnSomething() {
  return 100;
}
runMe(returnSomething);
```

מה שמתתרחש פה הוא שיש שתי פונקציות. פונקציית `returnSomething` ממחזרה 100. פונקציית `runMe` מקבלת משתנה. מה היא עושה בו? לוקחת אותו ו"מrixeh" אותו באמצעות סוגרים פותחים וסגורים. ככל מריה היא מניחה שהארגומנט הוא פונקציה. במקרה זהה הוגדרה פונקציה בשם `returnSomething` והועברה כמשתנה לפונקציית `runMe`. אבל אפשר להעיבר לפונקציית `runMe` גם פונקציית ח' אונומית:

```
function runMe(arg1) {
  const answer = arg1();
  console.log(answer);
}
runMe(() => {
  return 5;
});
```

אם תרצו את זה תראו הדפסה של מה שהפונקציה האונומית ממחזרה.



זה נראה מעט אבסטרקטי ואף מיותר, אבל משתמשים בזה בהמון מקומות בג'אויסקייפט. המוקם הראשון שבו רואים את השימוש בfonקציות אונומיות הוא בlolאות, אבל לא רק שם. יש לא מעט מקומות שבהם מעבירים fonקציה כארוגומנט. הfonקציה שעוברת כארוגומנט נקראת "קולבק" ובאנגלית **callback**. השם הזה ניתן לה כי היא "נקראת בחזרה" על ידי הfonקציה ברגע המתאים. זכרו את השם "קולבק", עוד אשוב אליו.

אפשר להעביר גם ארגומנטים לkolbek. שימוש לב לדוגמה זו:

```

function runMe(arg1) {
  const answer = arg1(2);
  console.log(answer);
}
runMe((myVar) => {
  return 5 * myVar;
});

```

היא זהה כמעט לchlוטין לדוגמה הקודמת, אבל במקרה זהה מצפים שbfonקציה האונומית שמעבירים יהיה ארגומנט אחד. במקרה הזה מועברים 2 לארוגומנט. אם העניין אבסטרקטי. מדי זה בסדר, רק זכרו שkolbekים יכולים לקבל ארגומנטים שהfonקציה הקוראת להם מעבירה. במקרה הזהfonקציית **runMe**ן, שאליה מעבירים ארגומנט שהואfonקציה (או kolbek), מעבירה 2 ארגומנט לkolbek.

fonקציה אונומית שmbודדת מהסקופ הגלובלי

fonקציה אונומית שמריצה את עצמה היא נהדרת בכל מה שקשרו לבידוד מהסקופ הגלובלי. ולממן אפשרות למתכנת להגדיר שכבה שרק דרךה אפשר להגיע אל רכיב הקוד שהוא כתוב. זה מעת מתיקד מדי נכון לעכשו, אבל אנסה להסביר בכל מקרה. שימוש לב לfonקציה זו:

```

let jQuery = {};
((jQuery) => {
  let foo = 'Hello';
}

```

```
jQuery.bar = 'World!'
})(jQuery);
console.log(jQuery.bar); // World!
console.log(foo); // foo is not defined because it is private and in
the function scope
```

הfonקציה האנונימית שמריצה את עצמה מקבלת ארגומנט בשם jQuery. כל מה שמרתחש בתוך הfonקציה הזה נשאר חסוי מסקופים אחרים. אם מגדירים foo (לא משנה אם בעזרת `let` או `var`) אז הוא יישאר חסוי. אפשר לבחור לחושף את `bar` אם מצמידים אותו לאובייקט `jQuery`. מן הסתם, אם רוצים להשתמש באובייקט `jQuery` על מנת לתקשר עם הסביבה החיצונית, הסביבה החיצונית תצטרך ליצור אותו.

כאמור, אם זה נשמע תלוש מעט זה בסדר גמור. בשלב הזה של הלימוד טוב לזכור שfonקציה אונימית שמריצה את עצמה לגיטימית לשימוש בספריות ג'אווהסקרייפט שונות (כמו ספריית `jQuery` ובכל מודול שהוא, שבו רוצים למשה סוף נפרד ופרט). לא נדר לראות ברשות דוגמאות של פונקציות אונימיות שמריצות את עצמן.

תרגיל:

צרו פונקציה בשם `myFunc` שמחזירה `null` למשתנה. הדפiso את המשתנה בקונסולה.

פתרון:

```
function myFunc() {
    return null;
}
let foo = myFunc();
console.log(foo); // null
```

הסבר:

יצרים פונקציה באמצעות המילה השמורה `function` וסוגרים עגולים. בסוגרים המסוללים רואים מה שקרה בתוך הפונקציה. בפונקציה לא קורה כלום והוא מחזירה רק `null`. מפעילים את הפונקציה באמצעות `()` ומכניסים את מה שהיא מחזירה, במקרה זה `null`, אל המשתנה `foo`, שהוא מדפיסים בקונסולה בשורה הבאה.

תרגיל:

צרו פונקציה בשם `ahlaBahla` שמחזירה את המספר 100 למשתנה. הדפiso את המשתנה בקונסולה.

פתרון:

```
function ahlaBahla() {
    return 100;
}
let foo = ahlaBahla();
console.log(foo); // 100
```

הסבר:

יצרים פונקציה באמצעות המילה השמורה `function` וקוראים לה `ahlaBahla`. מיד אחרי שמה שמים סוגרים עגולים `()`. בתוך הסוגרים המסוללים, שחייבים לשים, מתקיימת הפונקציה. במקרה זה היא לא עושה הרבה אלא רק מחזירה 100. איך היא מחזירה משהו? באמצעות המילה השמורה `return`, שמחזירה את מה שכותב אחרת, במקרה זה 100.

הfonקציה לא תתקיים ולא תרוץ אם לא תקראו לה. את זה עושים באמצעות:

```
let foo = ahlaBahla();
```

כאן יש קראיה לפונקציה והכנסה של מה שהוא מוחזירה ל-`foo`. הדפסה של `foo` תראה את זה.

תרגום:

צרו פונקציה בשם `yay` שמחזירה את מחרוזת הטקסט "yay". הכניסו את מה שהוא מוחזירה למשתנה `foo` והדפיסו את המשתנה בקונסולה.

פתרון:

```
function yay() {
    return 'yay';
}
let foo = yay();
console.log(foo); // yay
```

הסבר:

יצרים פונקציה בשם `yay` עם המילה השמורה `function`. מיד אחרי ההגדלה יש סוגרים מסולסים, שבהם מתרחש כל מה שקורה בפונקציה. במקרה זה, דבר לא מתרחש. הfonקציה מוחזירה את מחרוזת הטקסט "yay" באמצעות המילה השמורה `return`. מה שקורא לפונקציה – כי ללא הקראיה היא לא תופעל – הוא הקוד הבא:

```
let foo = yay();
```

הקוד הזה עושה שני דברים – קורא לפונקציה ומעביר את מה שהוא מוחזירה למשתנה `foo`. המשתנה `foo` יודפס ויכיל את מה שיש בו, כמובן מה שהfonקציה החזירה.

תרגיל:

כתבו פונקציה שמקבלת ארגומנט, מדפיסה אותו בקונסולה ואז מוסיף לו 1 ומחזירה אותו. קראו לפונקציה עם ארגומנט 1 והכניסו את התוצאה שלה למשתנה. הדפiso אותו בקונסולה.

פתרונות:

```
function addMe(arg1) {
  console.log(arg1);
  const answer = arg1 + 1;
  return answer;
}
let foo = addMe(1);
console.log(foo);
```

הסביר:

כתיבה הפונקציה שמקבלת ארגומנט היא פשוטה. ראשית עושים את זה:

```
function addMe(arg1) {  
}
```

כלומר, מגדירים פונקציה ובתוכה הסוגרים העגולים מכניסים ארגומנט. Unless צרי להוסיף את ההדפסה בקונסולה:

```
function addMe(arg1) {  
  console.log(arg1);  
}
```

הารוגמנט הוא משתנה לכל דבר והוא ח' בתוך הפונקציה ברגע שקוראים לה. אפשר להדפיס אותו בקונסולה, כמובן, כמו כל משתנה אחר. Unless צרי להוסיף לו 1 ולהחזיר את התוצאה. את זה כבר הכרתם:

```
function addMe(arg1) {  
  console.log(arg1);  
  const answer = arg1 + 1;  
  return answer;  
}
```

אחרי שכתבם את הפונקציה, צריך לקרוא לה. את הקראה מבצעים כך:

```
let foo = addMe(1);
console.log(foo);
```

מעבירים כารוגמנט את המספר 1 לפונקציה ומכניסים את מה שהוא מחזירה לתוך המשתנה foo, שאותו מדפיסים.

תרגום:

צרו פונקציה בשם whoAmI שמקבלת ארגומנט. אם הארגומנט זהה הוא מספר חיובי היא תדפיס בקונסולה +. אם הארגומנט הוא מספר שלילי היא תדפיס בקונסולה -. אם הארגומנט הוא לא מספר חיובי ולא מספר שלילי היא תדפיס בקונסולה ?. בצעו שלוש קראות לפונקציה - עם מספר חיובי, עם מספר שלילי ועם 0 - כדי לראות שהכל עובד.

פתרון:

```
function whoAmI(number) {
  if (number > 0) {
    console.log('+');
  } else if (number < 0) {
    console.log('-');
  } else {
    console.log('?');
  }
}
whoAmI(1); // +
whoAmI(-1); // -
whoAmI(0); // ?
```

הסבר:

כתבם את הפונקציה כמו כל פונקציה אחרת עם ארגומנט שהוא מקבלת. בחרתי את השם number עבור הארגומנט. הארגומנט זהה הוא המשתנה לכל דבר בתוך הסקופ של הפונקציה, ואפשר לכתוב עליו משפטי תנאי כפי שלמדנו בפרקם הקודמיים. שימו לב שלפונקציה זו אין return כיון שהיא ערך אלא מדפסה בקונסולה בלבד. לפיכך גם לא צריך להכניס את מה שהיא מחזירה לתוך המשתנה אלא לבצע קראה בלבד שמשמעותה אותה.

תרגילים:

כתבו פונקציה שמקבלת מספר ובודקת אם הוא מספר או סוג מידע אחר היא תדפיס בקונסולה ה/ח ותסימן את פועלתה. אם הוא מספר היא תחזיר true אם הוא זוגי או false אם הוא אי-זוגי.

רמז – את הבדיקה אם הארגומנט הוא מספר עושים באמצעות האופרטור typeof. את הבדיקה אם הוא זוגי או לא זוגי עושים באמצעות %, אופרטור הבודק חילוק לפ' שארית. למדנו על האופרטורים הללו בפרקם הקודמים.

פתרונות:

```
function whoAmI(number) {
  if (typeof number !== 'number') {
    console.log('n/a');
    return;
  }
  if (number % 2 === 0) {
    return true;
  } else {
    return false;
  }
}
let foo = whoAmI(2);
console.log(foo); // true
let bar = whoAmI(1);
console.log(bar); // false
let baz = whoAmI('a');
console.log(baz); // n/a & undefined
```

הסבר:

את הפונקציה מגדרים כרגיל, כמו בתרגילים הקודמים, כולל ארגומנט. הארגומנט ח' וק'ים בפונקציה כמו כל משתנה. שמו הוא number. ראשית בודקים אם סוג המידע שלו שונה ממספר באמצעות האופרטור typeof שמחזיר את סוג המידע הק'ים במשתנה. אם הוא לא מספר מדפיסים בקונסולה ה/ח ומבצעים ה-return ה-h-return, כפי שלמדנו, יסימן את פועלתה הפונקציה סופית, אבל הוא יופעל, כמובן, רק אם התנאי יתמשך.

בנחחה שההתנאי לא התמשך, ניגשים לבדיקה הזוגי/אי-זוגי ואת זה עושים באמצעות האופרטור שארית, שסימנו %. האופרטור זה, להזכירם, מחזיר את השארית של החלוקה. אם מספר מחלק ב-2 ללא שארית סימן שהוא זוגי. אם יש שארית סימן שהוא אי-זוגי. כתובים את התנאי ומחזירים ערך בוליאני באמצעות המילים השמורות true או false. כל מה שנותר לעשות הוא לבדוק את הפונקציה עם כמה ארגומנטים שונים.

תרגילים:

צרו פונקציה שמקבלת ארגומנט. הארגומנט אמור להיות פונקציה. אם הוא לא פונקציה – מופעלת שגיאה. אם הוא אכן פונקציה מרכיבים אותו ומדפיסים את התוצאה. בדקו פעמיים – עם פונקציה אונומית שמחזירה מחרוזת טקסט ועם פונקציה אונומית שמחזירה מספר.

פתרונות:

```
function callMe(arg1) {
  if (typeof arg1 !== 'function') {
    console.log('Not a function');
    return;
  } else {
    const answer = arg1();
    console.log(answer);
  }
}
// Test
callMe(() => { return 'hello'; }); // hello;
callMe(() => { return 5; }); // 5
```

הסבר:

פונקציית callMe מקבלת את arg. בודקים אותו באמצעות האופרטור typeof. אם הוא לא פונקציה, מדפיסים הודעה שגיאה ומחזירים את הפונקציה כריקה. אם arg הוא פונקציה, מתייחסים אליו כפונקציה וקוראים לה. את מה שהיא ממחזירה מדפיסים.

עד כאן זה פשוט; הבדיקה מעט יותר מסובכת. על מנת לבדוק צריך ליצור פונקציה שמחזירה משהו. את זה עושים באמצעות פונקציית חץ אונונימית. הפונקציה האונונימית הראשונה ממחזירה מחרוזת טקסט והפונקציה האונונימית השנייה ממחזירה מספר.

פרק 6

אוֹבִיְקָטִים



אובייקטים

כבר למדנו על סוגים מיידע פרימיטיביים בג'אווהסקריפט — מחרוזות טקסט, מספרים, `null`, `undefined`, `Symbol` ובוליאני. בפרק הקודם למדנו על פונקציה והסבירתי שמדובר בסוג מיידע שאינו פרימיטיבי (כלומר סוג מיידע מורכב) מסוג `function`. אם יוצרים פונקציה ובודקים את הסוג שלה באופרטור `typeof` מגלים שהוא `function`.

סוג מיידע נוסף שאינו פרימיטיבי הוא **אובייקט**, ומדובר בסוג המידע החשוב ביותר בג'אווהסקריפט. המטרה של אובייקט היא ליצור סוג מיידע מורכב שיכול להכיל סוגים מיידע אחרים, פרימיטיביים או אפילו אובייקטים ומערכות (עליהם מדובר בפרק הבא). איך יוצרים אובייקט? הדרך פשוטה ביותר היא זו:

```
let myObject = {};
```

אם מרכיבים `typeof` על המשתנה שמכיל את האובייקט, מגלים שהוא מסוג `object`.

```
let myObject = {};
let foo = typeof myObject;
console.log(foo); // object
```

אובייקט יכול להכיל בתוכו מידע לפי " מפתחות ". מפתחות הם דרך להכניס מידע נוסף תחת שם מזוהה לאובייקט — כך שהוא יוכל לראות את המידע זהה ולשייר אותו לקטגוריה (שהיא בעצם ה" מפתח ") . כך למשל מכניסים מידע לאובייקט תחת המפתח `id` :

```
let myObject = {};
myObject.id = 1;
console.log(myObject); // Object {id: 1}
```

יצרים אובייקט ומכניסים אותו למשתנה `myObject` . יצירת המפתח נעשית באמצעות החלטה על שם המפתח (במקרה זהה `id`) והכנסת ערך, בדיק כmo משתנה. שימוש לב נקודה בין `myObject` ל- `id` . זהו אחד המבנים הבסיסיים של השפה.

אפשר להכניס עוד מפתחות, כמוון. למשל מחרוזת טקסט:

```
let myObject = {};
myObject.id = 1;
myObject.name = 'Moshe';
console.log(myObject); // Object {id: 1, name: "Moshe"}
```

גישה למפתח נועשית באמצעות שם המשתנה שמכיל את האובייקט, נקודת ועוד שם המפתח. כך אפשר לקבל את שם הערך או לשנות אותו:

```
let myObject = {};
myObject.id = 1;
myObject.name = 'Moshe';
console.log(myObject.id); // 1
console.log(myObject.name); // Moshe
```

בכל שלב שהוא אפשר להכניס מפתחות לאובייקט וכמוון לשנות אותן. בדיק-בדיק כמו המשתנים! כאן לדוגמה משנים את ה-`id` כמה פעמים:

```
let myObject = {};
myObject.id = 1;
myObject.name = 'Moshe';
console.log(myObject); // Object {id: 1, name: "Moshe"}
myObject.id = 1223;
console.log(myObject); // Object {id: 1223, name: "Moshe"}
myObject.id = 'foo';
console.log(myObject); // Object {id: "foo", name: "Moshe"}
```

אפשר לראות שכשמדפים את האובייקט, המפתחות והערכים מסודרים באופן זהה:
`key1: value1,`
`key2: value2,`

וכן הלאה. אפשר ליצור את האובייקט ישירות, ממש כך:

```
let myObject = {
  id: 1,
  name: 'Moshe',
};
console.log(myObject); // Object {id: 1, name: "Moshe"}
```

זה בדיק כמו ליצור את המפתחות בדרך הקודמת. הכל מאד-מאוד גמיש וצריך לזכור את זה. הדרך הישירה לייצור האובייקט – קלומר יצירה שלו כבר עם כל המפתחות מראש – מתקבלת יותר וגם עדיפה מבחינה ביצועים.

שימוש לב: יש פסיק בסיום כל הגדרת משתנה. במקרים מסוימים יותר, פסיק בשורה האחרונה באובייקט (במקרה שלנו הפסיק מייד אחרי `Moshe`) נחשב לשגוי, אך משנת 2016 אפשר להשתמש בפסיק גם בשורה האחרונה בהגדרת האובייקט.

אובייקט בג'אווהסקריפט הוא גמיש. מובן שאפשר להכניס לתוכו ערכים עם משתנים, למשל באופן הבא:

```
let id = 1;
let name = 'Moshe';
let myObject = {
  id: id,
  name: name,
};
console.log(myObject); // Object {id: 1, name: "Moshe"}
```

לא להתבלבל! יוצרים כאן שני שמות משתנים שבמקורה זהים לחילוטין לשמות המפתחות. נכוון, `name: name` עלול לבלבל – אבל הראשון הוא המפתח והשני הוא הערך או, נכון יותר, שם הערך שהמפתח נכנס אליו. חשוב לציין שמדובר הצבה של המשתנה במפתח המתאים באובייקט, הקשר בין המשתנה לערך נעלם. המשתנה יכול להשתנות, אבל הערך יישאר כפי שהיא ברגע הצבה. זה קורה כי למעשה נוצר אובייקט בזיכרון שמכיל את הערך ולא הפניה למשתנה שנמצא במקום אחר בזיכרון. הינה דוגמה:

```
let id = 1;
let name = 'Moshe';
let myObject = {
  id: id,
  name: name,
};
console.log(myObject); // Object {id: 1, name: "Moshe"}
name = 'yakkov';
console.log(myObject); // Object {id: 1, name: "Moshe"}
```

זה אכן מבלבל, אבל זה קורה המון בפונקציות – מתכנתים אוהבים מאוד להשוו את שמות הארגומנטים לשמות המפתחות. משהו בסגנון זהה:

```
function createMyObject(id, name) {
  let myObject = {
    id: id,
    name: name,
  };
  return myObject;
}
let result = createMyObject(1, 'Moshe');
console.log(result); // Object {id: 1, name: "Moshe"}
```

כמפתחי ג'אויסקייפ מנוסים, אתם לא אמורים להיבהל מהקוד הזה – יש פונקציה שמקבלת שני ארגומנטים, מכניסה אותם לאובייקט ומחזירה אותו. זה משהו שאתם אמורים להכיר אחרי הפרק הקודם על פונקציות. את הקוד הזה תראו חוזר כמה וכמה פעמים במערכות מבוססות ג'אויסקייפ, עד כדי כך שהכניסו לג'אויסקייפ את היכולת ליצור מפתחות וערכים באופן אוטומטי לפי שמות הערכים. כך הפונקציה שלעיל תהיה שוקולה:

```
function createMyObject(id, name) {
  let myObject = {
    id,
    name,
  };
  return myObject;
}
let result = createMyObject(1, 'Moshe');
console.log(result); // Object {id: 1, name: "Moshe"}
```

יצירת המפתחות האוטומטית זו נקראת "יצירה מקוצרת" והיא הולכת ותופסת תאוצה בשנים האחרונות. לא משתמש בה בדוגמאות, אבל אני זכרו שהיא קיימת כי נעשה בה שימוש רב, בעיקר על ידי מתכנתים מנוסים (כולל עבדכם הנאמן).

כאמור, אובייקט יכול להכיל כל נתון שהוא בתור ערך, כולל... אובייקטים אחרים! שימוש לבדוגמה זו למשל:

```
let user = {
  id: 1,
  name: 'Moshe',
};

let profileExtendedData = {
  profileImg: null,
  address: 'Derech Hashalom',
  language: 'HE-IL',
}

user.moreData = profileExtendedData;
console.log(user); // Object {id: 1, name: "Moshe", moreData: Object}
/*
id:1
name:"Moshe"
moreData:
{
  address:"Derech Hashalom"
  language:"HE-IL"
  profileImg:null
}
*/
```

אם אתם רואים בקונסולה `{...}` ולא את האובייקט המלא, לחצו על השורה זו והאובייקט המלא יוצג לפניכם.

זאת דוגמה כמעט אמיתית – יש אובייקט `user` ואובייקט מידע מורחב. מכנים את האובייקט של המידע המורחב אל מפתח שנקרא `moreData` באובייקט `user`. כשמבצעים הדפסה של `user` בקונסולה אפשר לראות את האובייקט `profileExtendedData` מופיע במלואו תחת המפתח `moreData`. נפלא, לא?

אפשר לגשת אל מפתח גם באמצעות סוגרים מרובעים ולא באמצעות נקודה:

```
let myObject = {};
myObject.id = 1;
myObject.name = 'Moshe';
console.log(myObject.id); // 1
console.log(myObject['id']); // 1
console.log(myObject.name); // Moshe
console.log(myObject['name']); // Moshe
```

משתמשים בזה עיקר כאשר שם המפתח נמצא בתוך משתנה מסוג מחזורת טקסט. כך למשל:

```
let myObject = {};
let propertyName = 'name';
myObject[propertyName] = 'Moshe';
console.log(myObject.name); // Moshe
console.log(myObject[propertyName]); // Moshe
```

זה קורה המון בפונקציות ובולולאות. פעמים רבות מקבלים את המפתח כמחזורת טקסט. על מנת לקרוא למפתח מתוך האובייקט משתמשים בסוגרים מרובעים, ובתוכם שמים את המשתנה שמכיל את שם המפתח.

אם תנסו לעשות משהו זהה (הו המשתנה שמכיל את שם המפתח) תקבלו undefined.

```
console.log(myObject.propertyName); // Undefined
```

לפיכך לעולם לא משתמשים בנקודה אם שם המפתח נמצא בתוך המשתנה, אלא בסוגרים מרובעים.

מחיקת מפתח

מחיקת המפתח נעשית באמצעות האופרטור **delete**:

```
let myObject = {};
myObject.id = 1;
myObject.name = 'Moshe';
console.log(myObject); // Object {id: 1, name: "Moshe"}
delete myObject.name;
console.log(myObject); // Object {id: 1}
```

אפשר להשתמש, כמובן, בנקודות או בסוגרים מרובעים כדי להסביר לאופרטור **delete** איזה מפתח הוא צריך למחוק.

הכנסת פונקציה כערך

כפי שציינתי קודם, כל סוג מידע יכול להכנס לאובייקט כערך. הדוגמתי זאת באמצעות סוג מידע פרימיטיבים, כמו מספר וטקסט, וגם הראיתית שאפשר להכנס אובייקט כערך לאובייקט אחר. מובן שגם פונקציה יכולה להכנס כערך ואפשר ליצור פונקציות כערך באופן הבא:

```
let myObject = {
  id: 1,
  alertMe: function () {
    console.log('hi!');
  }
};
console.log(myObject); // Object {id: 1, alertMe: f () }
myObject.alertMe(); // hi!
```

כאן יוצרים אובייקט כפי שהוא קודם. משתמשים במבנה **alertMe** ומכוונים לתוכו פונקציה שכל מה שהיא עשוה הוא להדפיס בקונסולה את מחרוזת הטקסט "hi". יצירת הפונקציה זהה במאה אחז ליצירת פונקציה רגילה והכנסתה לתוך משתנה. הקראיה לפונקציה בתוך אובייקט גם כן זהה לחולוטין לקראיה לפונקציה וונעשית כפי שקוראים לכל ערך אחר.

לפונקציה בתוך אובייקט יש חשיבות גדולה מאוד ואdon בה בהמשך. כרגע, מה שחשוב הוא שתדעו שכל מבנה מידע יכול להכנס לאובייקט, בין שמדובר בסוגי מידע פרימיטיבים ובין שבסוגי מידע מורכבים יותר.

אפשר להכנס פונקציה גם כתיב מקוצר (פונקציית חצ, שעליה למדנו בפרקים הקודמים) באופן הבא:

```
let myObject = {
  id: 1,
  alertMe: () => {
    console.log('hi!');
  }
};
console.log(myObject); // Object {id: 1, name: "Moshe"}
myObject.alertMe(); // hi!
```

קינון ובדיקה מפתח

כפי שכתבנו אובייקטים יכולים לכלול אובייקטים אחרים. כך למשל אני יכול ליצור מפתח בשם address באובייקט שמכיל אובייקט נוסף שבו יש את שם העיר ושם הרחוב. למשל:

```
let myObject = {
  id: 1,
  name: 'Moshe',
  address: {
    city: 'Bat-Yam',
    street: 'Balfur 20'
  }
};

console.log(myObject.id); // 1
console.log(myObject.name); // Moshe
console.log(myObject.address); // {city: "Bat-Yam", street: "Balfur 20"}
```

אם אני רוצה לגשת ישירות לשדה מסוים באובייקט שנמצא תחת המפתח, כל מה שנדרש ממי לעשוט זה להוסיף נקודה ואז את שם המפתח באובייקט הפנימי. למשל:

```
console.log(myObject.address.city);
```

אבל אם אנו קוראים למפתח שאינו קיים באובייקט, אנו נקבל שגיאה. כך למשל, אם יש לי אובייקט שבו אין address אני אקבל שגיאה.

```
let myObject = {
  id: 1,
  name: 'Moshe'
};

console.log(myObject.id); // 1
console.log(myObject.name); // Moshe
console.log(myObject.address); // undefined
console.log(myObject.address.city); // Uncaught TypeError: Cannot read
property 'city' of undefined
```

השגיאה מתקבלת כיון ש-`myObject.address` הוא `undefined` ואני מנוטים להתייחס אליו כאובייקט. הפתרון הוא להשתמש בשרשור אופציוני (Optional Chaining). שזו נקודה עם סימן שאלה לפניה. השרשור האופציוני לא יציג שגיאה אלא `undefined`.

```
let myObject = {
  id: 1,
  name: 'Moshe'
};

console.log(myObject.id); // 1
console.log(myObject.name); // Moshe
console.log(myObject.address); // undefined
console.log(myObject.address?.city); // undefined
```

לא נדר לראות במערכות מורכבות, שבהן יש אובייקטים מוקוונים רבים, את השימוש בשרשור אופציוני שנכנס לשפה רק בשנת 2020 באופן רשמי.

אובייקט קבוע

אחד הדברים החשובים שצריך לזכור הוא שם מגדרים אובייקט קבוע, אין שום בעיה לשנות אותו. נכון, אי-אפשר לשנות את הסוג שלו (זאת אומرت, קבוע שמשמעותו במקומות אובייקט יכול קבוע מסטר), אבל אפשר להוסיף לו תכונות או לשנות תכונות קיימות, ככלומר לשנות את השדות בתוך האובייקט ולא לעשות השמה לאובייקט אחר. מדובר בהבדלמשמעותי:

```
const myObject = {};
myObject.name = 'Moshe';
myObject.id = 22;
console.log(myObject); // Object {name: "Moshe", id: 22}
myObject = 1; // Uncaught TypeError: Assignment to constant variable.
```

בדוגמה שלעיל אפשר לראות איך מגדירים את `Object` כקבוע, אך למרות זאת אפשר לשנות את הערכים בתוכו. אבל אם מנסים לשנות את הסוג מאובייקט למשהו אחר (במקרה הזה מספר), נתקלים בשגיאה.

יש כמה הבדלים משמעותיים בין סוג מידע פרימיטיבים לבין אובייקטים וסוג מידע מורכבים יותר. אחד השינויים ה/cgi מבלבלים הוא עניין הצבעה, שאינו כל כך מרכיב כלפי שוחשבים. כשכתבתי על משתנים פרימיטיבים, ציינתי שאפשר להעתיק את המשתנה למשתנה אחר. אם מנסים את המשתנה האחר, המשתנה המקורי נשאר כהיה. הינה דוגמה:

```
let a = 5;
let b = a;
b = 6;
console.log(a); // 5
```

אבל מה קורה כשעשים משהו דומה באובייקט? אם מעתיקים אותו למשתנה אחר? אם עושים את זה ומשנים את המשתנה الآخر, רואים שהאובייקט העיקרי השתנה! נסו להריץ את הקוד הזה למשל:

```
let objectA = { value: 5 };
let objectB = objectA;
objectB.value = 6;
console.log(objectA); // {value: 6}
```

שינוי באובייקט, שאמור להיות במשתנה השני, משפייע על האובייקט המקורי! זה קורה כי שימושים אובייקט שנמצא במשתנה `A` למשתנה `objectB` לא מבצעים העתקה אלא הפניה, וההבדל ביןיהםמשמעותי. הסיבה לכך היא שיקולי זיכרון. אם רוצים לבצע העתקה של אובייקט ולשנות אותו בלי שהוא ישפייע על האובייקט המקורי, צריך לעשות הליך שנקרא `clone` ואפשר לבייצוע באמצעות לויאות, ועל כך בפרק הבא.

תרגיל:

צרו אובייקט של `computer` שיש לו `id`, `name` ו-`price`.

פתרון:

```
let computer = {
  id: 1,
  name: 'Name',
  price: 20,
};
console.log(computer); // Object {id: 1, name: "Name", price: 20}
```

הסבר:

יצרים משתנה ומכוונים אליו אובייקט, באובייקט יש שלושה מפתחות: `id`, `name` ו-`price`. כל אחד מהם קיבל ערך. ההגדרה של האובייקט נעשית באמצעות סוגרים מסולסלים, שבתוכם שם המפתח, נקודותים ואז ערך. אפשר ליצור את האובייקט גם באופן הבא:

```
let computer = {};
computer.id = 1;
computer.name = 'Name';
computer.price = 20;
console.log(computer); // Object {id: 1, name: "Name", price: 20}
```

בפועל יוצרים כאן אובייקט ריק ואז מכוונים את המפתחות שלו בזה אחר זה. הקראיה למפתח נעשית כמו משתנה – שם האובייקט, נקודה ואז שם המפתח.

תרגיל:

צרו פונקציה שמקבלת מספר מ-1 עד 7. הפונקציה מחזירה אובייקט בפורמט זהה:

```
{
  dayName: 'Sunday',
  dayNumber: 1,
}
```

פתרון:

```
function findDayName(dayNumber) {
  let dayName;
  switch (dayNumber) {
    case 1:
      dayName = 'Sunday';
      break;
    case 2:
      dayName = 'Monday';
      break;
    case 3:
      dayName = 'Tuesday';
      break;
    case 4:
      dayName = 'Wednesday';
      break;
    case 5:
      dayName = 'Thursday';
      break;
    case 6:
      dayName = 'Friday';
      break;
    case 7:
      dayName = 'Saturday';
      break;
    default:
      console.log('Not 1-7 number');
      return {};
  }
  let answer = {
    dayNumber: dayNumber,
    dayName: dayName,
  }
  return answer;
}
let foo = findDayName(1);
console.log(foo); // Object {dayNumber: 1, dayName: "Sunday"}
```

הסבר:

הfonקציה ארוכה, אך לא צריך להיבהל. היא מקבלת ארגומנט שקוראים לו `dayNumber`, והמטרה היא למצוא את מספר היום. לצורך כך משתמשים במשפט `switch case` שלMANDO.

אם המספר הוא לא בין 1 ל-7 מדפיסים הודעה שגיאה ומחזירים אובייקט ריק. אם המספר הוא בין 1 ל-7 מוצאים את היום ומכניסים אותו למשתנה `dayName`. את `dayNumber` ואת `dayName` מכניסים לאובייקט ומחזירים אותו. זה הכל.

כיוון שהשמות המפתחות זהים לשמות הערכים, אפשר ליצור את האובייקט באופן קצר:

```
let answer = {
  dayNumber,
  dayName,
}
```

תרגיל:

צרו פונקציה שמקבלת שלושה ארגומנטים: אובייקט, שם מפתח וערך. הפונקציה מכניסה את שם המפתח והערך לאובייקט ולאחר מכן מחזירה את האובייקט.

פתרון:

```
function addThisProperty(obj, property, value) {
  obj[property] = value;
  return obj;
}
let myObject = {};
myObject = addThisProperty(myObject, 'id', 1);
myObject = addThisProperty(myObject, 'name', 'moshe');
console.log(myObject); // Object {id: 1, name: "Moshe"}
```

הסבר:

יצרים פונקציה עם שלושה ארגומנטים. הראשון הוא האובייקט, השני הוא שם המפתח והשלישי הוא הערך שצריך להיכנס למפתח. כיוון שם המפתח מגיע כמשתנה, אי-אפשר לעשות משהו כזה:

```
object.property = number;
```

אלא צריך להשתמש בסוגרים מרובעים להגדרה. כל שנותר לעשות אחרי הוספת התוכנה והערך הוא להחזיר את האובייקט ולבדוק. זה הכל.

תרגיל:

צרו אובייקט שיש בו שתי פונקציות. הראשונה מדפיסה בקונסולה את מה שמועבר אליו והשנייהמחזירה אובייקט שיש בו {id: 1, name: 'Moshe'}

פתרונות:

```
let object = {
  say: function (arg1) {
    console.log(arg1);
  },
  returnObject: function () {
    const answer = {
      id: 1,
      name: 'Moshe',
    }
    return answer;
  },
};
object.say('Hello world!!'); // Hello world!!
let foo = object.returnObject(); // Object {id: 1, name: "Moshe"}
console.log(foo);
```

הסבר:

יצירת האובייקט היא פשוטה. יוצרים שני מפתחות: מפתח בשם say, שהוא פונקציה המתקבלת ארגומנט אחד ומדפיסה אותו, ומפתח returnObject שהוא פונקציה שמחזירה אובייקט. זה הכל. אחר כך כל מה שנותר לעשות הוא לקרוא לפונקציות באמצעות המפתחות.

תרגיל:

צרו פונקציה בשם displayClass שמקבלת אובייקט בשם user שנראה כך:

```
let userObject = {
  id: 4,
  name: Moshe,
  enrichedData: {
    class: 4,
    icon: 3454
  }
}
```

הfonקציה מחזירה את ה-class שבתוך enrichedData אם הוא קיים. אם הוא לא, מוחזר undefined.

פתרונות:

```
function displayClass(userObject) {
  return userObject.enrichedData?.class;
}

let userObject1 = {
  id: 4,
  name: 'Moshe',
  enrichedData: {
    class: 4,
    icon: 3454
  }
}

let userObject2 = {
  id: 4,
  name: 'Moshe',
}

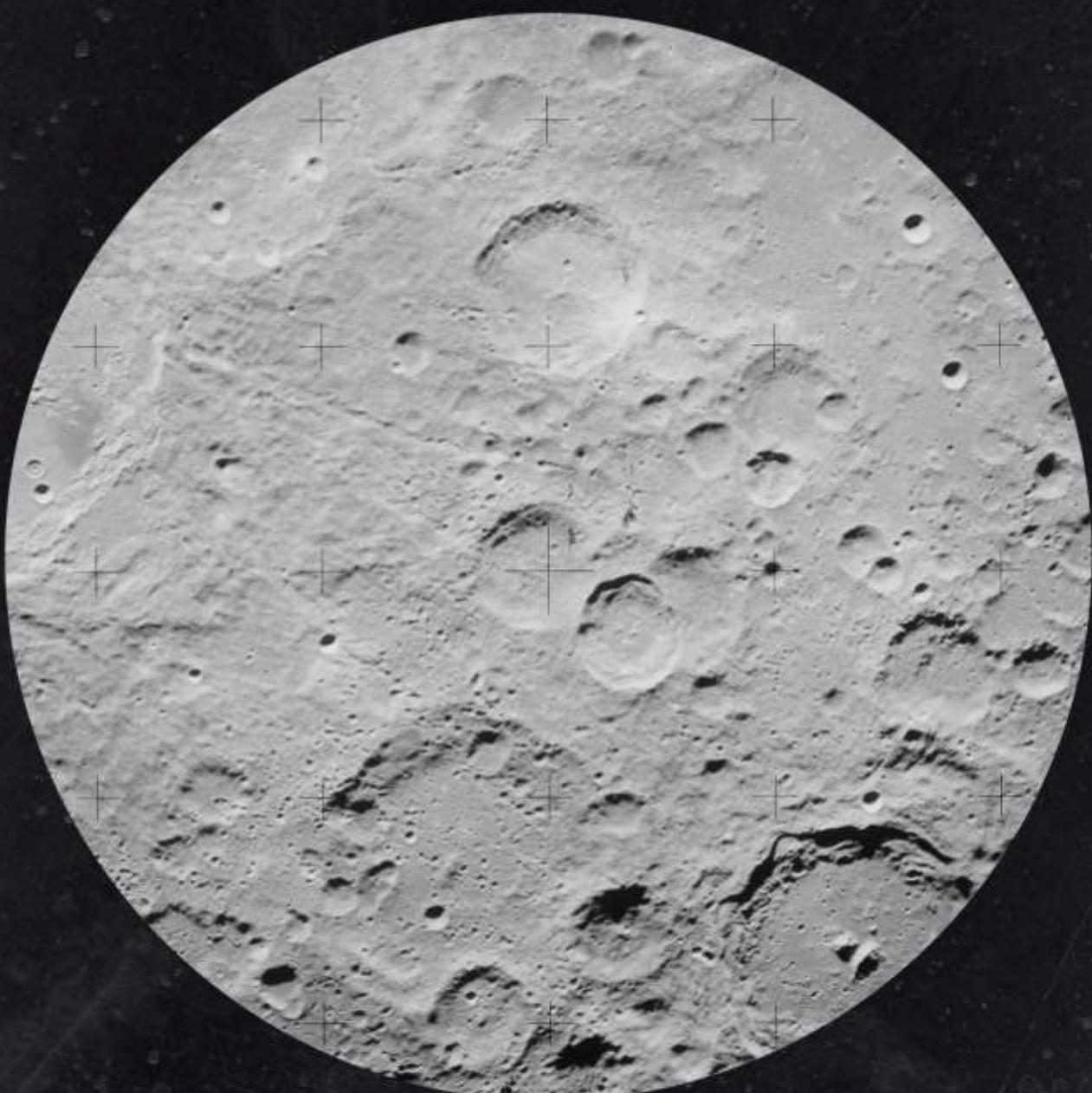
let result = displayClass(userObject1);
console.log(result); // 4
result = displayClass(userObject2);
console.log(result); // undefined
```

הסבר:

אני משתמש בשרשור אופציונילי כדי להחזיר אובייקט מוקדם. אם אני משתמש בשרשור רגיל (userObject.enrichedData.class) אני מקבל שגיאה באובייקט השני שבו אין את תכונת.enrichedData.class. לפיכך אני אכניס את (userObject.enrichedData?.class).

פרק 10

מִנְרָכִים



מערכות

מערכות הם סוג מידע מורכב המשמש לאחסון מידע. הם דומים למדוי לאובייקטים, אך המפתחות שלהם הם מספרים מ-0 עד אינסוף. אל המערך אפשר להכנס כל סוג מידע שהוא. מקובל להתייחס אל הערכים שנמצאים במערך כאל איברים; האיבר הראשון נמצא במקום 0, האיבר השני נמצא במקום 1 וכך הלאה.

זה דבר שתמיד מבלב מתכנתים. כשאדבר על לולאות תבין את ההיגיון של להתחיל מ-0, אבל חשוב לשנן ולזכור שבכל הנוגע למערכות תמיד סופרים מ-0.

יצירת המערך נועשית כך:

```
let myArr = [];
```

כאן יוצרתי מערך ריק. על מנת להכנס לתוך המערך איבר, אפשר לעשות את הדבר הבא:

```
myArr[0] = 'someValue' ;
```

הדפסה של המערך תראה את האיבר שהכנסתי:

```
console.log(myArr); // [ "someValue" ]
```

אם רוצים להכנס עוד איבר, אפשר להוסיף אותו כך:

```
myArr[1] = 'someValue' ;
```

אפשר להכנס איברים כבר בשלב היצירה של המערך ולגשת אליהם בכל שלב:

```
let myArr = [ 'value1', 'value2', 'value3' ];
console.log(myArr[0]); // value1
myArr[0] = 'new value';
console.log(myArr[0]); // new value
```

כאן למשל יוצרים מערך וכבר בשלב היצירה מכניסים לתוכו שלושה איברים לפי הסדר - למקום 0, למקום 1 ולמקום 2. אחרי היצירה מדפיסים בקונסולה את האיבר שבמקום הראשון (מקום 0), משנים אותו ומדפיסים אותו שוב.

שימוש לב: פניה לקבלת ערך מຕוך מערך נועשית על ידי סוגרים מרובעים וגם האינדקס של האיבר.

אפשר כמובן להכנס גם אובייקטים לתוך מערכים זהה אפילו מקובל:

```
let usernameObject = [ { id: 1, userName: 'Avraham' }, { id: 2,
userName: 'Itzhak' }, { id: 3, userName: 'Yaakov' }, ];
console.log(usernameObject[0]); // {id: 1, userName: 'Avraham'}
console.log(usernameObject[0].id); // 1
```

כאן יוצרים מערך שכל איבר שלו הוא אובייקט שבו יש שני מפתחות – `id` ו-`userName`, ואלה שולפים את האובייקט הראשון מהמערך ואףלו לא את כל האובייקט הראשון, אלא מפתח נבחר מהאובייקט הראשון.

וכן, בדיק כפוי שמערך יכול להכיל כל סוג מידע שהוא, כולל אובייקטים, מערך יכול להכיל גם מערכים. למערך שמכיל מערכים נוספים קוראים מערך דו-ממד. הינה דוגמה לדוגמה צזה:

```
let myArray = [
  ['a', 'b', 'c'], // 1st array
  ['d', 'e', 'f'], // 2nd array
];
```

מערך בעצם ממש מבנה נתונים שידוע בתכנות כ"מחסנית". כמו שיש מחסנית ובה כדרים, קר יש מחסנית של נתונים. הכנסה של איבר חדש נקראת "דחיפה", ואפשר למשתמש איתה באמצעות שימוש בפונקציה מיוחדת לסוג המידע מערך בלבד, שנקראת "push":

```
let myArray = ['oldValue'];
myArray.push('newValue');
console.log(myArray); // ["oldValue", "newValue"]
```

push היא פונקציה מיוחדת שעובדת אך ורק על סוג מידע שהם מערכים. ככל-mdנו על סוג מידע, הראית כל מיני פעולות שאפשר לעשות על מחרוזת טקסט או על מספרים; ובכן, push היא פעולה שאפשר לעשות אך ורק על מערך. מכניםים לתוכה ארגומנט את המידע שרצים להכנס למערך בתור האיבר החדש והאחרון – בין שמדובר במספר ובין שבמחרוזת טקסט, באובייקט, במערך וכו'. כל דבר יכנס כArgument ב-push, ויוצר איבר חדש בסוף.

הדחיפה מוסיף איבר חדש למערך על גבי האיברים האחרים. אם היה איבר אחד במקומות 0, הדחיפה תוסיף איבר חדש למקומות 1.

משיכה היא בדיק ההפר מדחיפה – היא מאפשרת למשוך את האיבר האחרון מהמערך ולקבל אותו כמשתנה. המשיכה נעשית באמצעות פקודה **pop**, שגמ היא ייחודית לסוג המידע מערך ועובדת כך:

```
let myArray = ['a', 'b', 'c'];
let foo = myArray.pop();
console.log(myArray); // ["a", "b"]
console.log(foo); // c
```

כשמשתמשים ב-**pop** על המערך, מקבלים בחזרה את האיבר האחרון של המערך, והמערך עבר שינוי (מוותיצה), או נכון יותר המחסנית, מתקצר באיבר אחד.

בעוד push מכניס איבר לסוף המערך, הפונקציה **unshift** מכניסה איבר לתחילת המערך. כמו שתי הפונקציות הקודמות, גם **unshift** ייחודית למערך ולא תעבוד בסוגי מידע אחרים. הינה דוגמה לאיך ש-**unshift** פועל:

```
let myArray = ['a', 'b', 'c'];
myArray.unshift('z');
console.log(myArray); // ["z", "a", "b", "c"]
```

בדוגמה יש מערך שבו שלושה איברים, הראשון הוא a והאחרון הוא c. אם מכניםים את מחרוזת הטקסט z לתוכו המערך באמצעות unshift, אז האיבר הראשון כבר לא יהיה a אלא z. האיבר האחרון עדין נותר c.

ולבסוף, שיליפת האיבר הראשון במערך נעשית באמצעות פונקציית shift:

```
let myArray = ['a', 'b', 'c'];
let foo = myArray.shift();
console.log(myArray); // ["b", "c"]
console.log(foo); // a
```

בדוגמה זו רואים מערך שבו האיבר הראשון הוא מחרוזת טקסט a והאיבר האחרון הוא מחרוזת טקסט c. שימוש בפונקציה shift מוחזיר את האיבר הראשון. כרגע במערך האיבר הראשון הוא a והאיבר האחרון הוא עדין c. המערך קצר משלושה איברים לשניים.

אפשר לסכם את ארבע הפעולות שניתן לעשות על מערכים באופן הבא:

תיאור	פעולה
שליפת האיבר האחרון של המערך	pop
הכנסת איבר לתחילת המערך	unshift
שליפת האיבר הראשון של המערך	shift
הכנסת איבר לסופם של המערך	push

תכונה נוספת שקיימת וייחודית לסוג המידע של מערך היא אורך. מדידת האורך של המערך נעשית באופן הבא:

```
let myArray = ['a', 'b', 'c'];
let foo = myArray.length;
console.log(foo); // 3
```

כך אפשר לראות את אורך המערך. שימוש לב שמדובר במקרה טריקי. אם למשל יוצרים מערך ומוכנסים אליו שני ערכים בלבד – אחד במקומות ה-0 (שהוא המיקום הראשון, אנו זוכרים שהמיקום הראשון במערך הוא 0) והשני במקומות ה-99 – האורך של המערך יהיה 100:

```
let myArray = [];
myArray[0] = 'a';
myArray[99] = 'b';
let foo = myArray.length;
console.log(foo); // 100
```

למה? כי ברגע שמכניסים ערך למערך וקובעים את המיקום שלו, ג'אווהסקריפט תיצור את כל האיברים האחרים במערך עד המיקום שהוכנס. כל איבר צזה יהיה מסוג `undefined` (שימו לב – לא `null` אלא `undefined`).

מצד שני, ככה תמיד אפשר לקבל את הערך האחרון במערך, שהוא תמיד ה-`length` פחות 1:

```
let myArray = [];
myArray[0] = 'a';
myArray[99] = 'b';
let last = myArray[myArray.length - 1]
console.log(last); // b
```

למה פחות 1? כי המערך מתחילה מ-0. האיבר הראשון תמיד יהיה במקומות 0. כמו באובייקט, מחיקת איבר במערך תתבצע באמצעות האופרטור `delete` שבעצם לוקח את האיבר ומוכנס אליו `undefined`:

```
let myArray = ['aba', 'ima', 'bamba', 'savta'];
delete myArray[2];
console.log(myArray); // ["aba", "ima", undefined, "savta"]
```

שימוש לב ש캐שרוצים למחוק את הערך השלישי, צריך לציין [2] כיוון שהמערך מתחילה תמיד מ-0.

הבעיה בדרכּ הּזּוּ הּיא שּׁה מֵעָרָק נִשְׁאָר בּוּ בּגּוּדּל קְבּוּעּ, וְהּאִיבּר שּׁעָוְשִׁים לֹא delete הּוּפּקּן undefined. כדי למחוק אִיבּר בּמֵעָרָק וְלִקְצַר אֹתוֹ יִשְׁלַׁחַת בּ-splice.

כמו המתוודת הקודמות של מדרנו עליון, גם המתוודה splice שייכת ל-`מֵעָרָק` בלבד. היא מקבלת כמה ארגומנטים. הראשון הוא מאייזה מקום ב-`מֵעָרָק` להתחילה והשני הוא כמה איברים למחוק. נניח שיש מערך של אבות ורוצחים להוריד את משה, כי הוא לא אחד האבות המקראיים.

```
let fathers = ['Avraham', 'Itzhak', 'Moshe', 'Yaakov'];
```

אם מוריידים את משה באמצעות `delete`, יהיה undefined ב-`מֵעָרָק` וככיוול יהיה " ארבעה אבות" והשיר "אחד מי יודע" ישתבש לגמרי. אז מה עושים? משתמשים ב-`splice`. מה המקום של האיבר שרוצחים למחוק (משה)? המיקום ה-2; 0 זה אברהם, 1 זה יצחק ו-2 משה. כמה איברים רוצחים למחוק? אחד. איך תיראה המחיקה? כר:

```
let fathers = ['Avraham', 'Itzhak', 'Moshe', 'Yaakov'];
fathers.splice(2, 1);
console.log(fathers); // ["Avraham", "Itzhak", "Yaakov"]
```

מערכות ומחרוזות טקסט

כדי להגדיל את השמחה ואת הבלבול הכללי, בג'אווהסקריפט מחרוזות טקסט חולקות לא מעט תכונות עם מערכים והן מותנהגות בחלוקת מהמקרים כמו מערכים שבהם האיבר הראשון הוא האות הראשונה, האיבר השני הוא האות השנייה וכך הלאה. שימו לב לمثال לדוגמה ה兹ו:

```
let myString = 'Hello World!';
console.log(myString[0]); // H
console.log(myString[myString.length - 1]); // !
```

כאן יש מחרוזת טקסט חביבה בשם "Hello World!". אם רוצים לקבל את האות הראשונה, אפשר להתychס אליה כמערך ולשלוף את האות הראשונה באמצעות [0] ואפשר גם לשלוף את האות الأخيرة בדיקון כפי ששלופים את האיבר האחרון במערך. האופרטורים **delete** וגם **push** ו-**pop** לא יעבדו, אך כל שאר האופרטורים כן. כרגע אין לנו שימוש אופרטיבי, אבל כדאי לזכור את זה.

תרגיל:

צרו מערך שמכיל את המספרים 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

פתרונות:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
console.log(myArray); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

הסבר:

יצירת מערך נעשית באמצעות סוגרים מרובעים ואז כל איברי המערך מופרדים בפסיק. האיברים יכולים להיות כל נתון שהוא.

תרגיל:

למערך הקודם שיצרתם, הוסיפו את המספר 11 בסוף.

פתרון:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
myArray.push(11);
console.log(myArray); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

הסבר:

באמצעות פונקציית `push`, הייחודית למערך בלבד, אפשר להוסיף ערך בטור האיבר האחרון במערך. בטור הסוגרים העגולים שלאחר ה-`push` מכנים את הערך שרצים שייהי במערך.

תרגיל:

למערך הראשון שיצרתם, הוסיפו את הסורה 0 בהתחלה, כך שהאיבר הראשון יהיה 0.

פתרון:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
myArray.unshift(0);
console.log(myArray); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

הסבר:

הדחיפפה של ערך כאיבר הראשון של המערך והזזה של כל האיברים נעשוות באמצעות הפונקציה `unshift`, שכמו `push` מקבלת את הערך שרצים שייהי ראשון במערך, במקרה זה 0.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר הראשון.

פתרון:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
myArray.shift();
console.log(myArray); // [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

הסבר:

הfonקציית `shift`, שיחודה למערך, מסירה את האיבר הראשון ומציצה את כל שאר האיברים.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר החמישי.

פתרון:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
delete myArray[4];
console.log(myArray); // [1, 2, 3, 4, undefined, 6, 7, 8, 9, 10]
```

הסבר:

הօperator `delete` יכול למחוק משתנים וגם איברים במערך. במקרה זהה רציתם למחוק את האיבר החמישי ולפיכך הייתם צריכים לכתוב `delete myArray[4];` למה 4? כיון שהמערך מתחילה ב-0 והאיבר החמישי יהיה `myArray[4];`

this-i new

אפשר ליצור אובייקטים מאפס באמצעות המילה השמורה **new**. בג'אווהסקריפט משתמשים המון בטכניתה זו ליצור אובייקטים שונים דרך פונקציות. מגדירים פונקציה ואז, באמצעות **new**, יוצרים אובייקט:

```
let ObjMaker = function () { };
let myObj = new ObjMaker();
console.log(myObj); // {}
```

ראשית מגדירים פונקציה פשוטה כמו כל פונקציה אחרת. עושים את זה באמצעות הכנסתה למשתנה. את הפונקציה זו מתחלים באמצעות **new**. שימו לב שימוש מתחלים את הפונקציה. ברגע שימושים במילה **new**, הפונקציה מחרירה אובייקט.

הfonקציה שמתוכנת להיקרא באמצעות **new** נקראת **פונקציה בנית**, ובאנגלית **constructor**. בתשעיה מקובל לכתוב שם של פונקציה בנית באות גדולות בהתחלה, למשל **ObjMaker** ולא **objMaker**, כמו פונקציה רגילה.

יצירת אובייקט ריק פשוטה מאוד, אבל הגדולה של הפונקציה זו היא ביצירת אובייקט בעל תכונות. יצירת אובייקט בעל תכונות נעשית באמצעות המילה השמורה **this**. בתוך פונקציה בנית המשמעות של **this** היא האובייקט שייצור כתוצאה מהפונקציה. **this** מצביע לאובייקט – קלומר הוא מפנה אל האובייקט זהה. אם למשל רוצים להוסיף תכונה בשם **foo**, עושים משהו כזה:

```
let ObjMaker = function () {
  this.foo = 'bar';
};
let myObj = new ObjMaker();
console.log(myObj); // { foo: "bar" }
```

מגדירים את אותה פונקציה בナイית שהוגדרה קודם, אבל במקום פונקציה ריקה, יש בתוך הפונקציה הכרזה על תכונת `foo` שיש לה ערך `bar`. האובייקט שנוצר מהפונקציה הבナイית הוא לא רק אובייקט ריק אלא אובייקט שיש לו `foo`. כל מה שמכריזים עליו באמצעות `this` יקבל ביטוי באובייקט. למשל:

```
let ClientObjMaker = function () {
  this.userFirstName = 'Moshe';
  this.userLastName = 'Cohen';
  this.userCity = 'Holon';
  this.userCar = 'Subaru';
};

let mosheObject = new ClientObjMaker();
console.log(mosheObject);
```

כאן יוצרים אובייקט באמצעות פונקציה בナイית. הפונקציה הבナイית הכרזה על שם פרט, על שם משפחה ועל תכונות נוספות של הלוקה. אם תציבו אל האובייקט באמצעות הקונסולה: תראו שהוא מכיל את כל הארגומנטים שהגדרתם באמצעות `this`:

```
Object {
  userCar: "Subaru",
  userCity: "Holon",
  userFirstName: "Moshe",
  userLastName: "Cohen"
}
```

משתמשים בדרך כלל בפונקציות בונות על מנת ליצור אובייקטים קבועים. למשל, אם רוצים ליצור אובייקטי לקוח אחידים, יוצרים פונקציה בונה איחידה לאובייקט לקוח ומכניםים בכל פעם משתנים אחרים:

```
let ClientObjMaker = function (firstName, lastName, city, car) {
  this.userFirstName = firstName;
  this.userLastName = lastName;
  this.userCity = city;
  this.userCar = car;
};

let mosheObject = new ClientObjMaker('Moshe', 'Cohen', 'Holon',
'Subaru');
let aviObject = new ClientObjMaker('Avi', 'Levi', 'Bat Yam', 'Opel');
console.log(mosheObject);
console.log(aviObject);
```

דוגמה זו יוצרים פונקציה בナイית איחידה לאובייקט משתמש. משתמשים במילה השמורה השעה על מנת להחזיר למשתנה בכל הפעלה אובייקט אחר, בהתאם למשנים שמכניםים לפעולה הבונה.

אפשר להוסיף לפעולה הבונה גם הגדרות של פונקציה שהן חלק אינהרנטי מהאובייקט שהפעולה הבונה יוצרת. לפעולות אלו קוראים "מתודות" (וביחיד "מתודה"). מדובר בפונקציה שמוצמדת לאובייקט ואפשר להפעילה بكلות מהאובייקט שהפונקציה הבナイית יוצרת:

```

let ClientObjMaker = function (firstName, lastName, city, car) {
    this.userFirstName = firstName;
    this.userLastName = lastName;
    this.userCity = city;
    this.userCar = car;
    this.getFullName = function () {
        return this.userFirstName + ' ' + this.userLastName;
    }
};

let mosheObject = new ClientObjMaker('Moshe', 'Cohen', 'Holon', 'Subaru');
let fullName = mosheObject.getFullName();
console.log(fullName); // "Moshe Cohen"

```

כאן למשל יוצרים מตודה בפונקציה הבנאית שנקראת `getFullName`. היא מוצמדת ל-`this` כמו התוכנות, אבל היא פונקציה ולא מחרוזת טקסט. במקרה זהה היא מוחזירה את השם המלא של המשתמש המורכב מתוכנות השם הפרט, שם המשפחה ורוחם בינויים.

אפשר להגדיר משתנים פרטיים בתחום הפונקציה הבנאית – פרטיים במובן שא-אפשר לקבל אותם מן החוץ אלא אם כן מגדירים פונקציה שתגדיר אותם. למשל:

```

let ClientObjMaker = function (firstName, lastName, city, car) {
    this.userFirstName = firstName;
    this.userLastName = lastName;
    this.userCity = city;
    this.userCar = car;
    const id = '6382020';
};

let mosheObject = new ClientObjMaker('Moshe', 'Cohen', 'Holon', 'Subaru');
console.log(mosheObject.id); // undefined

```

כאן יש הגדרת משתנה בשם `id` בתחום הפונקציה הבנאית. בנווגוד לחברו, המשתנה זהה לא מוצמד ל-`this` אלא מוגדר ממש בתחום פונקציה. האם אפשר לגשת אליו מרוחוק? לא. רק מה שמצוידים ל-`this` יהיה נגיש החוצה. מה שלא, מוגדר כפרתי.

לכן צריך לכתוב פונקציה נוספת:

```
let clientObjMaker = function (firstName, lastName, city, car) {
  this.userFirstName = firstName;
  this.userLastName = lastName;
  this.userCity = city;
  this.userCar = car;
  const id = '6382020'; // It doesn't have to be const
  this.getId = function () {
    return id;
  }
};
let mosheObject = new clientObjMaker('Moshe', 'Cohen', 'Holon',
'Subaru');
let id = mosheObject.getId();
console.log(id); // 6382020
```

פונקציה מהסוג זהה נקראת **פונקציית get**, כיון שהיא פונקציה שמשמשת לקבללה (באנגלית (get) של משתנים פרטיים שאין גישה אחרת אליהם. כמו שיש get יש גם set, שזו פונקציה שקובעת את המשתנה הפרט):

```
let clientObjMaker = function (firstName, lastName, city, car) {
  this.userFirstName = firstName;
  this.userLastName = lastName;
  this.userCity = city;
  this.userCar = car;
  let id = '6382020';
  this.getId = function () { // Get function
    return id;
  }
  this.setId = function (newId) { // Set function
    id = newId;
  }
};
let mosheObject = new clientObjMaker('Moshe', 'Cohen', 'Holon',
'Subaru');
mosheObject.setId('246810');
let id = mosheObject.getId();
console.log(id); // 246810
```

נשאלת השאלה, למה בדיק ציריך set ו-get אם אפשר פשוט לחושף את ה-id ב-this כמו שאר המשתנים? התשובה נעה במשמעות. לפעמים רוצים לעשות ולידציה. למשל, רוצים לוודא id הוא תמיד מספר.

הדרך הכי טובה לעשות זאת היא לאכוף את זה באמצעות `set`:

```
let clientObjMaker = function (firstName, lastName, city, car) {
  this.userFirstName = firstName;
  this.userLastName = lastName;
  this.userCity = city;
  this.userCar = car;
  let id = '6382020';
  this.getId = function () { // Get function
    return id;
  }
  this.setId = function (newId) { // Set function
    if (typeof newId === 'number') {
      id = newId;
    } else {
      console.log('Error! not a number!!!!');
    }
  }
};

let mosheObject = new clientObjMaker('Moshe', 'Cohen', 'Holon',
'Subaru');
mosheObject.setId('Some String'); // "Error! not a number!!!!"
let id = mosheObject.getId();
console.log(id); // 6382020
```

בפונקציית `set` בודקים באמצעות האופרטור `typeof` את סוג הארגומנט שהועבר. אם הוא מסוג מספר מכנים אותו ל-`id`, ומעכשיו המשטנה `id` מכיל את הערך זהה. אם לא, מחזירים שגיאה ולא משנים את ה-`id` המקורי.

از מה ההבדל בין הגדרה וגילה של אובייקט לפונקציה בנאית? בדרך כלל משתמשים בפונקציה בנאית על מנת להגדיר אובייקטים שיש להם תפקיד – ייחידת תוכנה שיש לה תכונות ומונדות. דבר על `car` בהמשך. בדרך כלל קוד מודרני של ג'אווהסקריפט ארוז באובייקטים שמבוססים על פונקציה בנאית. ומה עם אובייקטים רגילים? בהם משתמשים בדרך כלל לאחסן מידע בלבד.

בהמשך תראו את התועלת שיש בפונקציה בנאית כאשר מפתחים מערכות מורכבות יותר. בינהים, זכרו שברגע שאתם רואים `new`, זה סימן שתקיבלו אובייקט חדש. דבר על פונקציות בנאיות בפרק על אובייקטים מובנים בגל'אווהסקריפט.

תרגיל:

צרו פונקציה בנית לאובייקט מכונית המקבלת שם, צבע ומספר מנוע. לכל מכונית יש מספר זהה ייחודי המורכב מchipor של דגם, צבע ומספר מנוע. למשל, אם שם היצרן של המכונית הוא opel, הצבע הוא white ומספר המנוע הוא 1,200, מספר זההו שלה יהיה opelwhite1200. צרו לאובייקט מכונית פונקציה המחזיר את מספר זההו. הדגם, הצבע ומספר המנוע הם נתונים שחוופים החוצה.

פתרון:

```
let carObjMaker = function (name, color, engine) {
  this.name = name;
  this.color = color;
  this.engine = engine;
  const modelNumber = this.name + this.color + this.engine;
  this.getModelNumber = function () {
    return modelNumber;
  }
};
let opelObject = new carObjMaker('opel', 'white', '1200');
let id = opelObject.getModelNumber();
console.log(id); // opelwhite1200
```

הסבר:

יצרים פונקציה בנית המקבלת שלושה ארגומנטים: שם, צבע ומספר מנוע. שלושת הארגומנטים הללו נחשפים החוצה באמצעות `this`. בתוך הפונקציה הבנית יוצרים `modelNumber`, שמורכב משלשות הארגומנטים הללו. לא חושפים אותו החוצה באמצעות `this`. השלב הבא הוא ליצור פונקציה שתחזיר אותו. הפונקציה נחשפת החוצה באמצעות `this` ומחזירה את המשתנה הפרט.

תרגיל:

במה שיר לתרגיל הקודם, כתבו פונקציה `set` שתאפשר לשנות את ה-`modelNumber`.
כרצונכם.

פתרונות:

```
let carObjMaker = function (name, color, engine) {
  this.name = name;
  this.color = color;
  this.engine = engine;
  let modelNumber = this.name + this.color + this.engine;
  this.getModelNumber = function () {
    return modelNumber;
  }
  this.setModelNumber = function (newModelNumber) {
    modelNumber = newModelNumber;
  }
};
let opelObject = new carObjMaker('opel', 'white', '1200');
opelObject.setModelNumber('test');
let id = opelObject.getModelNumber();
console.log(id); // test
```

הסביר:

התשובה זהה לתשובה של התרגיל הקודם, למעט הפונקציה `set` שמש לה מטרה אחת –
לשנות את משתנה `modelNumber`. היא מקבלת ארגומנט אחד וקובעת אותו כמשתנה
ה-`modelNumber`. זו גם הסיבה ש-`modelNumber` מוגדר כמשתנה.

פרק 11

תבנית טקס



מבנה טקסט

מבנה טקסט מאפשר ליצור מחרוזות טקסט בклות משתנים שונים. למדנו בפרק על מחרוזות טקסט שאפשר לחבר בין מחרוזות טקסט. כולם אמרו אם משתנה שיש בו "Hello" ומשתנה שיש בו "World", יתקבל מש浩ו זהה אם תחברו אותם:

```
let var1 = 'Hello';
let var2 = 'World';
let combined = var1 + var2;
console.log(combined); // "HelloWorld"
```

אם רצים רוח בין שני המשתנים, צריך להוסיף אותו. למשל מש浩ו זהה:

```
let var1 = 'Hello';
let var2 = 'World';
let combined = var1 + ' ' + var2;
console.log(combined); // "Hello World"
```

אם לדוגמה יש מספר ורוצים להציג אליו את התו \$ (כמו במחירים) צריך לעשות מש浩ו זהה:

```
let price = 10;
let currency = '$';
let combined = price + currency;
console.log(combined); // "10$"
```

אפשר גם לא להכניס את סימן הדולר (\$) כמשתנה ולחבר אותו ישירות אל המשתנה הראשון:

```
let price = 10;
let combined = price + '$';
console.log(combined); // "10$"
```

הבעיה היא שזה מסובבל מאד, וכך שחלף הזמן ותוכנות הג'אווהסקרייפט התפתחו והפכו למורכבות יותר. היה אפשר למצוא בתוכנות מבוססות ג'אווהסקרייפט תפלצות מהסוג הזה (למשל):

```
const user = {
  name: 'Ran',
  localtime: 'Morning',
};

let welcomeString = 'Hello, ' + user.name + '. How are you doing? Good
' + user.localtime + '!';
console.log(welcomeString); // "Hello, Ran. How are you doing? Good
Morning!"
```

כל החיבורים האלה מסובבים מאד. אבל מואוד. לפיכך נוצרה דרך להציג "מבנה" בג'אווהסקרייפט או, נכון יותר, דרך ליצור מחרוזות טקסט מורכבת בלי כל אופרטורי החיבור האלה. איך עושים את זה? יש גרש מסולסל (באנגלית `backtick`), שנמצא במקלדות סטנדרטיות משמאלי למספרה 1 ומעל ה-Tab. הוא נראה כך: `

מדובר בגורם שעבוד בדיקן כמו גרש רגיל' או גרשימים כפויים" בכל מה שנוגע לטקסט. כמובן, משה זהה:

```
let myVar = `Hello world`;
```

בבחירה יעבוד, ו-myVar ייחס למחרוזת טקסט לגיטימית לכל דבר. אבל לגורם הממוסלסל יש יכולת שאין לגורשיים הרגילים, והוא להציג תבנית. אם רצים להכניס משתנה מסוים לתוך מחרוזת הטקסט צריך להקיף אותו בדולר (\$) ובסוגרים מסולסלים, והוא יכנס למחרוזת הטקסט בשלהמו. הנה דוגמה:

```
const user = {
  name: 'Ran',
  localtime: 'Morning',
};

let welcomeString = `Hello, ${user.name} How are you doing? Good
${user.localtime}!`;
console.log(welcomeString); // "Hello, Ran. How are you doing? Good
Morning!"
```

כלומר, הדבר זהה:

```
let welcomeString = `Hello, ${user.name} How are you doing? Good
${user.localtime}!`;
```

זהה לחלוטין לדבר זהה:

```
let welcomeString = 'Hello, ' + user.name + '. How are you doing? Good
' + user.localtime + '!';
```

מה נראה טוב יותר ומובן יותר? התשובה ברורה. תבניות מטפלות באופן נאה מאוד בשורות, ואילו מחרוזות טקסט רגילות לא מסוגלות להתמודד עם ריבוי שורות. אם נוציא ירידת שורה בדוגמה שלעיל, הדוגמה תראה כך:

```
let welcomeString = `Hello, ${user.name},
How are you doing? Good ${user.localtime}!`;
let welcomeString = 'Hello, ' + user.name + '\nHow are you doing? Good
' + user.localtime + '!';
```

מה עדיף? אני חושב שההתשובה ברורה. בשנים האחרונות יש מעבר חד-משמעות אל שימוש בתבניות טקסט בג'אווהסקריפט, ואני ממליץ גם לכם להשתמש בהן.

תרגיל:

נתון מערך שיש בו בוקר וערב.

```
const timeOfDay = [ 'Morning', 'Evening' ];
```

הדףו באמצעות התבנית טקסט את המשפט "Good Morning" ללא שימוש באופרטור חיבור.

פתרונות:

```
const timeOfDay = ['Morning', 'Evening'];
let welcomeString = `Good ${timeOfDay[0]}`;
console.log(welcomeString); // "Good Morning"
```

הסביר:

מחרוזת הטקסט שאותה מדים'ים מוקפת בגרש מסולסל ` מכל צד. הגרש מסמל תבנית טקסט. כיוון שרוצים את האיבר הראשון במערך, קלומר את:
`timeOfDay[0]`

מקיפים אותו בסוגרים מסולסלים שבתחילה יש \$:

```
 ${ timeOfDay[0] } 
```

ומשבצים אותו בתוך מחרוזת הטקסט המוקפת בגרש מסולסל מכל צד. הערך של הביטוי ייכנס לתוך מחרוזת הטקסט.

תרגיל:

כתבו פונקציה שמקבלת שני ארגומנטים, מספר וסמל מטבע, ומחזירה מחרוזת טקסט של שני הארגומנטים מחוברים. למשל, אם מעבירים 30 ו-₪ הפונקציה תחזיר 30₪.

פתרונות:

```
function giveMeLocalAmount(amount, currency) {
  let answer = ` ${amount} ${currency}`;
  return answer;
}
let NISAmount = giveMeLocalAmount(30, '₪');
console.log(NISAmount); // "30₪"
```

הסביר:

cotבאים פונקציה רגילה שמקבלת שני ארגומנטים, amount ו-currency. שני הארגומנטים הללו נוכנסים למחרוזת טקסט אחת שמקופת בגרש מסויל ` מכל צד. הגרש הזה מצביע על הימצאותה של תבנית טקסט שכל משתנה שנמצא בתוכה ומוקף בסוגרים מסוילים וב-\$ – ערכו "יכנס למחרוזת הטקסט. מחרוזת הטקסט פה היא פשוטה:

```
` ${amount} ${currency}`;
```

כלומר, מחרוזת הטקסט היא המספר והערך צמודים. מחזירים את מחרוזת הטקסט הזו. על מנת לבדוק אותה בודקים את הפונקציה – קוראים לה עם שני ארגומנטים, 30 וסמל המטבע של השקל (مוקף בגרשיים כיון שהוא מחרוזת טקסט), ובודקים את התוצאה. ההדפסה תראה שצדקתם.

פרק 12

לולאות



לולאות

לולאות מאפשרות לעבור על האיברים של מערך או של אובייקט בקבוצות רבה. יש לא מעט לולאות בג'אווהסקריפט. חלקן מיועדות למערכים וחלקן לאובייקטים, וכל אחת מהן משמשת למטרה אחרת עם יתרונות וחסרונות משלה. לשם הבירה – אני משתמש במונח "לולאה" לכל איטרציה שהיא.

לולאת `for`

לולאות מאפשרות, בעצם, להריץ קוד שוב ושוב, כמה פעמים שרוצים. מספר הפעמים תלוי בתנאי מסוים. כר' למשל אם רצים שקווד יrotch עשר פעמים, יוצרים משתנה ומצביעים שהקווד הזה יrotch כל עוד המשתנה קטן מ-10 ובכל ריצה מעלים את המשתנה ב-1. כשהמשתנה הגיע ל-10 הריצה תיעצר.

air עושים את זה? כר' :

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

קווד הקוד הזה נקרא **"לולאת for"** והוא יrotch עשר פעמים בדיק, מ-0 ועד 9. הנה נתוח אותו: ביטוי ה-`i`-for מרכיב מהamilת השמורה **for** ומסוגרים עגולים שבתוכם ביטוי ה-`i`-for, המרכיב שלושה חלקים:

1	2	3
<u>for (let i = 0; i < 10; i++) {</u>	<u> console.log('Iteration number ' + i);</u>	
<u>}</u>		

החלק הראשון הוא הגדרת המונה. המונה הוא המשתנה שעולה, או יורד, בכל הפעלה של הלולאה. במקרה הזה קבועים את שמו ומאגדים שהוא יהיה שווה 0. החלק השני עד מתי תרוץ הלולאה. קבועים שהוא תרוץ כל עוד קטן מעשר. אפשר להשתמש فيه בכל אופרטור השוואתי. החלק השלישי הוא מה שקורה למונה בכל הפעלה של הלולאה. במקרה הזה משתמש באופרטור שמוסיף 1. כמו כן כל פעם שהלולאה רצתה, יעלה ב-1. בתוך הסוגרים המסוללים נמצא מה שקורה בכל פעם שהלולאה רצתה.

הבה נראה מה קורה בראיצה הראשונה:

ריצה ראשונה

מעלים את i באחד עומד בתנאי $i = 0$

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

i שווה ל-0. נעשית בדיקה אם i קטן מ-10. כיוון שהוא קטן מ-10, מה שיש בתוך הלולאה עובד, הקונסולה מדפסה את i ומעלים את i ב-1. מרגעים ליריצה הבאה:

ריצה שנייה

מעלים את i באחד עומד בתנאי $i = 1$

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

פה כבר i שווה ל-1. נכון, כתוב פה $i = 0$, אבל זה תקף אך ורק ליריצה הראשונה של הלולאה. ביריצה השנייה תזכור שלא צריך לאותל את i אלא רק להעלות אותו. כיוון ש-1 קטן מ-10 מרים את מה שיש בתוך הסוגרים המסתולסים ואחרי כן מעלים את i ב-1. ביריצה הבאה הוא יהיה שווה ל-2, ונמשיך ליריצה הבאה.

ריצה שלישית

מעלים את i באחד עומד בתנאי $i = 2$

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

פה i שווה ל-2, כיוון שהיא עלה ב-1 בפעם הקודמת. 2 קטן מ-10 והתנאי עדיין תקף. הלולאה מרים את מה שיש בתוך הסוגרים ומעלה את i ב-1. ביריצה הבאה הוא יהיה שווה ל-3 וכך הלאה. עד שנגיע ליריצה האחרונה:

ריצה עשירית ואחרונה

מעלים את i באחד עומד בתנאי $i = 9$

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

הሪיצה האחרונה היא הריצה העשירית. נזכיר את מה שקרה בתוך הסוגרים המסלולים ומעלים את ? ב-1. עכשו הוא שווה ל-10. בריצה הבאה התנאי לא מתקיים, 10 לא קטן מ-10 והלולאה נעצרת.

שימוש לב: כל ריצה או סיבוב של הלולאה נקראים **"איטרציה"**. זה המונח הסטנדרטי. מובן שאפשר לעשות לוලאה שיורדת. שימוש לב:

```
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

אם מרים את הלולאה זו מקבלים את ההדפסות הבאות בקונסולה:

```
"Iteration number 5"  
"Iteration number 4"  
"Iteration number 3"  
"Iteration number 2"  
"Iteration number 1"
```

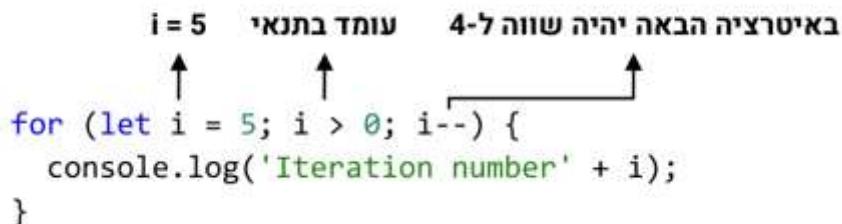
שלושת חלקיו שלולאה הם:

1. אתחול מונה הלולאה והציבתו על 5.
2. קביעת התנאי שהלולאה תרוץ כל עוד ? גדול מ-0.
3. בכל איטרציה ? יורד ב-1.

הבה נראה מה קורה באיטרציה הראשונה:

ריצה ראשונה

באייטרציה הבאה יהיה שווה ל-4 עומד בתנאי $i = 5$



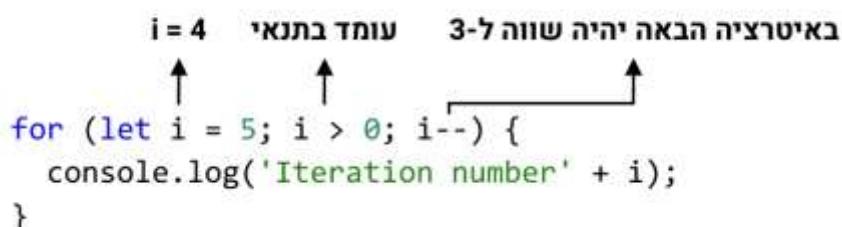
```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

באייטרציה הראשונה מתחילה את i וקובעים אותו על 5. האם הוא עומד בתנאי? כן. 5 גדול מ-0. מדפיסים את מה שיש בתוך הסוגרים המסורלים ומורידים את i ב-1 באמצעות האופרטור $--$.

ובבירם לאייטרציה השנייה:

ריצה שנייה

באייטרציה הבאה יהיה שווה ל-3 עומד בתנאי $i = 4$

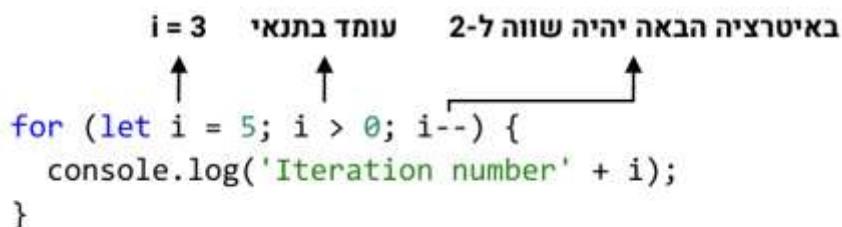


```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

באייטרציה השנייה זו כבר שווה ל-4. הוא עומד בתנאי, 4 גדול מ-0, לפיכך מה שכתוב בתוך הסוגרים המסורלים קורה (יש הדפסה בקונסולה) ו- i מופחת ב-1. הלאה!

ריצה שלישית

באייטרציה הבאה יהיה שווה ל-2 עומד בתנאי $i = 3$

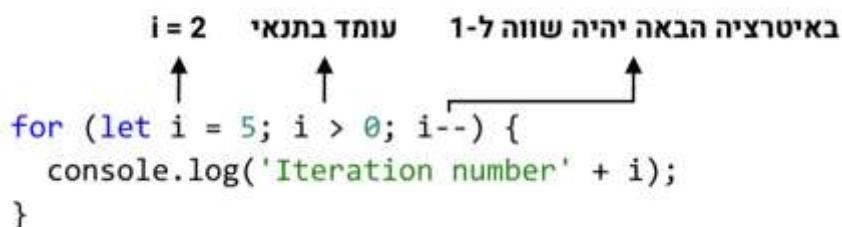


```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

באייטרציה השלישית זו שווה ל-3, עדין יש עמידה בתנאי. מרייצים את מה שקרה בתחום הסוגרים המסורלים ואז מורידים את i ב-1.

ריצה רביעית

באייטרציה הבאה יהיה שווה ל-1 עומד בתנאי $i = 2$



```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

באייטרציה הרביעית נ-2 שווה ל-2. עדין גדול מ-0. מתרחשת הרצה נוספת של מה שיש בסוגרים המסולסים והפחתה של 1 מ-ו.

ריצה חמישית ואחרונה

באייטרציה הבאה יהיה שווה ל-0 עומד בתנאי *i* = 1

↑ ↑ ↑

```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

באייטרציה החמישית והאחרונה נ-1 שווה ל-1. 1 גדול מ-0 ולפיכך מה שיש בסוגרים המסולסים ירוץ. וירד ב-1 ובריצה הבאה יהיה 0.

באייטרציה שלא תקין יש בדיקה אם *i* גדול מ-0. כיוון ש-*i* שווה ל-0, התנאי לא מתקיים. לא גדול מ-0 ולפיכך האיטרציה לא תרוץ. הלולאה נעצרת:

אייטרציה שלא תקיים

לא עומד בתנאי *i* = 0

↑ ↑

```
for (let i = 5; i > 0; i--) {
  console.log('Iteration number' + i);
}
```

שימוש לב: מקובל מאוד לקרוא למשתנה של הלולאה בשם *i*, *j* או *index*.

הינה לדוגמה שמוסיפה ל-4 בכל איטרציה 4 ותפעל כל עוד קטן מ-12 או שווה לו:

```
for (let i = 0; i <= 12; i = i + 4) {
  console.log('Iteration number ' + i);
}
```

הפלט של לולאה צו יהיה:

```
"Iteration number 0"  
"Iteration number 4"  
"Iteration number 8"  
"Iteration number 12"
```

תרגול טוב יהיה לכתוב את הלולאה על נייר ולנסות להבין למה הפלט הוא צהה. הבה נתחיל. באיטרציה הראשונה, מן הסתם הוגדר כך שהיא שווה ל-0. כיוון ש-0 קטן מ-12, ההדפסה בקונסולה מתרחשת ואז מעלים את צהה.

איטרציה מס' 1

```
i = 0           ↗          ↗ i = 4  
for (let i = 0; i <= 12; i+4) {  
  console.log('Iteration number' + i);  
}
```

באיורציה השנייה, א שווה ל-4. מן הסתם 4 קטן מ-12 ומה שיש בתוך הסוגרים הממוסלים יקרה. א. עלה בעוד 4.

איטרציה מספר 2

```
i = 4           i = 8
↑             ↑
for (let i = 0; i <= 12; i+4) {
  console.log('Iteration number' + i);
}
```

באיורציה השלישית, שווה ל-8. עדין קטן מ-12 והפעולה בלולאה תתקיים. עליה ב-4, ל-12.

איטרציה מספר 3

```
i = 8           בריצה הבאה = 12
↑             ↑
for (let i = 0; i <= 12; i+4) {
  console.log('Iteration number' + i);
}
```

זו האיטרציה האחרונה שתתבצע. נ שווה ל-12 ועדין עומד בתנאי. למה? כי התנאי הוא קטן או שווה. 12 שווה ל-12 ויש עמידה בתנאי. הפעולה תרוץ ו-*עליה* ל-16. בריצה הבאה נראה ש-16 גדול מ-12, ולפיכך הפעולה לא תתבצע לעולם. **בום!**

אייטרציה מס' 4 ואחרונה

בפעם הבאה, שלא תרוץ

```

i = 12           i = 16
↑               ↑
for (let i = 0; i <= 12; i+4) {
  console.log('Iteration number' + i);
}

```

דיברתי על **סקופים** בפרק על הפונקציות. גם בלולאות יש סקופ. כל מה שקיים בתוך הסוגרים המסתולסים נחשב לסקופ משלו. זו גם את החוזקות של *let* לעומת *var* הישן יותר, שהוא בעבר *let* שומר על סקופ משלו בלולאה. אם תנסה לגשת אליו מחוץ לולאה תקבלו שגיאה. למשל:

```

for (let i = 0; i <= 12; i = i + 4) {
  console.log('Iteration number ' + i);
}
console.log(i); // Uncaught ReferenceError: i is not defined

```

אבל באמצעות *var* כן אפשר לגשת אליו ממחוץ, וזה מכך את הסקופ הכללי בצורה שלא תיאמן.

```

for (var i = 0; i <= 12; i = i + 4) {
  console.log('Iteration number ' + i);
}
console.log(i); // 16

```

אם אתם רוצים לשמר על הסקופ הכללי שלכם נקי – ואתם רוצים, תאמינו לי, שאתם רוצים – **אל תשתמשו ב-var**. יש בתקן החדש `let`, הבה נשתמש בו. צריך לזכור שהסקופ עובד גם פה, ול-`for` יש סקופ משלה שמתפקידו בכל איטרציה.

לולאת `for` משחיקת יפה מאוד עם מערכים. הבה נדגים. נניח שיש מערך שרוצים להדפיס את כל תוכנו. איך אפשר לעשות את זה? כר:

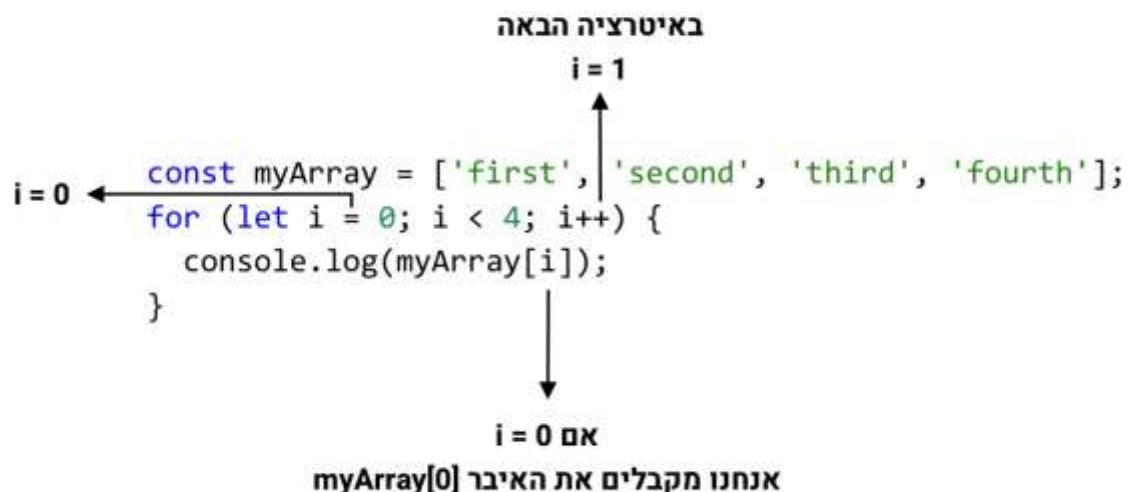
```
const myArray = ['first', 'second', 'third', 'fourth'];
console.log(myArray[0]);
console.log(myArray[1]);
console.log(myArray[2]);
console.log(myArray[3]);
```

אבל זה ארוך ומעצבן, במיוחד אם יש הרבה איברים. במקום זה אפשר לכתוב לולאת `for` פשוטה:

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < 4; i++) {
  console.log(myArray[i]);
}
```

כידע, זו משתנה בכל איטרציה, מ-0 ועד 4 (0, 1, 2, 3). אפשר להציב אותה בתור מספר האיבר במערך ולקבל אותו! הבה נבחן את האיטרציות:

איטרציה מספר 1



באייטרציה הראשונה | מאותחל ושווה ל-0. מן הסטם 0 קטן מ-4, ומה שיש בתוך הסוגרים המסולסים מתקיים. כיוון $sh=0$ אפשר להשתמש בו. זה בדיקן כמו לכתוב:

```
let i = 0;  
console.log(myArray[i]);
```

פשוט במקרה זהה ? כבר קיימ. לא צריך להגיד אותו. מה שיודפו בكونסולה הוא first. עוברים לאייטרציה הבא.

איטרציה מס' 2

באייטרציה הבא

i = 1

i = 2

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < 4; i++) {
  console.log(myArray[i]);
}


i = 1 אמ'



אנו מקבלים את האיבר myArray[1]


```

באייטרציה הבאה א' כבר שווה ל-1. התנאי מתקיים כי $1 < k < 4$, ומה שיש בתחום הסוגרים המסולסים רצ. אפשר להדפיס את האיבר הראשון (שהוא האיבר השני, כי במערכות מתחילה מ-0). עוברים לאייטרציה הבאה, שבה א' עולה ל-2.

איטרציה מס' 3

באייטרציה הבא

i = 2

i = 3

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < 4; i++) {
  console.log(myArray[i]);
}


i = 2 ו

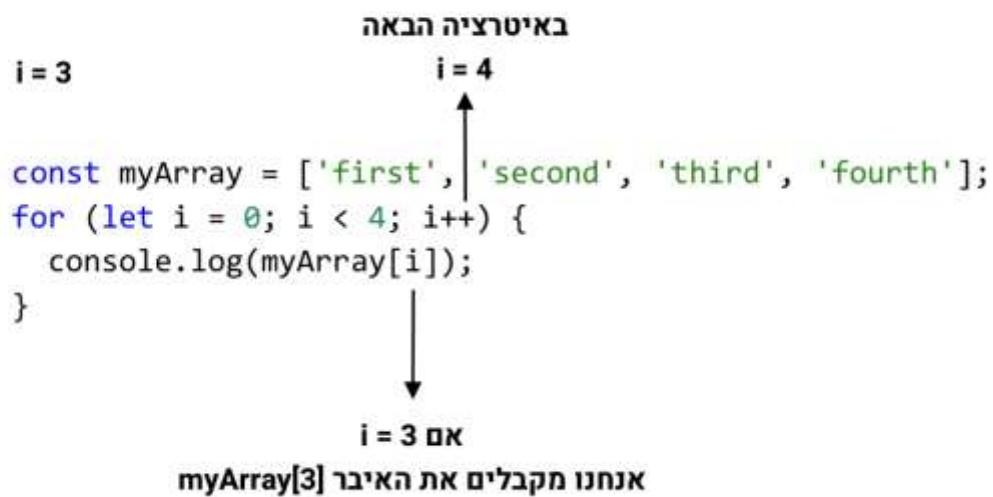


אנו מקבלים את האיבר myArray[2]


```

גם פה, כמו באיטריציות הקודמות, התנאי מתקיים. 2. קטע מ-4. אפשר להשתמש בו-, שכרגוע הוא 2, להדפיס את האיבר השלישי במערך ולבור לאיטריציה הבאה, שבה עולה שוב, הפעם 4-3.

איטרציה מספר 4



באייטרציה الأخيرة זו שווה ל-3 ועדין עומד בתנאי. אפשר לקחת את זה ולהדפיס את האיבר הרביעי בסדרה. באיטרציה הבאה זו שווה ל-4, אבל זה כבר לא מעניין כי 4 לא קטן מ-4 והתנאי לא מתקיים. זו הריצה الأخيرة של הלולאה הזו והמערך הודפס.

הבעיה בגישה זו היא שמניחים שגודל המערך הוא 4. אבל מה אם לא יודעים מה גודל המערך? זוכרים שאפשר לחwlץ את גודל המערך באמצעות שימוש בתוכנה המיוחדת למערך? אפשר להשתמש בה בollowה:

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < myArray.length; i++) {
  console.log(myArray[i]);
}
```

בנוסף, `myArray.length` הוא 4, בדיקן גודל המערך.

לולאה אינסופית

יש כאן לולאה בעייתי. מה הטעיה בה?

```
for (let i = 0; i >= 0; i = i++) {
  console.log('Iteration number ' + i);
}
```

הבעיה בollowאה הzo היא שהיא לא תעוצר לעולם. למה? כי היא תתקיים כל עוד גודל מ-0, ו- גודל ב-1 כל הזמן ולעולם לא יהיה קטן מ-0.ollowאה הzo, שלא תסתהים לעולם, נקראתollowאה אינסופית. בדרך כלל, דברים כאלה יקרו אחריו המן איטרציות. אם מרים את זה בדף, מתישו הלשונית תהיה לא יציבה ותקרוס.uschcotbimollowאה (ולא משנה מאי זה סוג) והדף קופה, סביר להניח שזאת הסיבה.

לולאת while

לולאה נוספת היא **לולאת while**. היא פשוטה יותר מlolאת for ומכליה תנאי אחד שכל עוד הוא נכון הלולאה מתקיימת. למשל:

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

כל עוד התנאי מתקיים, הולאה רצה.

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

כל עוד התנאי הזה שווה ל-true
הולולה תרוץ

בלולאות אלו, כמובן, חייבים להגיד את מהו לולאה, אחרת בכל ריצה של הלולאה יתאפשר משתנה ה-z.

הנתנאי יכול להיות כל תנאי שהוא, כולל תנאים מורכבים, ובגלל זה הלוואה זו שונה מילולאית זו שמקובל לכתוב אותה רק עם תנאי אחד. למשל הלוואה המורכבת זו:

```
let i = 0;  
let n = 20;  
while (i < 100 && n > 0) {
```

```
console.log('n' + n);
console.log('i' + i);
i++;
n--;
}
```

היא עבדת כל עוד ח'יובי נ- קטן מ-100. ברגע שאחד משנייהם לא נכון, היא תעצור. קצת מרכיב ומלבלל, אבל כדאי לזכור את זה. אחד השימושים שלו הוא אם (מסיבה מסוימת) רוצים ליצור לולאה אינסופית, אז כתובים משהו זהה:

```
while (true) {
}
```

אבל צריכה להיות לכם סיבה טובה מאוד ליצור לולאה אינסופית.

לולאת **do** **while**

עוד סוג של לולאה זהה הוא **לולאת while do**. סוג הלולאה הזה מאד לא שכיח אבל כדאי להכיר אותו. הלולאה הזו ייחודית כי ביגוד לולאות **for** ולולאות **while**, היא תמיד מבצעת פעולה אחת ואז בודקת את התנאי, במקומם לבדוק את התנאי ואז לrox. כלומר, תמיד הלולאה רצה לפחות פעם אחת.

הינה דוגמה:

```
let i = 100;
do {
  console.log(i);
  i++;
}
while (i < 0); // 100
```

יש כאן מילה שומרה, **do**, ומײַד אחריה סוגרים מסולסלים. מה שיש בתוכם תמיד יקרה, ורק אחר כך יגיע ה-**while** שבודק את התנאי. כאן, למשל, הלולאה תתקיים כל עוד **i** הוא שלילי. לפניהם התנאי קבועים את **i** על 100. לפי כל הלולאות שלמדנו זה אומר שהלולאה לא תרוץ לעולם, אבל כיוון שימושם ב-**while do**, הלולאה תרוץ לפחות פעם אחת.

לולאת **forEach**

הלולאות שלמדנו עד כה הן לולאות שעובdot עם תנאים, אבל בעולם הג'אווהסקריפט מקובלות יותר לולאות שעובdot עם אובייקטים או עם מערכim באופן שלם יותר. למה צריך את זה? באחד התרגילים בפרק על מערכim ראייתם שאפשר למחוק כל איבר במערך. למשל:

```
let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
delete myArray[4];
console.log(myArray); // [1, 2, 3, 4, undefined, 6, 7, 8, 9, 10]
```

אפשר לראות שיש איבר ריק. אם משתמשים בלולאות **for** או **while** עלולים להסתבך כי אי-אפשר להיות בטוחים שהאיבר אכן מוגדר. במקום להתחליל לשבור את הראש על גודל המערך ועל התאמתו לולאת **for**, שלא לדבר על מה שיקרה אם יש איברים ריקים, יש דרכי אלגנטיות יותר לעבור על כל האיברים בו. הדרך הראשונה והידועה מכלן היא שימוש ב-**forEach**.

מדובר במתודה שקיימת אך ורק במבנה נתונים מסווג מערך (לא אובייקט) ומקבלת פונקציה אונומית עם שני ארגומנטים – `value` שהוא האיבר של המערך ו-`key` שהוא המספר של המפתח. איך זה עובד? ככה:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
// "value: first, key: 0"
// "value: second, key: 1"
// "value: third, key: 2"
```

יש כאן מערך בשם `myArray` ובו שלושה איברים. האיבר במיקום 0 הוא מחוץ טקסט `first`. האיבר במיקום 1 הוא מחוץ טקסט `second` והאיבר האחרון, במיקום 2, הוא מחוץ טקסט `third`. כיוון שמדובר במערך, אפשר להציגו `:forEach` את מתודה `myArray.forEach();`

המתודה הזאת היא פונקציה שצריכה לקבל ארגומנט. מה הארגומנט? פונקציה נוספת, שבדרך כלל היא אונומית. על פונקציות חוץ אונומיות למדנו בפרק על הפונקציות. הפונקציה שאותה מעבירים פה תרוץ בכל איבר של המערך. הארגומנטים שלה הם האיבר עצמו והמספר שלו:

```
(value, key) => {
  console.log(`value: ${value}, key: ${key}`);
}
```

זה כמו לכתוב משהו זהה:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach(iteratorFunction);
function iteratorFunction(value, key) {
  console.log(`value: ${value}, key: ${key}`);
}
```

הקוד שלעיל זהה כמעט לחלוטין לקוד של פונקציה אונומית, אבל פה יוצרים עוד פונקציה ו"מלככים" את הסkop. מקובל מאוד להשתמש בפונקציה אונומית זהה מה שעשווים כאן. הפונקציה האונומית רצה על כל איבר. מאיפה באים הארגומנטים `value` ו-`key` בלי שקראו להם? זה הכוח של `forEach`, הוא שותף אותם שם.

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
```

הבה נבחן את ההרצה. בפעם הראשונה, האיבר הראשון מגיע לפונקציה:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
```

מה שיש בפונקציה מופעל. זה הכוח של `forEach`. אל תתבללו מהסינטקס המוזר עם החז', בסופו של דבר זו פונקציה. מה שיש בתוך הסוגרים המסורתיים רץ ורך הארגומנטים מתחלפים. בሪזה הראשונה של `forEach`, תור האיבר הראשון להיכנס לפונקציה האונומית. יש `value` שהוא האיבר של המערך ויש `key` שהוא המקום של האיבר במערך. למה 0? כי מערך תמיד מתחילה מ-0.

אחרי שהאיבר הראשון רץ והפונקציה מסתיימת, מגיע תור האיבר השני:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
```

הפונקציה רצתה שוב, אבל ה-key וה-value שלה שונים. הפעם אלו ה-key וה-value של האיבר השני. ה-value הוא מה שיש בתוך האיבר השני במערך, ובמקרה זה - מחזורת טקסט. ה-key הוא המקום של האיבר השני במערך. למה 1? כי מערכים מתחילה מ-0.

בפעם השלישייה והאחרונה, האיבר השלישי ייכנס לפונקציה האונומית:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
```

שםות הארגומנטים כਮובן נתונים לבחירה. גם אם תקראו להם arg1 ו-arg2 הם יעבדו:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((arg1, arg2) => {
  console.log(`value: ${arg1}, key: ${arg2}`);
});
// "value: first, key: 0"
// "value: second, key: 1"
// "value: third, key: 2"
```

מה החיסרון של הלולאה? עם `forEach` אי-אפשר לשבור את הלולאה. היא תרצה לכל אורך איברי המערך. למה רוצים לשבור את הלולאה? בגלל אלף ואחת סיבות. בדרך כלל אם מחפשים משהו ומוצאים אותו, אין צורך בהרצת כל הלולאה. בדיק בשביל זה יש את לולאת `.for of`.

הפונקציה שאנו מעבירים יכולה להיות גם לא אונונימית. אבל מקובל להעביר אונונימית.

לולאת for of

לולאת `of` פועלת כך: היא לא מחדירה את האינדקס של האיבר במערך, אבל כן מקבלים את האיבר. הבה נניח שיש מערך שהאיברים שלו הם אובייקטים של חפצים. יש את החפץ ויש את המחיר. מהו בסגנון זהה:

```
let orderArray = [{ name: 'pen', price: 11 }, { name: 'pencil', price: 5 }, { name: 'TV', price: 2345 }];
```

לא צריך להיבהל. מדובר במערך שבמוקם מספרים או מחרוזות טקסט יש בו אובייקטים. לכל אובייקט יש שתי תכונות – `name` ו-`price`. `name` מכיל מחרוזת טקסט ו-`price` מכיל מספר. לולאת `for` שתעביר עליו תיראה כך:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
for (let order of orderArray) {
  console.log(`The price of ${order.name} is ${order.price}.`);
}
// "The price of pen is 11."
// "The price of pencil is 5."
// "The price of TV is 2345."
```

מה יש פה? שימוש במילה השמורה `for`, שאotta הכרנו בלולאת `for`. אך במקום הסינטקטו המוכר יותר, יש הגדרת משתנה שהוא בעצם האיבר התווך במערך, המילה השמורה `of` ושם המערך. בתוך הסוגרים המסולסלים ירוץ בכל פעם האיבר התווך במערך:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
for (let order of orderArray) {
  console.log(`the price of ${order.name} is ${order.price}`);
}

```

משתנה לפי האיבר במערך

הבה נראה את האיטרציה הראשונה:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
for (let order of orderArray) {
  console.log(`the price of ${order.name} is ${order.price}`);
}
{ name: 'pen', price: 11 },

```

באייטרציה הראשונה, האיבר הראשון הוא שנכנס לתוכה המשתנה `order`. יש לו תוכנה בשם `name` ותוכנה בשם `price`. כיוון שאנחנו מדפיסים אותו, מה שראים הוא:

```
// "The price of pen is 11."
```

בהרצתה השנייה, האיבר השני נכנס אל תוך המשתנה זהה ומה שיש בסוגרים המסוללים רץ שוב:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
for (let order of orderArray) {
  console.log(`the price of ${order.name} is ${order.price}`);
}
{ name: 'pencil', price: 5 },
```

מה שראים בהדפסה הוא:

```
// "The price of pencil is 5."
```

בהרצתה الأخيرة, האיבר השלישי ייכנס אל תוך המשתנה `order` בדיק כמו האיברים שלפניו. עכשו נבעור לשבירת הלולאה. בואו נניח שהדרישה היא להציג את המוצר הראשון שהמחיר שלו נמור מ-10. לא את כל המוצרים אלא את המוצר הראשון. דרישת זו יכולה כמובן להיות אמיתית לחלוטין. איך שוברים את הלולאה? באמצעות המילה השמורה `:break`:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
let answer;
for (let order of orderArray) {
  console.log(`I am passing ${order.name}.`);
  if (order.price < 10) {
    answer = order;
    break;
  }
}
// "I am passing pen."
// "I am passing pencil."
```

מה יש פה? תנאי שאומר שברגע שמחיר של מוצר כלשהו קטן מ-10, מכניסים את המוצר למשתנה `answer` ושוברים את הלולאה. אפשר לראות באמצעות העורות שהלולאה אכן נשברת ולא ממשיכת אל האיבר השלישי כיון שהאיבר השני שעונה לתנאי מוחזר ו-`break` מופעלת.

כך למשל באיטרציה הראשונה רואים:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
let answer;
for (let order of orderArray) {
  console.log(`I am passing ${order.name}.`);
  if (order.price < 10) { לא עונת לתנאי
    answer = order;
    break;
  }
}
{ name: 'pen', price: 11 },
```

ש הדפסה של:

```
// "I am passing pen."
```

אך התנאי אינו מופעל כיון שהמחיר הוא 11 והתנאי הוא מחיר שקטן מ-10.

באיורציה השנייה:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
let answer;
for (let order of orderArray) {
  console.log(`I am passing ${order.name}.`);
  if (order.price < 10) {
    answer = order;
    break;    הלויה נשבה
  }
}

```

ש הדפסה:

```
// "I am passing pencil."
```

כיוון שהתנאי מתקיים, המשפט `break` מופעל והלויה לא עוברת לאיורציה השלישית. זה היתרון הגדול של `for of` – שהוא מציג את היכולת של `while` לבצע שירה ואת האלגורניות של `.forEach`.

לולאת **map**

פעמים רבות רוצים לשנות את המערך עצמו. למשל, הבה נניח שהקוו שמייע מישראל רואה את המחרים ורוצים להציג לו אותם בשקלים, כמו לuplicル להכפיל את המחיר פי ארבעה. אף עושים את זה? אפשר להשתמש בלולאת `for`, `while`, `forEach` או בלולאת `map` — אך אם רוצים לשנות את האובייקטים בתוך המערך מקובל מאוד להשתמש ב-`map`, שמאחורי הקווים לוקחת את האובייקטים במערך הקווים ומשנה אותם ומחזירה את המערך הקווים עם המשתנים החדש. לולאת `map` מיועדת לשינוי אובייקטים מערכים או במערכות והיא פועלת באופן דומה ל-`forEach`, אלא שהפונקציה האנוונימית יכולה להחזיר את האובייקט החדש שייכנס במקום האובייקט המקורי, וכך לעשות לו מוטציה. בואו נבדוק את הדוגמה:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value;
});
console.log(orderArray);
/*
[
  { name: 'pen', price: 44 },
  { name: 'pencil', price: 20 },
  { name: 'TV', price: 9380 }
]
*/
```

זה דומה מאוד לולאת `forEach` שעליה כבר למדנו. ה-`value` שהפונקציה הפנימית מקבלת הוא האיבר שיש במערך. אבל אפשר לראות שכן הפונקציה מחזירה משהו. מה היא מחזירה? את האובייקט שנמצא באיבר החדש שעבר שינוי. האובייקט החדש שייכנס במקום האיבר הקודם, כולל כל השינויים שהכנסתם אליו:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value; → { name: 'pen', price: 44 },
});

```

נכנס למקום הראשוני
במערך המקורי

אפשר להציג מה שרצים, אפילו מחרוזת או אובייקט מורכב יותר. כל מה שמכניסים ייכנס במקום האיבר הראשון של המערך. באירועה השנייה משנים את האיבר השני, ומה שמחזירים נוכנס במקום האיבר השני. הנה:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value; → { name: 'pencil', price: 20 },
});

```

נכנס למקום השני
במערך המקורי

ולסימן, באיטרציה האחורונה מקבלים את האיבר השלישי וממה שמחזירים ייכנס למקום השלישי
במערך המקורי:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }  
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value; → { name: 'TV', price: 9380 }
});
```

נכנס למקום השלישי
זהו האחרון במערך
ה המקורי

זהו. מה שיש במערך המקורי הוא לא מה שהוא פעם אלא האובייקטים החדשים.

אם אנו משנים מערך פשוט קיימ, העריכים לא השתנו ויש להורות לפונקציית `map` להחזיר את המערך שהשתנה לערך אחר:

```
let orderArray = [11, 12, 13, 14];
let newArray = [];
newArray = orderArray.map((value) => {
  let newPrice = value * 4;
  return newPrice;
});
console.log(orderArray); // [11, 12, 13, 14]
console.log(newArray); // [44, 48, 52, 56]
```

lolāt filter

הlolāt האחורונה היא lolāt שמצויה איברים מהמערך. בעוד **map** רק משנה איברים, באמצעות lolāt **filter** אפשר למצוות את המערך. הבה נסתכל על המערך הזה, למשל:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
```

נניח שמדוברים להציג ללקוח רק דברים שהמחיר שלהם גבוה מ-7. אפשר לעשות משהו כזה:

```
writingArray.forEach((value, key) => {
  if (value.price <= 7) {
    delete writingArray[key];
  }
});
```

כלומר, ליקח את המערך ולבור עליו עם lolāt. בכל איטרציה (מעבר) של lolāt בודקים את המחיר של האיבר הנוכחי. אם הוא קטן מ-7 או שווה לו, מוחקים אותו. האיבר הראשון, למשל, הוא `pen`, שה-`price` שלו הוא 11. התנאי לא יתקיים כי `value.price` הוא 11 והוא לא קטן מ-7 או שווה לו. האיבר השני `pencil` וה-`price` שלו הוא 5. במקרה הזה התנאי מתקיים. כיוון שלlolāt ה-`forEach` נותנת גם את ה-`key`, הלא הוא מספר האיבר, שבמקרה של האיבר השני הוא 1 (זכרים שמערך מתחילה מ-0, נכון?), אפשר להשתמש באופרטור `delete`, שעליו למדנו בפרק על מערכים, כדי למחוק את האיבר הזה, וכך הלאה. מה שנקבל בסוף הוא מערך עם שני `undefined`, היכן שהוא כל האיברים שמחירים נמוך מ-7.

```
[  
  { name: 'pen', price: 11 },  
  undefined  
  { name: 'paper', price: 10 },  
  { name: 'brush', price: 12 },  
  undefined  
]
```

זה קצת מbas, ואפשר לעקוף את זה בכל מיני דרכים או באמצעות `filter`. מדובר בlolāt שזהה ל-`map` אך הפונקציה שרצה בה מחזירה רק `true` או `false`. אם היא מחזירה `true` האיבר נשאר במערך. אם היא מחזירה `false` האיבר מופוגג מהמערך כאילו לא היה שם. בנויגוד לlolāt `map`, lolāt הזו לא משנה את המערך המקורי וצריך להחזיר את התוצאה למשתנה אחר.

הינה דוגמה:

```
const writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filteredArray = writingArray.filter((value) => {
  if (value.price <= 7) {
    return false;
  } else {
    return true;
  }
});
```

writingArray הוא מערך המכיל אובייקטים של מוצרי כתיבה. לכל אחד מהם יש שם ומחיר. כדי לסנן את כל האיברים שהמחירם שלהם נמוך מ-7, משתמשים בפונקציה 'יחודית' למערך שנקראת filter. הפונקציה הזאת מקבלת פונקציה אונונימית שאוותה כתובים כפונקציה חז. אני רק מזכיר - פונקציית חז היא דרך מסוימת לכתוב פונקציה אונונימית (לומר פונקציה שאין לה שם אלא עוברת כארגומנט). הפונקציה האונונימית מחזירה true או false. אם היא מחזירה true האיבר נשאר במערך. אם היא מחזירה false הוא עף מהמערך. פונקציית filter ממחזירה את המערך החדש.

הבה ננסה לפרק את זה. בסבב הראשון, מי שנכנס לטור הבדיקה של ה-`filter` הוא האיבר הראשון:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) { התנאי לא מתקיים
    return false;
  } else {
    הפונקציה מחזירה true
    האיבר זוכה להישאר במערך
  }
});
```

באיטרציה השנייה מגיע תור האיבר השני:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) { התנאי מתקיים
    return false; הפונקציה מחזירה false
  } else {
    האיבר הזה לא זוכה להיכנס לאיבר החדש
    return true;
  }
});
```

הוא כבר לא זוכה להיכנס, כיון שהמחיר שלו הוא 5 והתנאי הוא שאלם מ-7 או שווה לו מקבל `false`. וכך הלאה. כמו כל LOLAH אחרת, הלולאה הזאת עוברת על כל המערך. הפונקציה שהעבכנומחזירה `true` או `false` לגבי כל איבר. האיברים שמקבלים `true` זוכים להיכנס למערך

החדש, ואלו שמקבלים `false` – לא. בסופו של תהליך חיבים להחזיר את מה ש-`filter` מחזירה אל משתנה חדש. המערך המקורי נשמר כמו שהוא:

חיבים להחזיר את התוצאה אל מערך חדש

```
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) {
    return false;
  } else {
    return true;
  }
});
```

lolæt sort

lolæt חשובה נוספת המשמשת למינון מערכיים. lolæt זה לא משנה איברים ולא את גודל המערך, אלא פשוט מסדרת את האיברים במערך. ביגוד לאייטרציות אחרות, לא חיבים להעביר פונקציה אונימית שתעשה את הסדר. כבירית מחדל האיטרציה זה מסדרת לפי סדר הא-ב' – היא ממיר את הערכים למחוזות טקסט ועורך השוואת ביניהם, וכך מבצעת את הסדר. הנה דוגמה פשוטה מאוד שמדגימה את זה:

```
let months = [ 'March', 'Jan', 'April', 'Dec' ];
months.sort();
console.log(months); // [ "April", "Dec", "Jan", "March" ]
let numbers = [ 1, 42, 5, 7, 565656 ];
numbers.sort();
console.log(numbers); // [ 1, 42, 5, 565656, 7 ]
```

ובן שהענינים מתחילה האם במערך יש אובייקטים ולא רק מספרים או מחוזות טקסט, אבל לא נסעה את אייטרצית sort במלואה בסופר זהה.

lolæt על אובייקטים

כל lolæt שולמדנו עד כה מחקות יפה עם מערכיים ובאמצעותן אפשר לעבור על כל איבר או איבר במערך. אך נשאלת השאלה – איך לעבור על אובייקט? הכוונה למשהו בסגנון זהה:

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
```

מדובר באובייקט שיש בו נתונים על משתמש מסוים. נניח שרוצים להדפיס את כל נתונים האובייקט (בלי לדעת מה שמות התכונות שלו). איך עושים את זה? lolæt for או lolæt while לא יעזור ולולאות for of או forEach או forLoop לא מיעודות למערכיים בלבד ויניבו שגיאות אם תנסו להשתמש בהן. יש שתי שיטות למעבר על אובייקטים. הראשונה "לפי הספר", שמעטים משתמשים בה, והשנייה הפופולרית יותר.

lolæt in for

מדובר בlolæt שעוברת על כל תכונות האובייקט בדומה ל-of-for. כך היא נראה:

```

const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
}
/*
"key: userId, value: 12"
"key: userName, value: barzik"
"key: age, value: 40"
*/

```

לולאת `for-in` מכילה כמה חלקים. ראשית המילה השמורה `for`, ומיד אחריה סוגרים עגולים. בסוגרים העגולים מגדירים משתנה שהוא המפתח. בתוך הלולאה יקבל המשתנה זהה את שם המפתח המקורי. אחר כך מכניסים את המילה השמורה חוץ ואת המשתנה שבתוכו נמצא המערך.

בתוך הסוגרים המסוללים מתחוללת האיתרציה של כל מפתח. ברגע שיש את המפתח `key`, אפשר לגשת אל הערך שלו באמצעות סוגרים מרובעים כפי שמדנו בפרק על האובייקטים. המפתח של האובייקט הוא המשתנה `key` והערך הוא:

`userObject[key]`

באו נראה איך זה קורה. ראשית, חשוב להבין ש-`key` מגיע מהמפתחות של האובייקט. הוא לא מכיל את הערך:

```

const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
}

```

באייטרציה הראשונה שמתבצעת, `key` הוא המפתח הראשון שיש באובייקט, הלווא הוא מחזצת הטקסט `userId`. באמצעות המפתח אפשר לחוץ את הערך. איך? למדנו בפרק על אובייקטים שאם המפתח נמצא בשם של משתנה אפשר לחוץ אותו באמצעות שימוש בסוגרים מרובעים, זהה בדיק מה שעושים. את המפתח ואת הערך מדפיסים באמצעות שימוש בתבנית טקסט פשוטה, שגם עליה למדנו:

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};

for (let key of userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
}

userObject[userId] = 12
```

באייטרציה השנייה, key הוא המפתח השני של האובייקט, במקרה זה userName. ושוב גם כאן מחליצים את הערך באמצעות שימוש בסוגריים מרובעים כמו בפעם הקודמת:

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};

for (let key of userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
}

userObject[userName] = 'barzik'
```

באייטרציה האخונה, key הוא age.

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};

for (let key of userObject) {
  key = age
  console.log(`key: ${key}, value: ${userObject[key]}`);
}

userObject[age] = 40
```

הולאה חזן ניתנת לשבירה על ידי break;

בסק הכל פשוט ונחמדה, נכון? אבל האם היא שנדיר למצוא מתכנת שמשתמש בשיטה זו, מפני שלולאת `for` מביאה נתונים גם מאובייקטים אחרים שהם **פרוטוטיפ** – אב טיפוס – של האובייקט שלנו. כרגע זה נשמע סינית ועדיין לא דיברנו על זה, אבל על קצה המזלג אצין שבדרכ כל אובייקטים לא מגיעים כלוח חלק והם העתק של אובייקטים אחרים שיש להם מפתחות נוספים. לרוב לא מעוניינים ב מפתחות של האובייקטים האחרים, ולפיכך לוולאת `for` הוא עלולה להיות הרת אסון, וזה הסיבה שבקושי משתמשים בה. אם כבר משתמשים בה, מקובל מאוד לוודא שהתוכנה שבודקים היא התוכנה של האובייקט, באמצעות שימוש בפונקציה הייחודית לאובייקטים שנקראת `hasOwnProperty`, באופן הבא:

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) {
  if (userObject.hasOwnProperty(key)) {
    console.log(`key: ${key}, value: ${userObject[key]}`);
  }
}
```

ואם כל מה שכתבתי נשמע לכם כמו סינית זה בסדר. זכרו שלולאת `in` לשימוש באובייקטים היא בעיתית ונדר שמשתמשים בה.

Object.keys

הדרך השנייה והפולולית ביותר היא להשתמש בפונקציה שנקראת `Object.keys` לחילוץ כל המפתחות של האובייקט כמערך, ואז לעבור על המערך זהה באיזו לולאה שרצו, כי זה המערכת. איך ניגשים לעיר באובייקט? כמו בלולאת `in`; ברגע שיש את המפתח יש גם את הערך.

שימוש לב דוגמה הבאה:

```
const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
const userObjectKeys = Object.keys(userObject); // ["userId",
"userName", "age"]
userObjectKeys.forEach((key) => {
  console.log(`key: ${key}, value: ${userObject[key]}`);
});
/*
"key: userId, value: 12"
"key: userName, value: barzik"
"key: age, value: 40"
```

מה קורה פה? יש כאן את אותו אובייקט מהדוגמה הקודמת. באמצעות שימוש בפונקציית `Object.keys` מקבלים את כל המפתחות כמערך שנכנס ל-`userObject`, ועליו עושים `forEach`. כיוון שמדובר במערך, אפשר לעשות עליו `forEach`. נהדר ואלגנטי. סכム בטבלה המסבירה על כל סוג הולאות:

שם הולאה	יתרונות	חרונות
ולואות למערכות		
ולואת <code>for</code>	פשוטה לימוש ניתנת לשכירה באמצעות <code>break</code>	לא תנאים מורכבים צריך לדעת את גודל המערך קל להיכנס לולאה אינסופית מגיעים גם לאיברי <code>undefined</code>
ולואת <code>while</code>	פשוטה לימוש ניתנת לשכירה באמצעות <code>break</code> אפשר לכתוב תנאים מורכבים	צריך לדעת את גודל המערך קל להיכנס לולאה אינסופית מגיעים גם לאיברי <code>undefined</code>
ולואת <code>forEach</code>	לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים	צריך לכתוב פונקציית ח'ז לא ניתן לשכירה
ולואת <code>for of</code>	פשוטה לימוש ניתנת לשכירה באמצעות <code>break</code> לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים	מקבלים רק את הערך של אברי המערך צריכים לחלץ את המפתח בלבד
ולואת <code>map</code>	ולואה שעוברת על המערך ויוצרת ממנו מערך חדש באותו גודל. אפשר לשנות את האיברים בקלות.	לא מוחקת איברים מהמערך
ולואות להמרת מערכים		
ולואת <code>filter</code>	מוחקת איברים	צריכים להחזיר את התוצאה למשתנה אחר לא משנה את המערך המקורי
ולואות לאובייקטים		
ולואת <code>in</code>	פשוטה לימוש ניתנת לשכירה באמצעות <code>break</code> לא צריך לדעת את גודל האובייקט אין לולאה אינסופית לא נכנסים לאיברים ריקים	מקבלים רק את המפתח של האובייקט צריכים לחלץ את הערך בלבד חייבים להשתמש ב- <code>hasOwnProperty</code>

מקבלים רק את המפתח של האובייקט ואricsים לחלק את הערך בלבד אפשר להשתמש בלוואות של המערך ואז מקבלים את כל החסרונות של הלוואת בחירהם	אפשר להשתמש בלוואות של הערך ואז מקבלים את כל היתרונות של הלוואתבחירהם	שימוש ב- Object.keys
---	---	-------------------------

תרגיל:

באמצעות לולאות `for`, הדפיסו שלוש פעמים על המסך "I know how to use for loop".

פתרונות:

```
for (let i = 0; i < 3; i++) {
  console.log('I know how to use for loop');
}
```

הסבר:

זו לולאת `for` פשוטה שמתחליה כאשר `i` שווה ל-0. היא אמורה להימשך כל עוד `i` קטן מ-3. באיטרציה הראשונה כאמור `i` שווה ל-0. בשנייה הוא שווה ל-1, בשלישית הוא שווה ל-2 ובריביעית הוא שווה ל-3 ולא עונה על התנאי. לפיכך תהיה הדפסה רק שלוש פעמים.

תרגיל:

נתון מערך של שמות לקוחות:

```
const customersArray = ['Moshe', 'Yaakov', 'Yossi', 'Baruch', 'Yael'];
```

צרו לולאה המדפסה את שמות הלקוחות.

פתרונות:

```
const customersArray = ['Moshe', 'Yaakov', 'Yossi', 'Baruch', 'Yael'];
for (let i = 0; i < customersArray.length; i++) {
  console.log(customersArray[i]);
}
```

הסבר:

משתמשים בלולאת `for`. המונה הוא `i` ששויה ל-0 והתנאי הוא שהלולאה תרוץ כל עוד המספר קטן מאורך המערך, במקורה הזהה 5. משתמשים במונה `i` שזמן על מנת להדפיס את האיבר המתאים במערך. למשל, באיטרציה הראשונה `i` שווה ל-0. התנאי מתקיים כי `i` 0 קטן מ-5 (אורך המערך). משתמשים ב-`i` על מנת להדפיס את האיבר שנמצא במקומות 0. באיטרציה השנייה, `i` שווה ל-1, התנאי מתקיים כי 1 קטן מ-5 (אורך המערך) ומשתמשים ב-`i` כדי להדפיס את האיבר שנמצא במקומות 1, וכך הלאה.

תרגיל:

נתון מערך של אובייקטי הזמנות:

```
const invoices = [{ id: 1, price: 10 }, { id: 2, price: 32 }, { id: 3, price: 40 }];
```

הדפiso את סכום הזמנות במערך.

פתרונות:

```
const invoices = [{ id: 1, price: 10 }, { id: 2, price: 32 }, { id: 3, price: 40 }];
let amount = 0;
for (let i = 0; i < invoices.length; i++) {
    amount = amount + invoices[i].price;
}
console.log(amount); // 82
```

הסבר:

יצרים לולה שעוברת על המערך. המערך הוא לא של ערכים פרימיטיבים אלא של אובייקטים, אבל גם אובייקטים אפשר לשלוף באמצעות `[i]`, כשותנה לפי מספר האיטרציה. לפני שהלולה רצה, יוצרים משתנה בשם `amount` בשם שווה ל-0, ובכל פעם מוסיפים לו את ה-`price` באובייקט שיש באיטרציה. זה נעשה באמצעות הנקודה. להזכירם, אפשר לגשת למפתח מסוים באמצעות שם האובייקט והנקודה. במקרה זה האובייקט נמצא באיבר במערך ולא במשתנה עם שם משלו, אז אפשר לגשת למחיר באמצעות:

`invoices[i].price`

כשותנה, להזכירם, לשותנה לפי האיטרציה.

תרגיל:

צרו לולאת while שמקבלת מספר וסופרת ממנו עד 0.

פתרון:

```
let i = 10;
while (i >= 0) {
  console.log(i);
  i--;
}
```

הסבר:

לולאת while פשוטה שעובדת כל עוד i גדול מ-0 או שווה לו. בלולאה עצמה מדפיסים את המספר ואז מורידים 1 מ- i באמצעות כתיב מקוצר.

תרגיל:

צרו לולאת while שמקבלת מספר וסופרת ממנו עד 0. אם המספר הוא 0 או שלילי עדיין תהיה הדפסה אחת של המספר, למשל -1 וזהו.

פתרון:

```
let i = -1;
do {
  console.log(i);
  i--;
} while (i >= 0);
```

הסבר:

לולאת while so היא אידיאלית אם רוצים פעולה אחת לפחות, גם אם התנאי לא מתקיים. במקורה הזה קודם הלולאה עובדת לפחות פעם אחת אז התנאי נבדק. כיוון שהמספר הוא שלילי, התנאי לא מתקיים והלולאה לא תמשיך לroz, אבל היא עבדה פעם אחת והדפסה את המספר.

תרגיל:

נתון מערך:

```
let myArray = [
    { userName: 'Moshe', age: 20, },
    { userName: 'Yaakov', age: 25, },
    { userName: 'Ran', age: 40, },
]
```

מדובר בפרופילים של משתמשים באתר היכרויות. כתבו פונקציה `forEach` שתדפיס בקונסולה אך ורק את המשתמשים שהם בני פחות מ-30.

פתרונות:

```
let myArray = [
    { userName: 'Moshe', age: 20, },
    { userName: 'Yaakov', age: 25, },
    { userName: 'Ran', age: 40, },
];
myArray.forEach((value, index) => {
    if (value.age < 30) {
        console.log(value.userName);
    }
});
/*
"Moshe"
"Yaakov"
*/
```

הסבר:

משתמשים בפונקציית `forEach` שמקבלת פונקציה אונומית מסווג חז. הפונקציה זו נראית כה:

```
(value, index) => {
    if (value.age < 30) {
        console.log(value.userName);
    }
}
```

לא צריך להיבהל ממנה. זו פונקציה רגילה שרצה על כל איבר במערך באופן אוטומטי. יש ערך שבחרטוי לקרוא לו `value` והמספר של האיבר במערך. מה שמשמעותו הוא הערך. הערך הוא בעצם האיבר התווך במערך. בכל איטרציה יש איבר אחר. באיטרציה הראשונה מדובר באיבר הראשון, באיטרציה השנייה מדובר באיבר השני ובאיטרציה השלישית מדובר באיבר השלישי. זה הכל. בודקים את האובייקט שיש באיבר או, נכון יותר, את תוכנת `age` שלו, ומדפיסים את מי שהוא פחות מ-30.

תרגיל:

נתון מערך (אותו מערך כמו בתרגיל הקודם):

```
let myArray = [
```

```

    { userName: 'Moshe', age: 20, },
    { userName: 'Yaakov', age: 25, },
    { userName: 'Ran', age: 40, },
];

```

צרו פונקציה שמקבלת מספר וערך ומחזירה אך ורק מערך משתמשים שהגיל שלהם שווה למספר זהה או גדול ממנו.

פתרונות:

```

function getUsersAge(requestedAge, allUsersArray) {
    const answer = allUsersArray.filter((value) => {
        if (value.age < requestedAge) {
            return false;
        } else {
            return true;
        }
    });
    return answer;
}
const age40Users = getUsersAge(40, myArray);
console.log(age40Users); // [{ userName: 'Ran', age: 40, }]

```

הסביר:

יצרים פונקציה שמקבלת שני ארגומנטים – מספר מסוים וערך. משתמשים בפונקציית `filter` על מנת לסנן את המערך ולהשאיר שם רק את המשתמשים שרצים. איך נראה פונקציית `filter` בבדיקה כהה:

```

(value) => {
    if (value.age < requestedAge) {
        return false;
    } else {
        return true;
    }
}

```

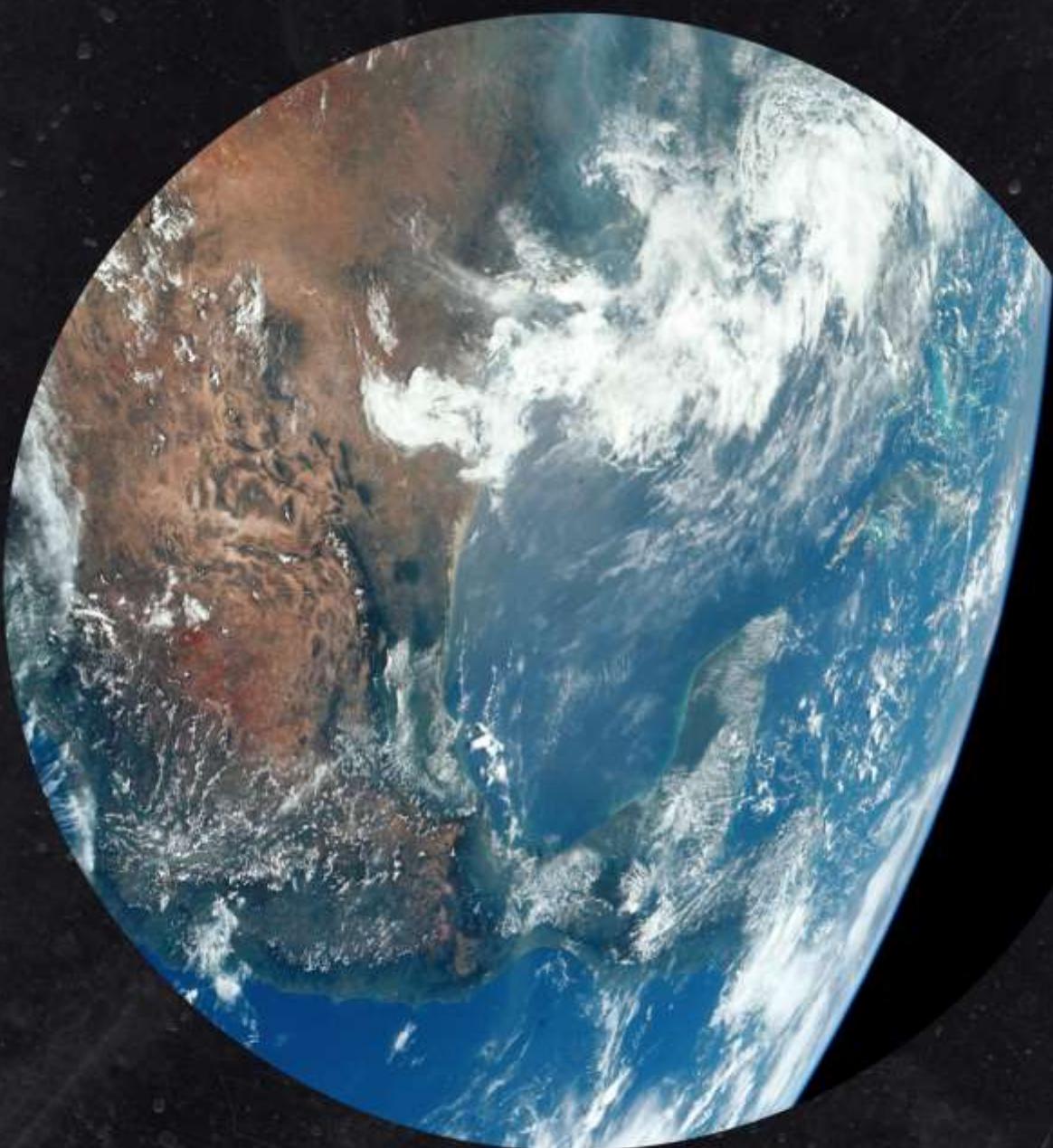
הפונקציה פשוטה יחסית ולא צריך להתבלבל מהחץ. ב-`value` יש בכל פעם איבר אחר מהמערך. אם פונקציית החץ מחזירה `true` הוא נשאר במערך. אם לא, הוא עף החוצה בובותה פנים. במקרה הזה מחזירים `false` על כל מי שהגיל שלו קטן מהתמורה `requestedAge`, שאוטו מקבלים מהפונקציה.

אל תשחחו שחייבים להחזיר את מה שמתקיים מה-`filter` אל משתנה אחר, כיון שפונקציית `filter` לא משנה את המערך המקורי.

כל מה שנותר לעשות הוא לבדוק את הפונקציה. מعتبرים לה 40, לדוגמה, ואת המערך של המשתמשים וראים שמקבלים משתמש אחד שהוא בן 40, כיון ש-40 לא קטן מ-40, הוא מקבל `true` בפונקציית הфиילטר.

פרק 13

ג'אואסקרייפט בסביבת דפסן



ג'אווהסקרייפט בסביבת דף

עד כה, כל התרגולים היו עם הקונסולה בלבד. סביבת העבודה היא בדף ועם עורך טקסט, אבל בגדול הדבר היחידי שעשיתם בדף היה בקונסולה. הגיע הזמן להתחיל למש את הידע שלמדנו וראות מה אפשר לעשות בעזרתו במצבות.

יום ג'אווהסקרייפט רצה ברגעון אדיר של סביבות, אבל בעבר הסביבה היחידה שבה היא רצתה הייתה הדף. כאמור, באתר אינטרנט השתמשו בה כדי ליצור התנהלות אינטראקטיבית עם המשתמש, אнимציות, אפקטים, וידאו ועוד. באופן עקרוני, כל הידע שלמדנו עד עכשיו הוא הבסיס הנדרש ליצירת כל הדברים האלה בדף. יכול להיות שלולאות, אובייקטים ופונקציות נראים לא מעניינים לעומת אнимציות וסרטונים, אבל הם מה שפועל מאחורי הקלעים כדי לאפשר את כל הדברים היפים בדף.

הספר הזה מלמד ג'אווהסקרייפט ולא HTML, שהוא עולם ומלאו. בפרק זהה נלמד HTML באופן בסיסי בלבד, רק מספיק כדי להבין איך ג'אווהסקרייפט קשורה לכך. לשפת ג'אווהסקרייפט יש קשר הדוק עם ה-HTML של הדף.

הסבר כללי על HTML

ראשי התיבות של HTML הם Hyper Text Markup Language והוא בעצם הטקסט המרכיב את כל דפי האינטרנט שאתה מכיר. כשנכנסים אל דף כלשהו בראשת, ולא משנה אם מדובר באתר חדשות או באתר משחקים או סרטים – רואים דף HTML. דף זה הוא בעצם דף טקסט. בדיקן כמו קוד ג'אווהסקרייפט שאפשר לכתוב ב-notebook, גם קוד HTML הוא קוד שאפשר לכתוב בכל מקום. אם אתם רוצים לראות את קוד ה-HTML של כל דף, לחוץ על הכptror הימני בעבר וاذ על "הראה מקור" או "source view". תראו מיד המונ-המוני טקסט עם סימנים כאלה < ו > ו < ו >. הדברים שנמצאים בתחום החצים נקראים "אלמנטים של HTML" ויכולים להכיל טקסט או אלמנטים אחרים. אלמנט בסיסי של HTML נראה ככה:

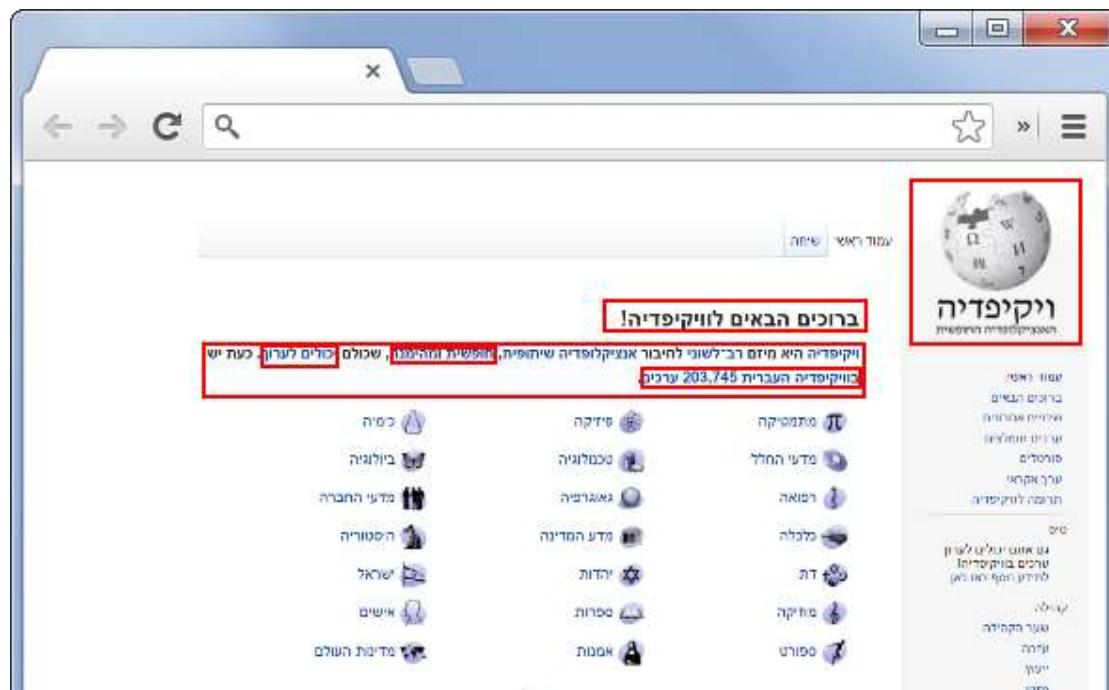
```
<elementName>Element Content</elementName>
```

אלמנט מורכב מתגית פתיחה עם סוג האלמנט, תוכן האלמנט, במרקחה זהה טקסט פשוט, ותגית סגירה שמורכבת מלוון ושם האלמנט.

אם מדובר באלמנט שנקרא "אלמנט שנסגר מלאיו", הוא נראה כך:

```
<elementName />
```

בגוזל, כל דבר שראהים בדף אינטרנט כלשהו עשוי מאלמנטים של HTML. כל דבר. למשל:



כאן רואים את דף הבית של ויקיפדיה. כל אובייקט, קישור או טקסט בו הם אלמנט HTML. הקפתוי כמו אלמנטים בריבועים. למשל הלוגו של ויקיפדיה הוא אלמנט מסווג תמונה. הטקסט "ברוכים הבאים לוויקיפדיה!" הוא אלמנט מסווג `h1`, כלומר כותרת ראשית. יש גם אלמנטים בתוך אלמנטים. טקסט הפתיחה הוא אלמנט ויש לו אלמנטים בנים של קישורים (סימנתי כמו מהם). כאמור אלמנט HTML נראה כך:

```
<elementName attributeName="attributeValue" id="elementId"
class="elementClass">
</elementName>
```

ה-`elementName` הוא שם האלמנט. למשל `img` הוא תמונה, `h1` הוא כותרת ראשית, `h2` הוא כותרת שנייה, `h3` הוא פסקה, `a` הוא קישור, `div` הוא אלמנט כללי וכן הלאה. יש שירותים מסווגי אלמנטים.

לכל אלמנט יכולות להיות תכונות (attributes). למשל, אם יש קישור, אחת התכונות של האלמנט היא המיקום שלו הינו קישור מפנה. התכונה הזו נקראת `href`. תכונה אחרת שיכולה להיות ל קישור היא `title` – הטקסט שמופיע אם מציבים את העכבר מעל הקישור. הנה דוגמה למה שראהים אם כתבים בתוך מסגר HTML אלמנט של קישור:

```
<a href="https://google.com" title="Link to google">Google.com</a>
```



אפשר לראות איך כל תכונה של האלמנט יכולה לשנות משזה בתנהוגותו.
יש אלמנטים שמכילים טקסט כמו קישור או פסקה:

```
<p>This is paragraph.</p>
```

יש כאלה שלא, כמו תמונה. למשל:

```

```

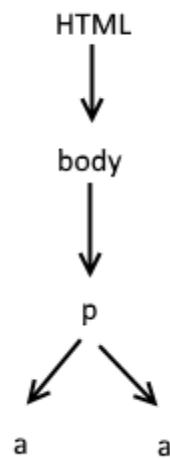
ההבדל הוא שהאלמנטים ללא הטקסט נסגרים בלוכסן בסופם ולא בשם האלמנט. כמו כן ממש לא כר:

```
</img>
```

כאמור, יכולים להיות אלמנטים בתוך אלמנטים.
הינה למשל פסקה שמכילה שני אלמנטים של קישור:

```
<p>There are several search engines like <a href="https://google.com" title="Link to google">Google</a> and <a  
href="https://duckduckgo.com/" title="Link to DuckDuckGo">DuckDuckGo</a>  
</p>
```

במקרה זהה המבנה של המסמך הוא:



כזכור ה-HTML עצמו שנחשב אבא, תגיית ה-`body`, ואז אלמנט הפסקה שהוא אלמנט האב. לאלמנט הפסקה (האב) יש שני ילדים שהם ה-`a` – שני קישורים, אחד לגוגל ואחד לדuckduckgo. בכל דף HTML ממוצע יש מאות אלמנטים כאלה מסוגים רבים ושונים ועל כלם אפשר להשפייע באמצעות ג'אווהסקריפט.

מזהים של תגיות HTML

אלמנטים של HTML יכולים להיות מזהים, משהו שבאזורתו אפשר לזרות את האלמנט לצרכים שונים. בדיקן כמו בני אדם שיש להם שם פרטי ושם משפחה, גם לאלמנטים יש שני מזהים. המזהה הראשון הוא כללי וקוראים לו `class`. לא להתבלבל עם קלאס של ג'אווהסקריפט. במקרה של `HTML`, `class` הוא מקביל לשם משפחה, בדיקן כפי שיש כמה אנשים בעולם עם שם המשפחה שלו, אפשר לתת את אותו `class` לכמה וכמה אלמנטים שונים. הם יכולים להיות בקשר אב-ילד או לא. אין הגבלה! אפשר גם לתת כמה שמות משפחה לאותו אלמנט. המזהה השני נקרא `id` והוא מזהה ייחודי. אסור לתת את אותו `id` לשני אלמנטים באותו דף. ה-`id` הוא ייחודי ומוחדר לכל אלמנט, ואפשר לתת `id` אחד בלבד לכל אלמנט.

אפשר כמובן להציג גם מזהה `class` וגם מזהה `id` לאלמנט אחד. במה משתמשים? תלוי במה שרצו לבצע ובמבנה הדף. לאלמנטים מיוחדים שרצו הגיעו אליהם בקלות באמצעות ג'אווהסקריפט נותנים `id` בלבד.

גישה אל תגיות באמצעות ג'אווהסקריפט

לתרגול, צרו דף HTML לדוגמה ותציבו בו ג'אווהסקריפט. הדף הזה זהה לדף שהסבירתי לעליון בפרק על בניית סביבת העבודה, אבל חשוב להזכיר על דבר אחד – דף דין מריץ את קוד הג'אווהסקריפט ומראנדר את האלמנטים בסדר מקבילי מלמעלה למטה. לצורך התרגול חשוב מאוד שקובצי הג'אווהסקריפט והקוד של הג'אווהסקריפט יהיו בתחתית הדף, כיוון שם הקוד

חפש את האלמנטים של ה-HTML או יתיחס אליהם, הדף דין יהיה חייב להכיר אותם לפני שהוא מרים את הקוד – ככלمر קובץ הג'אווהסקריפט שמתיחסים אלמנטים חייבים להיות מתחת לאלמנטים. זו הסיבה שבגללה יש להקפיד, לפחות כרגע, שקוד הג'אווהסקריפט יהיה מתחת לאלמנטים.

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <!-- HTML elements will be here -->
  <script src=".source.js"></script>
</body>

</html>
```

בקוד יש הערת HTML. כל מה שמדובר ב-HTML בתגיות <---> נחשב להערה ולא מודפס, בדומה להעתה ג'אווהסקריפט. היכן שיש הערת HTML – זה המקום שבו כתבים את האלמנטים. תגית הסקריפט תכיל קישור לקובץ הג'אווהסקריפט, ולפיכך היא חייבת להיות מתחת ל-HTML.

אלמנטים של HTML מתורגם לאובייקטים של ג'אווהסקריפט

כל אלמנט שיש ב-HTML יכול להיות מתורגם לאובייקט ג'אווהסקריפטי כשר למהדרין. כך קוד ג'אווהסקריפט ח'י בתוך HTML הוא יכול לגשת אל ה-HTML זהה באופן ישיר בלי צורך בטעינה. ה-HTML מתורגם לסוג של ישות שנקראת DOM, ראשית התיבות של Document Object Model. הישות הזאת נמצאת בזיכרון זמיןנה דרך קוד הג'אווהסקריפט. נשמע אבסטרקט? מדי? הנה נראה. אחרי שמציבים את הג'אווהסקריפט בתוך ה-HTML, יוצרים בתוכו אובייקט `h1`, שהוא בסגנון זהה:

```
<body>
  <!-- HTML elements will be here -->
  <h1 id="myHeader">Headline</h1>
</body>
```

בקובץ של הג'אווהסקריפט מכניסים את הקוד הבא:

```
let header = document.getElementById('myHeader');
header.innerHTML = 'Hello world';
```

אם טוענים את העמוד אפשר לראות שהטקסט שבסכורתת הוא לא "Headline" אלא "Hello world". איך עושים את זה? ראשית משתמשים באובייקט הגלובלי `document`. האובייקט

זהה קיימ בכל ג'אווהסקריפט שנמצא בתוך HTML ומכל את כל ה-DOM. הוא אובייקט כמו כל אובייקט אחר ויש לו מתודות.

אחד המתודות היא **getElementById**, שמקבלת ארגומנט אחד. איזה ארגומנט? את ה-*id*. במקורה שלנו ה-*id* הוא הערך שהושם ב-*attribute* מסוג *id* של האלמנט *h1*. המתודה מחזירה את הייצוג של האלמנט ב-DOM עם כל המתודות שלו. ומעכשיו? חגיגה שלמה. אחד המאפיינים הוא **innerHTML**, שמאפשר לשנות את תוכן האלמנט.

אפשר גם ליצור אלמנטים חדשים לחלוטין ולהכניס אותם למסגר:

```
let p = document.createElement('p');
p.innerHTML = 'I am a paragraph';
document.body.appendChild(p);
```

כאן למשל משתמש במתודה **createElement** כדי ליצור אלמנט מסווג פסקה או תגית **<p></p>**, להכניס אליו תוכן ווז להציגו אותה אל ה-*body* של המסמך באמצעות **appendChild**. עד שימושים במתודה **appendChild**, הייצוג של האלמנט נמצא בזיכרון. עד שהוא לא מחובר ל-DOM אין לו ייצוג ייזורי.

אירועים עם HTML וג'אווהסקריפט

אפשר גם להזין לאירועים שמתרחשים בדף. אירוע יכול להיות קלייק, מעבר של העכבר על אלמנט והמון דברים אחרים. בדוגמה הבאה אתמקד באירוע של קלייק. הנה ניצור ב-HTML כפטור שיש לו פונקציית חישוב:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="myButton" type="button">Click me!</button>
  <script src=".//source.js"></script>
</body>

</html>
```

אם תפתחו את הדף, תראו שנוצר כפטור עם הכיתוב "Click Me!".
בקובץ ה-ג'אווהסקריפט מוגשים את אלמנט הcpfotor כרגע עם id `myButton` ומכניסים אותו למשתנה:

```
let button = document.getElementById('myButton');
```

עכשיו נרצה שבכל פעם שיקלייקו על הcpfotor יראו משהו בקונסולה. את זה עושים בעזרת אירוע DOM. אירוע DOM מופעל על ידי אינטראקציה עם המستخدم שלוחץ על הcpfotor, והטיפול באירוע מתבצע באמצעות הפונקציה `addEventListener` שמאגדים אותה כמתפלת באירוע. מתודת `addEventListener` זמינה בכל אלמנט. היא מקבלת שני ארגומנטים. הראשון הוא סוג האירוע, ובמקרה זה: `click`. השני הוא פונקציה שפועלת בכל פעם שיש קלייק. משתמשים במקרה זה בפונקציית חץ בסיסית שכבר הופיעה בדוגמה:

```
button.addEventListener('click', (event) => {
  console.log('click!');
});
```

הקוד במלואו יהיה:

```
let button = document.getElementById('myButton');
button.addEventListener('click', (event) => {
  console.log('click!');
});
```

אם תרצו את זה תוכלו לראות ש בכל פעם שלוחצים על ה כפתור, בקונסולה רואים "קליק". אבל למה להסתפק רק בקונסולה? אפשר ליצור אלמנט ש כתוב בו "קליק" ולהכניס אותו אל המסמך!

```
let button = document.getElementById('myButton');
button.addEventListener('click', (event) => {
  let myP = document.createElement('p');
  myP.innerHTML = 'Click!';
  document.body.appendChild(myP);
});
```

במקום `console.log` אפשר לשים כל קוד אחר, ובמקרה זהה קוד שאתם כבר מכירים, שייצור אלמנט. ה קליק של המשתמש הוא בעצם ה קלט. ה פלט הוא הרינדור של האלמנט. אפשר ליצור ה קלט משוכל יותר באמצעות אלמנט ה-HTML שנקרא `input`. האלמנט זה יוצר שדה טקסט והוא נראה כך:

```
<input id="myInput" />
```

אפשר לגשת אליו כמו כל אלמנט HTML. תופסם את האלמנט באמצעות `getElementById` ונציגים אל תוכנת ה-`value` שלו:

```
let input = document.getElementById('myInput');
let content = input.value;
```

הצעד הבא יהיה לנקח את מה שיש ב-`input` ולהציב אותו ב-HTML בכל פעם שהמשתמש לוחץ על קליק:

```
let button = document.getElementById('myButton');
let input = document.getElementById('myInput');
button.addEventListener('click', (event) => {
  let myP = document.createElement('p');
  myP.innerHTML = input.value;
  document.body.appendChild(myP);
});
```

סביר שוב על הקוד המלא: ב-HTML יש שני אלמנטים, אלמנט אחד שיש לו ID בשם `myButton` והוא כפתור, ואלמנט נוסף של שדה טקסט שה-ID שלו הוא `myInput`. שניהם נכניסים אחר כבוד למשתנים המתאימים כדי שיתאפשר להתייחס אליהם. בשלב הבא מצמידים אירוע `click` לכפתור. האירוע עובד בכל פעם שיש לחיצה. מדובר בפונקציה טהורה, כלומר צוזה שהיא ג'אווהסקרייפט בלבד, שבמקרה זהה יוצרת אלמנט מסווג פסקה. ב-HTML פסקה היא תגית `<p>`. היא לנקח את מה שהמשתמש הכניס לתוכה שדה הטקסט, מכניסה אותו לתוכנת `innerHTML` של אלמנט הפסקה ומcmdיה את אלמנט הפסקה אל גוף ה-HTML, וכך אפשר לראות אותה.

מה שחשוב להבין הוא שמדובר כאן בג'אווהסקריפט לכל דבר אך בשילוב ה-DOM – כלומר אובייקטים וארוועים שקיימים מהאלמנטים שיש ב-HTML. ברגע שימושים ב-`document` כדי ליצור אלמנט או לקרוא לו, הוא מקבל גם תכונות כמו `innerHTML` ואפשר להציג אליו אירועים מיוחדים כמו "קליק" באמצעות `addEventListener`.

הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווהסקריפט

בכל הדוגמאות עד כה הריאתי איך מציגים את האלמנט שיצרנו ישירות ל-`document.body` שהוא בעצם תגיית ה-`<body>` שיש ב-HTML. זו תגיית ייחודית שמופיעה רק פעם אחת במסמך HTML. אפשר להציג אלמנטים לכל אלמנט אחר! אדגמים באמצעות תגיות `` ו-``. מדובר בתגיות של רשימה ב-HTML, שנראית כך:

```
<ul>
  <li>פריט ראשון</li>
  <li>פריט שני</li>
  <li>פריט שלישי</li>
</ul>
```

הבה ניצור דף אינטרנט בו רשימת קניות שהמשתמש יכול להכנס אליה פריטים. ה-HTML יכול חלון של שדה טקסט, כדי שהמשתמש יוכל להכנס פריטים, כפטור של "הכנס" וcmbobן שילד של הרשימה:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="myButton" type="button">Click me!</button>
  <input id="myInput" />
  <ul id="myList"></ul>
  <script src=".//source.js"></script>
</body>

</html>
```

קוד ה-ג'אווהסקריפט יהיה פשוט. בדיקן כמו בדוגמה הקודמת, מגדרים את שלושת השחקנים במשחק: הכפטור שמאפיע את כל הfonקציה של ההוספה, שדה הטקסט שבו המשתמש מכניס את מה שהוא רוצה להכנס לרשימה והרשימה הריקה.

```
const button = document.getElementById('myButton');
const input = document.getElementById('myInput');
const list = document.getElementById('myList');
```

ועכשו להצמדת האירוע לכפתור. את זה עושים באמצעות המתודה `addEventListener`. מعتبرים שני ארגומנטים. הראשון הוא סוג האירוע (קליק), והשני הוא ארגומנט של הפונקציה שתעבד כשהארגון יפעל. מعتبرים אותה כפונקציית חץ:

```
button.addEventListener('click', (event) => {  
});
```

הקוד שבתוך הפונקציה הוא פשוט. יוצרים אלמנט חדש מסוג או, מוכנסים את ה-`innerHTML` שלו בערך המתאים משדה הtekst – `value`, ואז מضافים אותו אל הרשימה:

```
button.addEventListener('click', (event) => {  
    const myListItem = document.createElement('li');  
    myListItem.innerHTML = input.value;  
    list.appendChild(myListItem);  
});
```

שינוי עיצוב

ה-**DOM** מאפשר לא רק ליצור אלמנטים חדשים אלא גם לעצב אותם באמצעות CSS. כאמור, הספר זהה אינו מלמד CSS או HTML. אבל קל ללמידה CSS, ובגדול אפשר לשלוט בעדרתו על העיצוב ועל המיקום של כל אלמנט HTML. לכל תוכנת CSS יש ערך שלה. למשל, אם רוצים לקבוע את רוחב האלמנט, התוכנה היא `width` והערך הוא (למשל) `10px`, שימושו הוא רוחב של 10 פיקסלים. תוכנת `background-color` קובעת את צבע הרקע, והערך יכול להיות צבע (כמו `red`) או ערך הקסדימלי של צבע כמו `#ff0000`.

על מנת לשנות תוכנת CSS או ליצור תוכנה חדשה משתמש בתוכנת `style`, שקיים בכל אלמנט DOM. לשם הדגמה אצור אלמנט HTML מסווג `div` עם `id` מסוים ב-HTML.

אלמנטים מסווג `div` הם אלמנטים בסיסיים שאין להם עיצוב:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <div id="myDivId"></div>
  <script src=" ./source.js" > </script>

</body>
</html>
```

אם תציגו את הדף זהה באמצעות הדפדפן לא תראו כלום. הנה נכניס קצת צבע לאלמנט זהה באמצעות ג'אווהסקרייפט:

```
const div = document.getElementById('myDivId');
div.style.height = '100px';
div.style.width = '100px';
div.style.backgroundColor = 'red';
```

יצרים תוכנות של גובה, של רוחב ושל צבע רקע על מנת ליצור ריבוע אדום. זה ממש נחמד. אפשר להכניס, לשנות או לקרוא כל תוכנה של `style` באופן חופשי כמובן. שימוש לב שמדובר בהשפעה על תגית `style` של האלמנט ולא בשינוי קובץ CSS שקשרו למסמר. לרוב לא מתקבל לשנות צבעים באופן ישיר בג'אווהסקרייפט, אלא ליצור `class` של CSS שמכיל את התוכנות המבוקשות וaz להויסף את הקלאסים הללו לאלמנטים או להוריד אותם מהם באמצעות ג'אווהסקרייפט.

למשל, בקובץ CSS יהיה סלקטור של `:activated`:

```
.activated {
  background-color: red;
  height: 100px;
  width: 100px;
}
```

ובקוד עצמו:

```
const div = document.getElementById('myDivId');
div.classList.add('activated');
```

במשתנה. אם כל מה שכתבם לעיל נשמע לכם ג'יבריש – זה הזמן ללימוד CSS. כאמור, אני לא מכסה HTML או CSS בספרו זה.

סלקטורים של DOM

סלקטור הוא שם כללי למетодות שモוצאות אלמנטים. עד כה השתמשנו ב-`document.getElementById`, שהוא סלקטור שモץ אלמנטים לפי ה-`id` שלהם. אך יש סלקטורים אחרים שאינם מבוססים על `id`. הסלקטור של `id` הוא ייחודי כי יצאים מנקודת הנחה שתמיד יש אלמנט אחד שיש לו את ה-`id` זהה. במקרים שיש סלקטורים המבוססים על `class` או על תכונה אחרת, יתקבל מערך של תוצאות שאפשר להתייחס אליו כל כל מערך, אלא שבמערך זהה יהיו אלמנטים של DOM.

הבה נדגים באמצעות הסלקטור `getElementsByClassName`. הסלקטור זה מחזיר מערך של כל האלמנטים שיש להם `class` מסוים. קיימם ה-HTML זהה:

```
<div class="rectangle">Rec 1</div>
<div class="rectangle">Rec 2</div>
```

(**שים לב:** לשם הנוחות אני לא מציב כאן את ה-HTML המלא כולל ה-`body` וה קישור לקובץ הג'אווהסקריפט החיצוני.)

על מנת לבחור את כל האלמנטים שיש להם את ה-class זהה ולצבוע אותם באדום (לצורך העניין) עושים משהו כזה:

```
const recs = document.getElementsByClassName('rectangle'); //  
[div.rectangle, div.rectangle]  
for (let el of recs) {  
    el.style.backgroundColor = 'red';  
    el.style.height = '100px';  
    el.style.width = '100px';  
};
```

מקבלים מערך של כל האלמנטים שיש להם class בשם rectangle באמצעות השורה:

```
const recs = document.getElementsByClassName('rectangle');
```

עוברים על המערך זהה בדיק כמו על כל מערך רגיל בג'אווהסקריפט, במקורה זהה באמצעות of, אבל אפשר להשתמש בכל לולהה שהיא. למדנו על לולאת of בפרק על לולאות, אז אתם אמרוים להבין מה קורה פה. עוברים על כל איבר בלולאה ונותנים לו גובה, רוחב וצבע. אם רצים לתת אותם רק לאלמנט הראשון, צריך לעשות משהו כזה:

```
recs[0].style.backgroundColor = 'red';  
recs[0].style.height = '100px';  
recs[0].style.width = '100px';
```

חשוב להבין שאף על פי שמדובר באלמנטי DOM, אנחנו בג'אווהסקריפט. כל מה שלמדנו עד עכשיו – אובייקטים, לולאות, מערכים, סוגים מידע – הכל רלוונטי גם לג'אווהסקריפט בסביבת דף. כשרמיצים ג'אווהסקריפט בסביבת דף – גם ה-DOM וגם הג'אווהסקריפט חיימ בתוך הדף והוא מנהל את עצ ה-DOM ואת המנווע של הג'אווהסקריפט ודואג לחבר בין השניים. למען האמת, מפני שאתם כבר מכירם את היסודות התיאורתיים של השפה, אתם אמרוים לשלוט גם בג'אווהסקריפט בסביבת דף. מדובר באותה שפת ג'אווהסקריפט בדיק.

אפשר להעביר ארגומנט רשיימה של קלואסים עם הפרש של רווח ביןיהם. אפשר גם לבחור אלמנטים לפי שם התגית. למשל, הקוד הזה בוחר את כל התגיות שהן מסוג `div` וצובע אותן באדום. העיקרונו הוא אותו עיקרונו כמו ב-`getElementsByClassName`:

```
const divs = document.getElementsByTagName('div'); // [div, div]
for (el of divs) {
  el.style.backgroundColor = 'red';
  el.style.height = '100px';
  el.style.width = '100px';
};
```

סלקטור נוסף הוא `All`, שבודח את האלמנטים לפי סלקטורים של CSS. כאן נדרש הינה ב-CSS, אבל כל סלקטור מרכיב של CSS יכול להיות שימושי גם בג'אווהסקריפט. למשל נסתכל על ה-HTML הזה:

```
<div class="good_parent">
  <div class="rectangle">Rec 1</div>
  <div class="rectangle">Rec 2</div>
</div>
<div class="bad_parent">
  <div class="rectangle">Rec 3</div>
  <div class="rectangle">Rec 4</div>
</div>
```

אם רוצים לבחור אך ורק את ה-`div` שה-`class` שלהם הוא `rectangle` והם יядים של `good_parent`, אפשר להשתמש בסלקטור של CSS שהוא:

`.good_parent div.rectangle`

אם המילים "סלקטור של CSS" נשמעות לכם כמו סינית, סימן שאתם צריכים לחזור על ה- CSS שלכם. הספר הזה אינו מכסה CSS ו-HTML.

הקוד יראה כך:

```
const divs = document.querySelectorAll('.good_parent div.rectangle');
// [div.rectangle, div.rectangle]
for (el of divs) {
  el.style.backgroundColor = 'red';
  el.style.height = '100px';
  el.style.width = '100px';
}
```

גם פה זה לא מסובך במיוחד. ברגע שבמביבנים שיש סלקטורים שמחזירים מערך של אלמנטי DOM במקום אלמנט DOM אחד, וההערך הזה הוא מערך ג'אווהסקרייפטி לכל דבר ועניין אלא שהוא מכיל אלמנט DOM – נגמר הסיפור.

ל-`querySelectorAll` יש גם גרסה של פונקציה בשם `querySelector` שמחזירה רק את התוצאה הראשונה של הסלקטור ולא מערך שלהם. כך למשל הקוד הזה:

```
const div = document.querySelector('.good_parent div.rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
```

זהה לחלוטין ל:

```
const divs = document.querySelectorAll('.good_parent div.rectangle');
// [div.rectangle, div.rectangle]
const firstDiv = divs[0];
firstDiv.style.backgroundColor = 'red';
firstDiv.style.height = '100px';
firstDiv.style.width = '100px';
```

לשם הנוחות אצרף רשימה של סלקטוריים נפוצים ותפקידם:

תפקיד	שם הסלקטור
בוחר אלמנט אחד ויחיד לפי ה-ID מחזיר אלמנט אחד	document.getElementById
בוחר כמה אלמנטים לפי קלאס אחד או יותר (הklassים מועברים כמחרוזת טקסט עם רווח) מחזיר מערך של אלמנטים	document.getElementsByClassName
בוחר כמה אלמנטים לפי שם האלמנט - סימן לכל האלמנטים ב- DOM מחזיר מערך של אלמנטים	document.getElementsByTagName
בוחר כמה אלמנטים לפי תכונת השם – name מחזיר מערך של אלמנטים	document.getElementsByName
בוחר כמה אלמנטים לפי סלקטור של CSS אפשר להכנס כמה סלקטורים עם פסיק ביןיהם מחזיר מערך של אלמנטים	document.querySelectorAll
בוחר את האלמנט הראשון שמציעת לסלקטור של ה-CSS מחזיר אלמנט אחד	document.querySelector

אירועים נוספים

עד כה למדנו רק על אירוע קליק, אך יש אירועים רבים נוספים הקשורים לאלמנטים של DOM. למשל `hover`, המופעל בכל פעם שהעכבר עובר מעל אלמנט מסוים. למשל, ל-HTML זהה:

```
<div id="rectangle">Rec 1</div>
```

אפשר לקבוע את הצבע ואת הגודל באמצעות שיבורים שכבר למדנו:

```
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
```

אבל אפשר גם לגרום לו לשנות את הצבע באמצעות אירוע, אם העכבר 'עליה' עליו. את זה עושים באמצעות הצמדת אירוע `mouseover` בשם `mouseover`. הטכניקה זהה לטכניקה שכבר למדנו:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('mouseover', (event) => {
    div.style.backgroundColor = 'yellow';
});
```

האירוע `addEventListener` מקבל שני ארגומנטים. הראשון הוא שם האירוע והשני הוא הפונקציה שעובדת כשהאירוע מופעל. משתמשים בפונקציית `צץ`, בדיקן כמו `click`. אם תרצו את הקוד הזה תגלו שצבע הריבוע משתנה לצהוב ברגע שהעכבר עולה על הריבוע האדום. חשוב לציין שעושים את זה בג'אווהסקריפט לשם התרגול וההסביר. במקרה אפקטים כאלה נעשים באמצעות CSS בלבד.

אני רוצה להתעכבר קצת על הארגומנט שפונקציית החץ מקבלת, והוא ה-`event`. ככל-mdנו על פונקציות חץ ועל קולבקים בפרק על הפונקציות, הסבירתי שככל קולבק יכול לקבל כמה ארגומנטים שמי שמאפיין אותם בעברית. במקורה הזה הארגומנט שמקבלים הוא אובייקט האירוע עצמו, שמכיל عشرות פרטיים. למשל, אם הcptוריהם Alt, Shift או Ctrl הינו לחוצים, מה המיקום המדויק של האירוע, באיזה שעה הוא התקיים וכו' וכו'. כך למשל אפשר לארע הקליק רק אם הcptור Alt לחוץ באופן הבא:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('click', (event) => {
  if (event.altKey === true) {
    div.style.backgroundColor = 'yellow';
  }
});
```

במקרה של הקוד הזה, רק לחיצה בשילוב cptור Alt לחוץ תגרום ליריבוע להשתנות לצבע צהוב. אובייקט ה-`event` (אפשר לקרוא לו בכלל שם כMOVEN) הוא אובייקט כמו כל אובייקט. אפשר לעשות עליו `console.log` כדי לתחות על קנקנו ואיפלו מומלץ להביט בו. אפשר לעשות איתו דברים מעניינים מאוד.

אפשר להפעיל כמה אירועים על אותו אובייקט. כאן למשל גורמים ליריבוע להפוך לצהוב כשהעכבר נמצא עליו ולהיות שוב אדם כאשר הוא ממשיך הלאה. האירוע הראשון משתמשים בו הוא `mouseenter`, שמאופעל ברגע שהעכבר נכנס אל האלמנט, והאירוע השני שמשתמשים בו הוא `mouseleave`, שמאופעל ברגע שהעכבר יצא מהאלמנט:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('mouseenter', (event) => {
  div.style.backgroundColor = 'yellow';
});
div.addEventListener('mouseleave', (event) => {
  div.style.backgroundColor = 'red';
});
```

שימוש לב שיש משמעות לAIROUTS. למה לא השתמשתי ב-mouseover? כיון שהוא מופעל בכל פעם שהעכבר נמצא מעל האלמנט, כלומר המון פעמיים — בכל פעם שהעכבר ZZ והतזוזה שלו היא מעל האלמנט. האירוע mouseenter מופעל רק פעם אחת — כמשמעותו נכנס לתוכן האלמנט. זה הכל.

יש המון AIROUTS, מאירועי מקלדת ועד AIROUT עכבר. יש גם AIROUTS של אלמנטים (למשל AIROUT שМОפעל אם מישו מונה את ה-CSS של האלמנט) וAIROUTS הקשורים ל-clipboard. הינה רשימה של כמה AIROUTS עיקריים:

AIROUT	הסביר
click	AIROUT של הקלקה באמצעות העכבר
mouseenter	סמן העכבר נכנס אל תחום האלמנט
mouseleave	סמן העכבר יצא מתחומי האלמנט
mousedown	המשתמש לוחץ על כפתור העכבר בתחום האלמנט
mouseup	המשתמש שחרר את כפתור העכבר בתחום האלמנט
wheel	המשתמש מגלגל את גלגלת העכבר כאשר סמן נמצא על האלמנט
keydown	לחיצה על כפתור במקלדת
keyup	שחרור של כפתור במקלדת
focus	המשתמש מבצע פוקוס על האלמנט (למשל באמצעות הטאב או העכבר)
blur	המשתמש יצא מפוקוס על האלמנט (למשל באמצעות הטאב או הקלקה על אלמנט אחר)

פעוף של AIROUTS

פעוף (באנגלית **bubbling**) הוא שם של תופעה שחווב להכיר באירועים, והוא מתרחשת כאשר יש יותר מכמה אלמנטים. למשל כמו בדוגמה הזו:

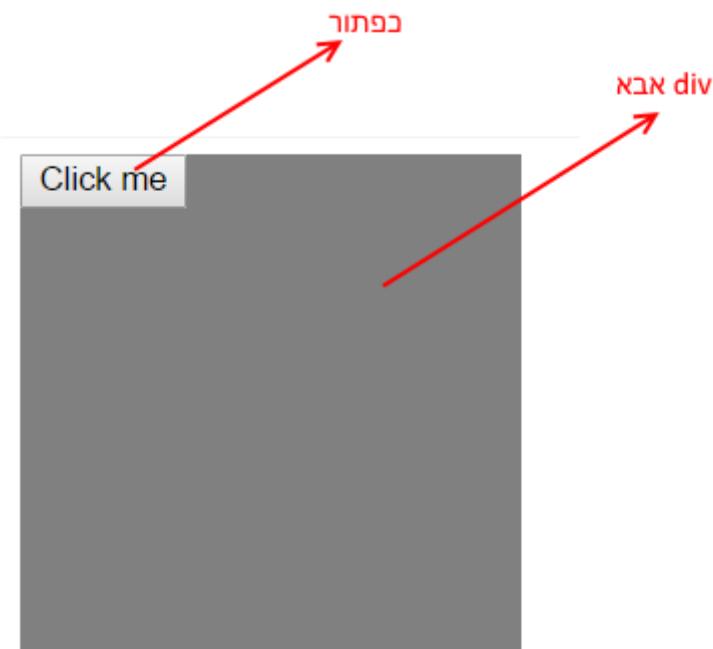
```
<div id="parent">
  <button id="button">Click me</button>
</div>
<div id="result"></div>
```

כאן יש שלושה אלמנטים. אלמנט עם id של parent שבתוכו יש כפתור עם id של button. בתחתית יש div שכתוב בו result.

קוד היג'אווהסקריפט הבא רץ באותו דף. מקבלים את שלושת האלמנטים הבאים של DOM באמצעות getElementById בהתאם ל-`id` של כל אחד מהם, צובעים את ה-`parent` באפור ונוטנים לו מדדים:

```
const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
```

התוצאה צריכה להיות משהו זהה:



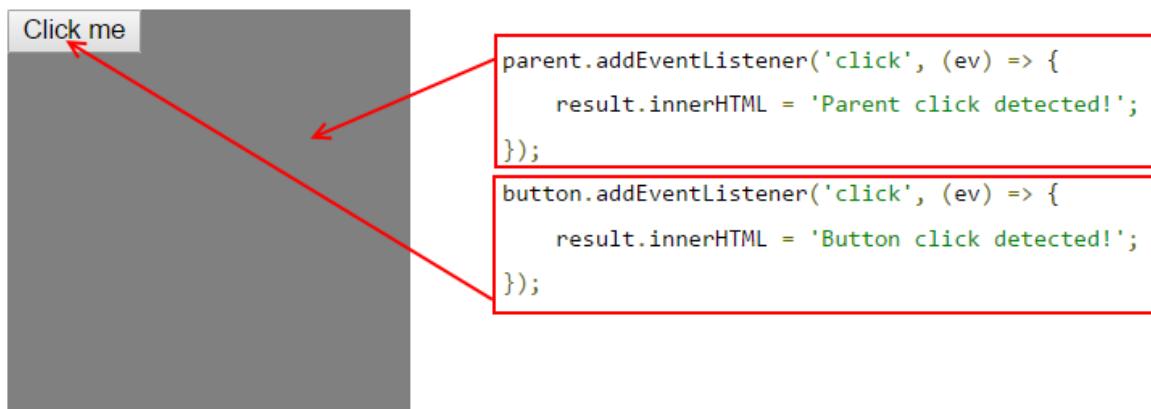
מה קורה אם מצלמים לאלמנט האב אירוע `click`? תזכורת: אלמנט האב הוא `div` שמכיל את הcptor. בדוגמה זו לאלמנט האב יש `div` בשם `parent` (דיברנו על אלמנטים אבות וילדים מוקדם יותר בפרק).

```
parent.addEventListener('click', (event) => {
    result.innerHTML = 'Parent click detected!';
});
```

זה קל – אם לוחצים על האב, רואים שבדף התוצאה כתוב: `Parent click detected`. מה קורה אם לוחצים על cptor? אותו הדבר! למה? כי האירוע מפעע גם לאלמנטים של ההורים. האירוע מתחילה בcptor ומפעע למעלה עד שיש אלמנט שמתפל בו (ובמקרה זהה האב). כלומר, אם אני אלמנט ויש אירוע על האלמנט שמכיל אותו (או על האלמנט שמכיל אותו וכו'), אז האירוע יפעע לפני מעלה עד שפגע באירוע כלשהו (או עד שיגיע ל-`body`). לוחצים על cptor, אין בו אירוע לחיצה. האירוע עולה למעלה. האם באלמנט האב יש אירוע? כן! אז האירוע מופעל וממשיך לעלות למעלה, והוא מפעיל את כל האירועים באלמנטי האב השונים.

זה נחמד ו שימושי, אבל העסוק מתחילה להסתבר כאשר יוצרים אירועים לאלמנט אב ולאלמנט בן:

```
parent.addEventListener('click', (event) => {
    result.innerHTML = 'Parent click detected!';
});
button.addEventListener('click', (event) => {
    result.innerHTML = 'Button click detected!';
});
```



מה קורה כשהילחו על cptor? האירוע של cptor יתרחש, ואכן ב-`div` של התוצאה יודפס `Button click detected`, אבל מיד אחר כך, האירוע יפעע למעלה ויפעל את האירוע של parent, שבתו יופיע מיד ב-`div` של התוצאה `Parent click detected`.
לעולם לא תראו את האינדייקציה של הלחיצה על cptor ב-`div` ה-`result`! זה קצת בעיתי אם אנחנו רוצים לראות בלחיצה על cptor `Button click detected`, ורק כשלוחצים על `div` של האב מקבל `Parent click detected`.



מה יקרה אם לא רוצים שהאירוע יפעע למעלה? באירוע שבו רוצים לעצור את הפעוע ח'יבם להפעיל את:

`event.stopPropagation();`

שזמן באירוע עצמו, לצד שאר תכונות האירוע. קחו למשל הקוד הזה:

```

const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
parent.addEventListener('click', (event) => {
  result.innerHTML = 'Parent click detected!';
});
button.addEventListener('click', (event) => {
  result.innerHTML = 'Button click detected!';
  event.stopPropagation();
});

```

הוא יעשה בדיק את מה שרצו שהוא יעשה. לחיצה על האב תדפיס שלחצתם על האב. לחיצה על הcptor תפעיל את האירוע של cptor, אך האירוע לא יחלחל להלאה והאירוע של האב לא יופעל.

הצמדת אירוע ג'אויסקרייפט לאלמנטים ב-HTML

השיטה של `addEventListener` נחשבת לעדיפה בהרבה, אבל אפשר גם להצמיד אירועי של ג'אויסקרייפט ב-HTML באמצעות תכונת `onclick` לקליקים. למשל משזה:

```

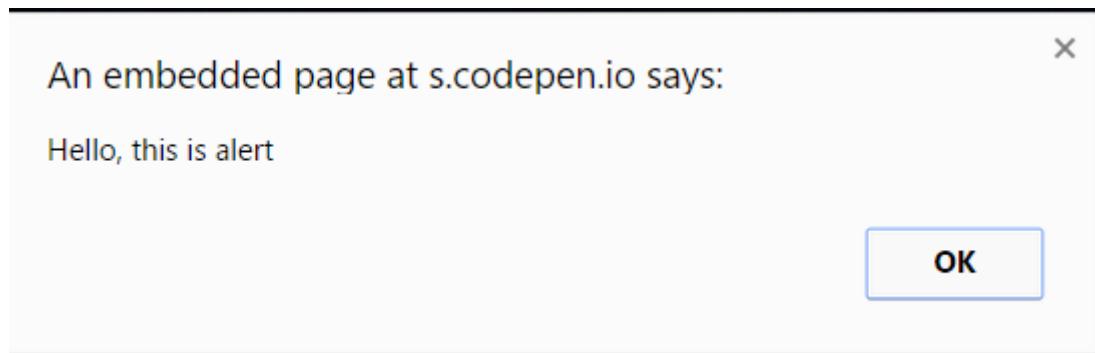
```

צירת `prompt`-`alert`

עוד דרך מישנת לתקשר עם משתמשים היא באמצעות `alert`. מדובר בפונקציה גלובלית זמינים בכל ג'אווהסקריפט שchina בסביבת דף. משתמשים בה כר:

```
alert('Hello, this is alert');
```

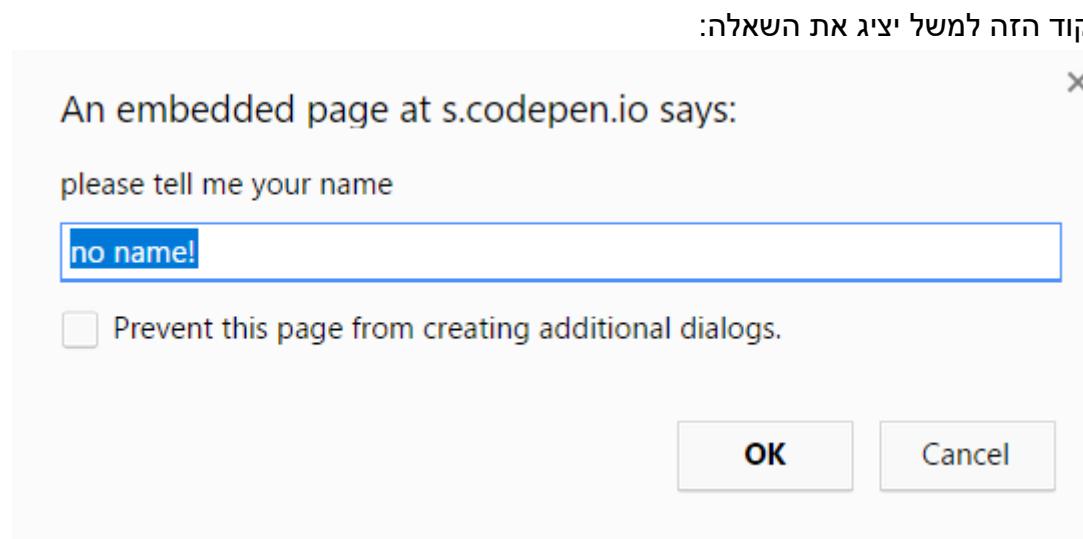
מה שהמשתמש יראה הוא חלון לא גרפי שיקפוץ עם הודעה.



עיצוב חלון תלוי בדף, בעיקר מפני שמדובר בחלון של מערכת הפעלה. כאמור, מדובר בדרך מישנת להציג הודעות למשתמש וכיום ממעטים להשתמש בה. מומלץ מאוד להציג למשתמש הודעות בדרך אחרת, כפי שהראיתי לכם – הקצתה `div` מיוחד והדפסת הודעות עליון.

דרך נוספת לתקשר עם המשתמש היא באמצעות הפונקציה הגלובלית `prompt`, שגם היא זמינים בכל ג'אווהסקריפט שchina בסביבת דף. הפונקציה זו מקבלת שני ארגומנטים. הארגומנט הראשון מכיל את המחרוזת למשתמש. כמשמעותו, מוצגת למשתמש המחרוזת שהוכנסה בארגומנט הראשון עם שדה טקסט שבו הוא יכול להכניס מחרוזת טקסט משלו, ואוותה פונקציית ה-`prompt` מחזירה. הארגומנט השני הוא ערך ברירת המחדל שモצג למשתמש.

```
const result = prompt('please tell me your name', 'no name!');  
alert(result);
```



אם המשתמש יכנס שם, מיד אחר כך יוקפץ alert עם השם שהוא הכנס. גם כאן מדובר בדרך מישנת מאוד ועדייף לא להשתמש בה. אם רוצים לתקשר עם המשתמש, עושים זאת ב-[זעקו](#), כפי שראינו בדוגמאות הקודמות.

תרגיל:

צרו אלמנט מסוג `span` עם תוכן של "אני יודע קוד" והכניסו אותו ל-HTML.

פתרון והסבר:

יצרים HTML בסיסי עם קישור לקובץ ג'אווהסקריפט, שייהי בתחתית קובץ ה-HTML.

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <!-- HTML elements will be here -->
  <script src=".source.js"></script>
</body>

</html>
```

בקובץ ה-HTML מכניסים את הקוד הבא:

```
let myObj = document.createElement('span');
myObj.innerHTML = 'אני יודע קוד';
document.body.appendChild(myObj);
```

הקוד עצמו פשוט למדי. יוצרים אלמנט באמצעות `createElement` שנמצאת באובייקט `document`. האובייקט זמין בכל קובץ ג'אווהסקריפט שנמצא ב-HTML. את התוצאה של המתודה מכניסים למשתנה `myObj`. מעכשו יש אלמנט DOM שיש לו מתודות מסוימות, וכל אחת מהן משפיעה עליו. אחת החשובות שבהן היא `innerHTML`, שמכניסה את התוכן לאלמנט שנוצר. כל מה שנוטר לעשות אחרי שיוצרים את האלמנט ואת התוכן שלו הוא להכניס את האלמנט ל-HTML באמצעות `appendChild`.

האלמנט

את HTML באמצעות `appendChild`.

תרגיל:

צרו דף אינטרנט שבו שדה טקסט וכפטור. אם המשתמש מכניס מספר לשדה הטקסט ולוחץ על הכפטור, הדף מציג רשימה ממוקפרת באורך של המספר. למשל, אם המשתמש המכניס 3,

תתקבל הרשימה:

- 1 •
- 2 •
- 3 •

פתרונות:

קובץ HTML:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="myButton" type="button">Click me!</button>
  <input id="myNumber" />
  <ul id="myList"></ul>
  <script src=".//source.js"></script>
</body>

</html>
```

בקוד עצמו:

```
const button = document.getElementById('myButton');
const number = document.getElementById('myNumber');
const list = document.getElementById('myList');
button.addEventListener('click', (event) => {
  const numberValue = number.value;
  for (let i = 0; i < numberValue; i++) {
    const myListItem = document.createElement('li');
    myListItem.innerHTML = i + 1;
    list.appendChild(myListItem);
  }
});
```

הסבר:

ב-HTML יוצרים שלושה אלמנטים: שדה טקסט שבו המשתמש יוכל להכניס מספר, רשימה ריקה וכפטור להפעלת הפונקציה שתיקח את המספר ותכניסו פריטים לרשימה. ב-HTML יהיה קישור לקובץ ג'אווהסקריפט שבו יהיה הקוד. אלו שלושת השחקנים.

בקוד הג'אווהסקריפט מגדירים את שלושת השחקנים באמצעות `document.getElementById`.

```
const button = document.getElementById('myButton');
const number = document.getElementById('myNumber');
const list = document.getElementById('myList');
```

מצמידים אירען קליק שמקבל שני ארגומנטים באמצעות `addEventListener`. הראשון הוא שם האירוע והשני הוא מה שקרה כשהוא מופעל. שימוש לב שהשתמשו כאן בפונקציית `cz`:

```
button.addEventListener('click', (event) => {
});
```

בפונקציה עצמה מגדירים את המספר שמקבלים מהמשתמש:

```
const numberValue = number.value;
```

ברגע שיש מספר, מפעילים לולאת `for` שתרוץ לפיו:

```
for (let i = 0; i < numberValue; i++) {
```

בתוך הלולאה יוצרים אלמנט `li` (ו-`i` הוא אלמנט של איבט ברשימה) ומכניסים אותו לרשימה. מאלסימ את תוכנו באמצעות `i + 1`. זה הכל.

תרגיל:

צרו כפטור ואלמנט מסווג div. לחיצה על הכפטור תהפוך את האלמנט לצהוב, ברוחב של 100 פיקסלים וגובהה של 100 פיקסלים.

פתרון והסביר:**קוד ה-HTML:**

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="myButton">Click to change color</button>
  <div id="myDiv"></div>
  <script src=".//source.js"></script>
</body>

</html>
```

קוד ה-ג'אווהסקריפט:

```
const button = document.getElementById('myButton');
const div = document.getElementById('myDiv');
button.addEventListener('click', (event) => {
  div.style.height = '100px';
  div.style.width = '100px';
  div.style.backgroundColor = 'yellow';
});
```

בקוד ה-HTML יוצרים שני אלמנטים. אחד הוא כפטור והשני הוא div. מקפידים לקשר בין ה-HTML לג'אווהסקריפט.

בג'אווהסקריפט מגדירים את שני השחקנים הראשיים – הכפטור וה-div. עושים את זה באמצעות `getElementById`. לאחר מכן מזמנים אל הכפטור אירוע מסווג קלייק, שמוספעל בכל פעם שלוחצים על הכפטור. באירוע עצמו משתמשים ב-`style`, שזמן לאלמנטים של DOM על מנת לקבוע את הגובה, את הרוחב ואת הצבע.

תרגיל:

במה שיר לתרגיל הקודם, כתבו קוד שיגרום לכך שלחיצה על הכפתור תקבע את האלמנט בצהוב, לחיצה נוספת יקבעו אותו באדום, לחיצה נוספת שוב בצהוב וכך הלאה.

פתרון והסביר:

קוד ה-HTML:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <button id="myButton">Click to change color</button>
  <div id="myDiv"></div>
  <script src=".//source.js"></script>
</body>

</html>
```

קוד ה-ג'אווהסקריפט:

```
const button = document.getElementById('myButton');
const div = document.getElementById('myDiv');
let activated = true;
div.style.height = '100px';
div.style.width = '100px';
div.style.backgroundColor = 'yellow';

button.addEventListener('click', (event) => {
  if (activated === true) {
    div.style.backgroundColor = 'yellow';
    activated = false;
  } else {
    div.style.backgroundColor = 'red';
    activated = true;
  }
});
```

קוד ה-HTML אינו שונה מהקוד בתרגילים הקודמים. שני השחקנים העיקריים הם div ו-button. בקוד הג'אווהסקריפט מגדירים את שני השחקנים כאלמנטים של DOM באמצעות getElementById. מגדירים משתנה בוליאני בשם activated בשם activated כ-true. בנוסף על כך מתחילה את הריבוע הצהוב. קובעים את הרוחב, את הגובה ואת הצבע התחלתיים. מצמידים אירוע קлик לכפתור באמצעות addEventListener, שכבר הכרנו בתרגילים קודמים. הפונקציה ש-addEventListener מפעילה בכל קлик מרכיבת מעט. ראשית בודקים אם true הוא activated. אם כן, צובעים את האלמנט בצהוב ומשנים את ה-activated ל-false. אם false הוא activated, הוא נקבע באדום ומשנים את ה-activated ל-true.

פרק 14

דיבואיגניט



דיבאגינג

בכל kali מפתחים שהוא – בין שמדובר בכרום ובין שבפיירפוקס או באdag' – יש אפשרות להפעיל kali שנקרה "דיבאגר". kali מפתחים הוא רכיב תוכנה שנמצא בכל דפדפן ללא צורך בהורדה או בהפעלה. מדובר בשם kali לקל שמשיע למפתחים לבדוק את הקוד שלהם. בדפדפן הכל kali זהה עוזר לפטור תקלות בג'אווהסקריפט ולהבין מה קורה בנבכי הקוד. עד כה השתמשנו בעיקר ב-console.log על מנת להדפיס דברים בקונסולה. kali זהה חוסך את התהיליך של הדפסת פלט בקונסולה ומאפשר לראות בדיקות בג'אווהסקריפט.

כדי לתרגל שימוש בדיבאגר, נניח שבפרויקט שלכם יש קובץ HTML סטנדרטי וקובץ ג'אווהסקריפט שמקשור אליו, בדיקן כמו שהסבירתי בפרק על התקנת סביבת העבודה. קובץ ה-HTML נראה כך:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
</head>

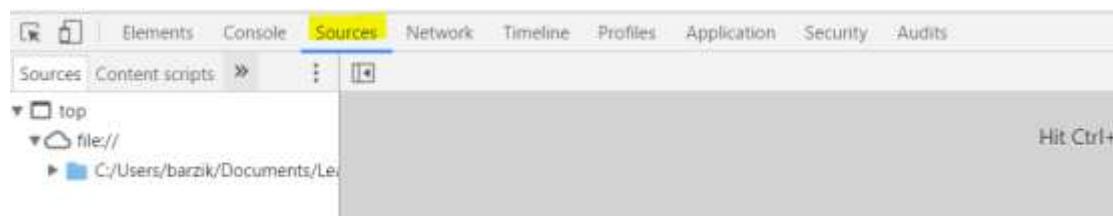
<body>
  <script src=".source.js"></script>
</body>

</html>
```

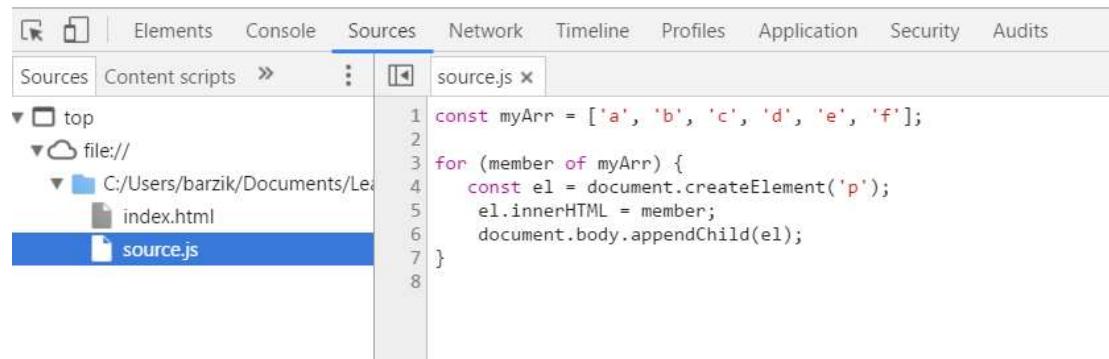
אפשר לראות שיש כאן קישור ל-source.js, שנמצא באותה תיקייה של קובץ ה-HTML. התוכן ב-source יהיה משהו זהה:

```
const myArr = ['a', 'b', 'c', 'd', 'e', 'f'];
for (member of myArr) {
  const el = document.createElement('p');
  el.innerHTML = member;
  document.body.appendChild(el);
}
```

כדי להריץ את הפרויקט פעם אחת בדפדפן על מנת לוודא תקינות. אם הכל תקין, הבה נפתח את הדיבאגר. בכרום, לחצו על F12 אם אתם עובדים בחלונות או על Cmd + Shift + ? אם אתם במק. עכשו מצאו את לשונית sources:



מצד שמאל אפשר לראות את כל עץ הפרויקט. פותחים אותו ומחפשים את קובץ `הג'אווהסקריפט source.js`. ברגע שלוחצים עליו אפשר לראות את כל הקוד מצד ימין.

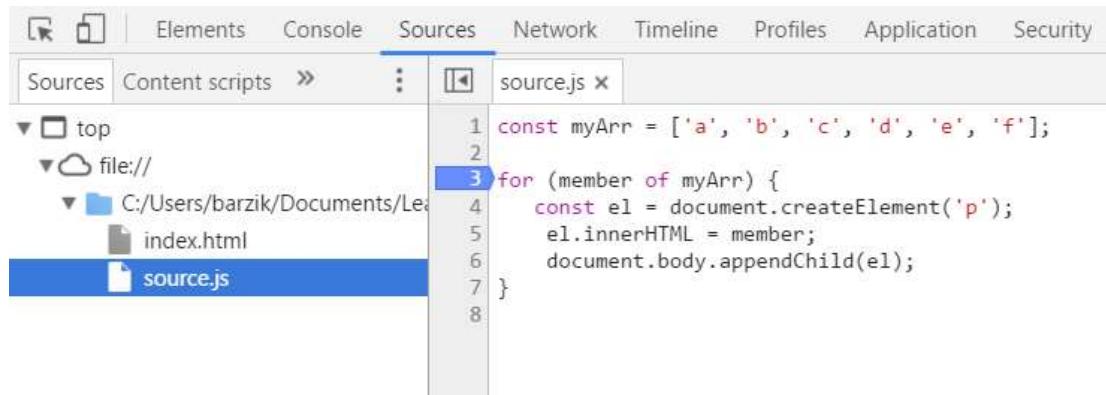


The screenshot shows the Chrome DevTools Sources tab. The left sidebar lists the file structure: 'top' (with 'file://'), 'C:/Users/barzik/Documents/Lead2Learn', 'index.html', and 'source.js'. 'source.js' is selected and highlighted with a blue bar at the bottom. The right pane displays the code content of 'source.js':

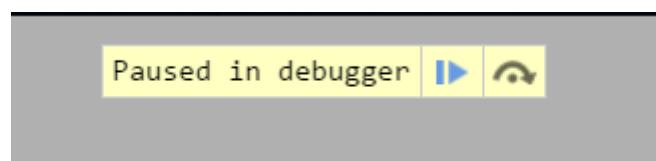
```
1 const myArr = ['a', 'b', 'c', 'd', 'e', 'f'];
2
3 for (member of myArr) {
4     const el = document.createElement('p');
5     el.innerHTML = member;
6     document.body.appendChild(el);
7 }
8
```

במקרה זהה העץ מצומצם מאוד כיון שהוא כולל תקיה אחת בלבד. אם העץ מורכב, אפשר ללחוץ על `Ctrl + p` (או `Cmd + p` במק) ולהקליד את שם הקובץ.

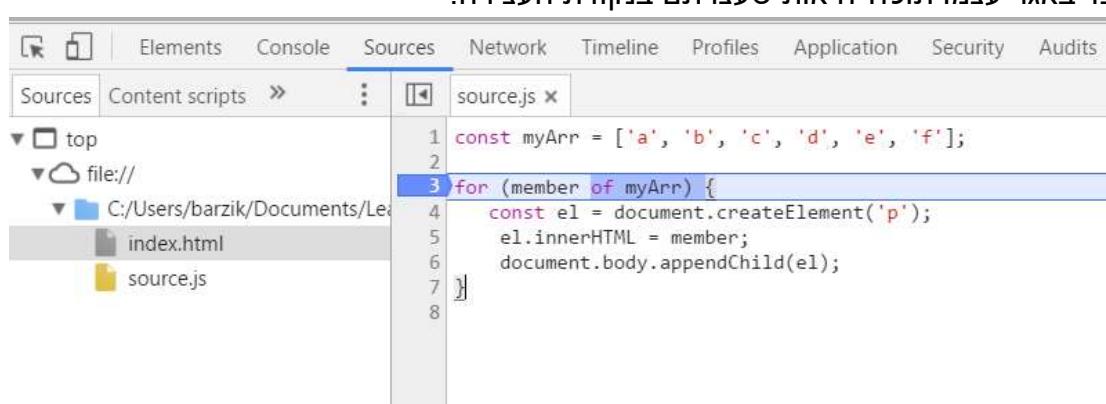
הדייבאגר עובד באופן עקרוני עם נקודות עצירה (באנגלית breakpoints). אפשר לבחור נקודה (או כמה נקודות) בסקריפט שבה הוא יעצור ויאפשר לבדוק את מה שקרה. הנה נראה! נלחץ על השורה השלישי, ממש היכן שנמצאת לולאת `for`. מיד יופיע סימן כחול המאשר שנקודות העצירה קיימות:



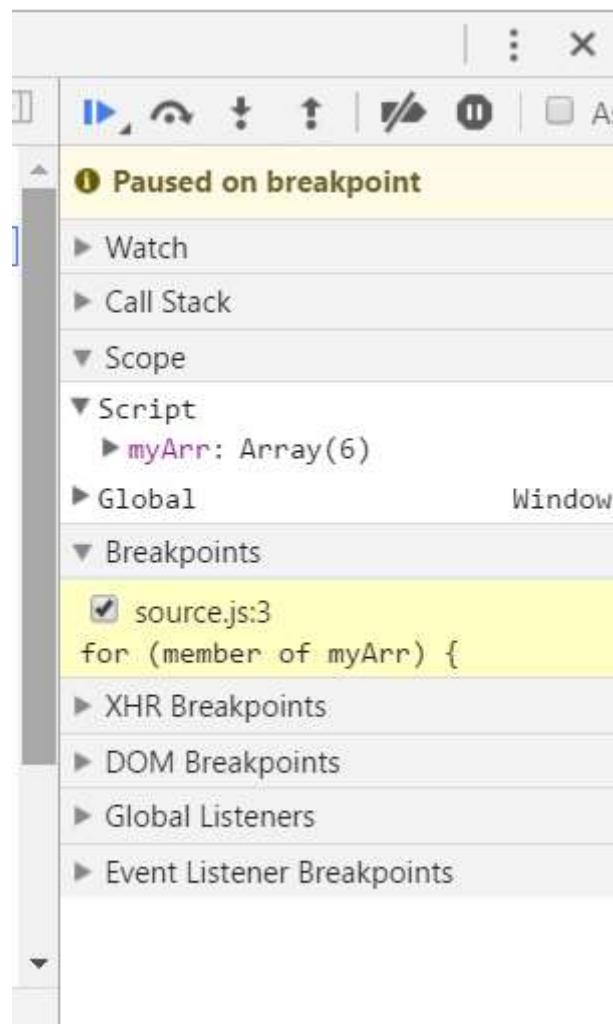
עכשו חיבים להפעיל את הסקריפט מחדש. את זה עושים על ידי טעינה של הדף שוב. שימו לב שבהרבה מקרים לא צריך להפעיל את הסקריפט מחדש. אם מציבים את נקודות העצירה באירוע שMOVED בלחיצה על קליק, אין צורך בהפעלה מחדש. כאן מדובר במקרה נדיר כיון שהסקריפט הזה כבר רץ ולא ירוץ אם לא נתען את הדף מחדש. טעינה מחדש של הדף לא תציג שוב את התוכן המקורי, אלא יופיע לפתע דף אפור שבו אינדיקציה שהדייבאגר מופעל:



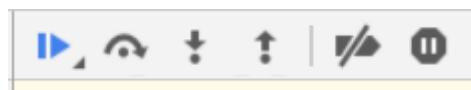
בדיבאגר עצמו תוכלו לראות שערכתם בנקודות העצירה:



מצד ימין אפשר לראות פרטים על המצב הנוכחי:



ב-`scope` למשל אפשר לראות את כל המשתנים הקיימים באותו `scope`, ובמקרה זה רק `myArr`. אפשר גם לראות את כל האלמנטים הגלובליים (כמו `document` שלמדו עליי בפרק על HTML וג'אווהסקריפט) ואובייקטים מובנים אחרים שעוד נלמד עליהם. אם עוברים לקונסולה, אפשר להקליד פקודות שיעבדו בנקודת הזמן הזה של הסקריפט. מעל דיווח המצב יש שורת פקדים שבהם אפשר להפעיל את הדיבאגר:



בלחץ הראשון אפשר להמשיך להריצ' את הסקריפט עד נקודת העצירה הבאה, אם יש כזו. אם אין כזו, הסקריפט ירוץ עד הסיום. בלחץ השני אפשר לעשות `over step` – כלומר להריץ את הסקריפט לשורה הבאה. הלחץ השלישי והרביעי, `stepIn`, `stepOut`, שמורים לכינסה אל פונקציות ולא נדונם בשלב זה. אפשר גם למחוק את כל נקודות העצירה.

הבה נלחץ על `over step`. אפשר גם ללחוץ על F10. תוכלו לראות שהתקדמתם צעד אחד מעבר לנקודת העצירה ורגע אתם בתוך הלולאה:

עכשו יש בסkop שמי חלקים – גם זה של הבלוק, כלומר של ה-`of`. אפשר לראות שיש את ה-`el`, אבל הוא עדין לא מוגדר. הוא יהיה מוגדר רק בשורה הבאה. בואו נתקדם אליה **באמצעות לחיצה שנייה על הפקד step over או על F10**

כעת אפשר לראות שה-`el` הוא אלמנט מסווג. אם לוחצים על החץ הקטן לצד ה-`el` אפשר לראות את האובייקט של ה-DOM במלוא תפארתו, לצד כל התכונות המיעילות של ה-DOM והתכונות שבאות ייחד עם הגדרתו כאובייקט, זה המונ'

בגדייל, אין פה יותר מדי, אך מדובר באחד הכלים החשובים ביותר למתכנתים וכדי מואוד ללמידה להשתמש בו – אז להשתמש בו – כבר בהתחלה. היכולת המרהיבה של הדיבאגר לסייע במציאת תקלות היא חשובה מאוד, והוא גם נותן תמונה מלאה על מה שתרחש בקוד – כולל המשתנים המופיעים בסkop או בקלווזרים השונים (אם שכחتم מה זה, חזרו לפרק על הפונקציות).

שימוש לב: מطبع הדברים עיצוב של דפדף משתנה כל הזמן ויתכן שצלומי המסך המופיעים פה לא מעודכנים מספיק. אם כן, לא צריך להיחז – העיקרון הוא אותו עיקרון גם אם לשונית קוראים בשם אחר או אם האיקון נראה קצת אחרת.

פרק 15

אובייקטים גלובליים ואובייקטים מוגנים



אובייקטים גלובליים ואובייקטים מובנים

הראיתי בכמה הzdמניות שקיימים אובייקטים שיש גישה אליהם. הדוגמה הטובה ביותר היא אובייקט `document` ה-`document` היא לא מילה שמורה כמו `if` או `function`, יש גישה אל פונקציית `log` שלו. מדובר באובייקטים גלובליים הקיימים לג'אווהסקריפט. ב"אובייקטים גלובליים" אני מתייחס לאובייקטים שמוגדרים בرمה הגלובלית ומשם מוחלחים לכל סקופ:

```
console.log('I am global');
() => {
  console.log('I am inside my scope');
  const myArray = [1, 2, 3, 4];
  for (member of myArray) {
    console.log(member);
  }
}();
```

בדוגמה הקוד הזה, למשל, `console` הוא אובייקט שזמן מיד בرمה הגלובלית, בסקופ של הפונקציה האנונימית המריצה את עצמה (שלמדנו עליה בפרק על הפונקציות) וגם בסקופ של `for`. אם דורסים את `console` בرمה הגלובלית ויוצרים אותו מחדש כאובייקט עם פונקציית `log` ריקה, ה-`log` לא יעבד:

```
console.log('I am global'); // Only this will work

() => {
  let console = {
    log: () => { };
  }
  console.log('I am inside my scope');
  const myArray = [1, 2, 3, 4];
  for (member of myArray) {
    console.log(member);
  }
}();
```

בדוגמה שלעיל, למשל, דורסים בתוך הסקופ של הפונקציה האנונימית את `console`. כיוון שלא מדובר במילה שמורה אלא באובייקט גלובלי, אפשר לעשות את זה בקלות. זה אומר שככל ש我们将 `console` בסקופ של הפונקציה האנונימית וכל ה-`log` `console` שהוצבו באותו סקופ במאיצעות `closure` (שגם עליינו למדנו בפרק על הפונקציות), לא יעבדו כיוון שפונקציית `log` תהיה ריקה ב-`console`. כמובן, הטעט שנעביר ב-`log` פשוט לא יציג.

כל הקדמה הארוכה זו נועדה להסביר שיש הבדל משמעותי בין מילה שמורה, שהיא עצם עצמותיה של ג'אווהסקריפט, לבין אובייקטים גלובליים שהם מכובדים וחוובים, אך עדין רק אובייקטים שמציאותם כללית השפה. יש אובייקטים גלובליים שזריםים אך ורק לג'אווהסקריפט שעבוד בסביבת HTML, כמו `document` שהכרנו קודם, `URL`, `Screen`, `Window` וכו', ויש

כאלו שזריםים אך ורק לג'אווהסקריפט שעבוד בסביבת שרת, כמו `process` למשל. אובייקטים גלובליים רבים זמינים לכל ג'אווהסקריפט שהוא, כמו `console`. כל האובייקטים האלה נקראים `Web APIs`.

כל אובייקט גלובלי יש תכונות או מתודות. על מנת לבחון אותן, אפשר לחפש אותן בתיעוד ברשות או פשוט להסתכל בקונסולה. בדוגמה הבאה פותחים את כל המפתחים בכרום, עוברים לקונסולה, מקלידים `console` ולחיצים על `Enter`. בבת אחת רואים את כל המתודות שיש בקונסולה.

```
> console
< ▼Object {debug: function, error: function, info: function, log: function, warn: function...} ⓘ
  ► assert: function assert()
  ► clear: function clear()
  ► count: function count()
  ► debug: function debug()
  ► dir: function dir()
  ► dirxml: function dirxml()
  ► error: function error()
  ► group: function group()
  ► groupCollapsed: function groupCollapsed()
  ► groupEnd: function groupEnd()
  ► info: function info()
  ► log: function log()
  ► markTimeline: function markTimeline()
  ► memory: (...)
  ► profile: function profile()
  ► profileEnd: function profileEnd()
  ► table: function table()
  ► time: function time()
  ► timeEnd: function timeEnd()
  ► timestamp: function timestamp()
```

אפשר לראות שיש שם `error`, למשל. ואכן, אם כותבים בקוד:

```
console.error('This is error!');
```

רואים שהקונסולה שולחת את הטקסט על רקע אדום כשגיאה. אם משתמשים ב-`console.info` רואים שמתקבלות התרעות בגוון אחר.

כאמור, ה-`API` מאפשר לקבל המונן מידע מהתקנים השונים. אחד האובייקטים השימושיים ביותר הוא אובייקט `window`, שיש לו תכונות ומетодות רבות. למשל `window.location` רץ. כך אפשר להעביר את שמחזיר את ה-`URL`, כלומר הכתובת, של הדף שבו הסקריפט רץ. כמו כן ניתן להעתיר את המשתמש למקום אחר. הקוד הזה, לדוגמה, מעביר את המשתמש לאתר גוגל.

```
window.location = 'https://google.co.il';
```

יש המונן אובייקטים גלובליים כאלה, וצריך רק לזכור מהם לא מגיעים משום מקום אלא מזרקיהם לסקופ הגלובלי על ידי ג'אווהסקריפט עצמה. מי שרצה לראות את רשימת האובייקטים המלאה של `Web APIs` מוזמן לגשת לאתר המקיים והטוב ביותר של תיעוד ג'אווהסקריפט: Mozilla Developer Network (MDN), שם יש פירוט מكيف יותר על כל האובייקטים. עם כל שינוי והתקדמות של הדדרפנרים נוספים אובייקטים.

שימוש לב: ניתן לגשת לאובייקט הגלובלי `window` גם באמצעות `globalThis`. האובייקט הזה הוא כינוי לאובייקט הגלובלי המשתנה מסביבה לסייעתה. במקרה של סביבת דפדפן, הוא מחליף את `window`. כך למשל, ניתן לכתוב:

```
globalThis.location = 'https://google.co.il';
```

למשל, בשנת 2015 נוספו אובייקטים גלובליים שמאפשרים תקשורת באמצעות ג'אווהסקרייפט עם מצלמת הרשת, וכך כל קוד ג'אווהסקרייפט יכול לגשת אל מצלמת הרשת, לאחר בקשת רשות מהמשתמש, ולהשתמש בפיד המגיע ממנה למטרות שונות. הרשימה, כאמור, נמצאת באתר MDN:

<https://developer.mozilla.org/docs/Web/API>

בדומה לאובייקטים גלובליים, יש בג'אווהסקרייפט גם אובייקטים מובנים. ההבדל הוא שכן האובייקטים לא מזורקים מהדף או מסביבה העבודה אלא מוגדרים בתכניות של השפה עצמה, כמו למשל `document` (המחלקה שנותה את כל חלקי המשפה). נשמע אולי שמדובר בהבדל דק, אך הוא ממשוני מאד.

אובייקט `document` מגיע מהדף. הוא לא חלק رسمي מהשפה אלא סוג של API שהדף מימוש. אובייקט מוגדר מסוג `Math`, למשל, הוא אובייקט ש מגיע מהשפה עצמה ומופיע בהגדירות שלה. בעוד אובייקט גלובלי כמו `console.error` יכול להיות מישם בדף כרום בצורה אחת, בפיירפוקס בצורה אחרת ובסביבת שרת בצורה אחרת, אובייקט מוגדר `PI` שמחזיר את ערך הפאי מתנהג באופן זהה לחלווטין בלי שום קשר לדף או אם ג'אווהסקרייפט ריצה בסביבת שרת.

חלק מהאובייקטים הם אובייקטים של ממש וחלקים מモומשיים כפונקציה (שגם היא אובייקט). גם את האובייקטים המובנים אפשר לדרוז. כאן לדוגמה את האובייקט `Math` וגורמים ל-`PI` להציג ערך של 5:

```
console.log(Math.PI); // 3.141592653589793
(() => {
  let Math = {
    PI: 5,
  }
  console.log(Math.PI); // 5
})();
```

במה שקדם נעבר על כמה אובייקטים מובנים חשובים.

parselnt

הfonקציה הגלובלית `parselnt` ממירת מחרוזת טקסט למספר. הfonקציה מקבלת שני ארגומנטים. הראשון הוא מחרוזת הטקסט והשני הוא הבסיס. המלצת הרשמית היא להשתמש בשני הארגומנטים, אבל בפועל צריך רק ארגומנט אחד. כך זה עובד:

```
const convertedNumber = parseInt('10230');
console.log(convertedNumber);
```

אפשר לראות שהารגומנט הראשון הוא מחרוזת טקסט, אבל מיד אחרי שמעבירים אותו ב-`parselnt` רואים בקונסולה ש-`convertedNumber` הוא מספר. הfonקציה תעבור גם אם יש המון רוחים:

```
const convertedNumber = parseInt('10230');
console.log(convertedNumber);
```

אבל אם היא מקבלת משהו שהוא לא יכול להתמודד איתו, היא מחרירה `NaN` (כאמור, הקיצור ל-`Not a Number`, לא מספר):

```
const convertedNumber = parseInt('Ahla Mispar 10230');
console.log(convertedNumber); // NaN
```

ראוי לציין שבמקרה ההופיע פונקציית `Doiak` עובדת. אם היא נתקלת במספר שיש אחרי טקסט, היא תונתת את המספר:

```
const convertedNumber = parseInt('10230 ze yofi shel mispar');
console.log(convertedNumber); // 10230
```

eval

eval היא פונקציה גלובלית שמקבלת ארגומנט אחד, שאמור להיות מחרוזת טקסט. eval לוקחת את הטקסט זהה ומРИיצה אותו כailו מדובר בג'אוوهסקרייפט. כמובן היא מעריצה אותו (evaluation)ailו הוא ג'אוوهסקרייפט ומחזירה את התוצאה.

בדוגמה זו למשל לוקחים את מחרוזת הטקסט `1 + 4` וMRIיצים אותהailו היא ג'אוوهסקרייפט. התוצאה של זה דבר היא `5`, זהה בDIוק מה שמתקיים:

```
let result = eval('1 + 4');
console.log(result); // 5
```

אפשר לKחת את זה רחוק יותר ולהכניס לתוך מחרוזת הטקסט הגדרות של משתנים ממש:

```
let a;
let result = eval('a = 1 + 4');
console.log(a); // 5
```

פה למשל מציבים במחרוזת הטקסט מספר לתוך משתנה `a` שהוגדר בסקוופ הגלובל. כיוון שפונקציית eval מMRIיצה את מחרוזת הטקסט כמו ג'אוوهסקרייפט, משתנה `a` מקבל ערך. כל מה שמקף ב-`eval`, לא משנה מה אורכו, ירוץ כמו ג'אוوهסקרייפט. כתיבה של מהו זהה:

```
let a;
let result = eval('a = "Hello";');
console.log(a); // Hello
```

דומה לכתיבה של מהו זהה:

```
let a;
a = "Hello";
console.log(a); // Hello
```

אחד המשפטים היכי נפוצים בקרב מוכנים הוא **eval is evil**. באופן עקרוני צריך להשתמש ב-`eval` רק אם אתם יודעים באמת מה אתם עושים. הסיבה היא ענייני אבטחת מידע – בדרך כלל מחרוזת הטקסט מגיעה בדרך זו או אחרת ממשתמש, וממן אפשרות למשתמש להריץ מה שהוא רוצה בקוד הוא פותח אDIR לביעות אבטחה ולביעות אחרות. לפיכך למדנו כאן את `eval`, אבל רק לצורך ההסביר מודע מומלץ לא להשתמש בה לעולם...

Math

Math הוא אובייקט גלובל שמאפשר לבצע פעולות הקשורות ל... מתמטיקה. אל תיפלו מהיכיא פה. יש לאובייקט לא מעט מתודות ותכונות, שאחת מהן (זאת) הכרנו בתחילת הפרק. רוב המוכנים פוגשים את האובייקט הזה כאשר הם רוצים ליצור מספר רנדומלי, למשל. כמשמעותם למשתמש שם משתמש חדש או כאשר מדים משתמש, משחקים, אнимציות וכו'.

באמצעות `math.random` יוצרים מספר רנדומלי מ-0 (כולל 0) עד 1 (לא כולל 1). המספר הוא שבר עם 16 ספרות לאחר הנקודה. אתם מוזמנים להריץ את הקוד הזה ולבזוק:

```
let result = Math.random();
console.log(result);
```

האם המספר הזה רנדומלי? לא ממש, כיוון שהוא מבוסס על הזמן ועל האלגוריתם שלוקח את חלקי הזמן ויוצר מספר רנדומלי. אבל התהילה הזה רנדומלי מספיק כדי שיתקבל מספר אקראי. נשאלת השאלה, איך לוקחים את המספר הזה והופכים אותו למספר הרנדומלי שרצוים לנו. לדוגמה – למספר מ-1 עד 10. האמת היא שזה די קל – רק צריך להכפיל את המספר שיצא ב-10 ו אז לעגל את התוצאה. למשל, אם המספר הרנדומלי שיצא הוא 0.9181803844895979 מוגלים אותו ב-10 ו מעגלים כלפי מטה. התוצאה תהיה 9. איך מוגלים? באמצעות תוכנה נוספת נספתח של Math שמשמשת ליצירת עיגול, שנקראת `round`. כך:

```
let result = Math.random();
result *= 10;
result = Math.round(result);
console.log(result);
```

בגרסאות ישנות יותר של ג'אווהסקריפט, לפני שהייתה קיימת האופרטור `**` שעליו למדנו באחד הפרקים הקדומים, היו משתמשים במתודת `pow` (קיצור של `power`) לביצוע חזקה. המתודה קיבלה שני ארגומנטים, המספר שעליו החזקה פועלת ומספר החזקה. בדוגמה הבאה מחשבים את המספר 2 בחזקת 3 (התוצאה היא 8):

```
let result = Math.pow(2, 3);
console.log(result); // 8
```

Date

התאריכים בג'אווהסקריפט מחושבים **כמילישניות** (אלפיות השנייה) וモתאים לשעון של המחשב. בעידן המודרני, שבו השעונים מסונכרנים באופן אוטומטי עם הרשת, התאריך מדויק למדוי כך. התאריך הוא מספר המילישניות שעברו מ-1.1.1970 והוא מוסכם על כל מערכות המחשב כ-`epoch time`, הזמן שבו כביכול מערכת הינויקס הראשונה החלה לפעול. כמעט כל שפות התכונות המוכרכות משתמשות במוסכמה זו. חלקן סופרות את השניות בעברית וחלקו את המילישניות. בג'אווהסקריפט סופרים מילישניות.

כשנדרשים לחשב תאריכים בג'אווהסקריפט, צריך לחשב אותם עם `Date`. בנויגוד לאובייקטים הגלובליים שלמדנו עד כה, `Date` הוא פונקציה בניתה שמאחלים עם המילה השמורה `new`. על פונקציה בניתה ועל `new` למדנו בפרק על `this` ועל `new`. לשם תזכורת, `new` מחזיר אובייקט שלהם. הפונקציה הבניתה `Date` מקבלת ארגומנט של הזמן שמעוניינים לחקור. כאשר אין ארגומנט, הזמן הוא ה-`epoch time`. ברגע שמאחלים את `Date`, נוצר אובייקט שמכoon לתאריך האתחול של האובייקט – כמוום היום, ברגע הזה בדיק.

הבה נדגים:

```
const dateObject = new Date();
let epochElapsed = dateObject.getTime();
console.log(epochElapsed);
```

כאן יוצרים `date` ללא ארגומנט, כלומר תאריך האתחול הוא הרגע הזה ממש. משתמשים במתודה `getTime` כדי לקבל את מספר המילישניות שעברו מהתאריך של ה-`time` epoch (1.1.1970) עד היום. נסו ותראו!

אם רוצים לאותחל את אובייקט `date` לתאריך מסוים, אפשר להכניס ארגומנטים של תאריך. הארגומנט הראשון הוא שנה, השני חודש, השלישי יומ, הרביעי שעה, החמישי דקה, השישי שנייה והשביעי והאחרון מילישניה. אפשר גם להכניס רק ארגומנט של שנה וחודש, אז האובייקט יאותחל ביום הראשון של החודש. שנה וחודש הם המינימום הנדרש, אבל אפשר להוסיף עוד ארגומנטים, עד המילישניה. CAN למשל מאתחלים את האובייקט בתאריך המלא `16.09.1977 14:30`, שעה 30, 20 שניות ו-500 מילישניות. אחרי שמאתחלים את האובייקט אפשר להשתמש ב-`getTime` כדי לבדוק כמה זמן עבר מהתאריך של ה-`time` epoch עד התאריך הזה. שזה 245,849,420,500 מילישניות בדוק.

```
const dateObject = new Date(1977, 9, 16, 14, 30, 20, 500);
let myBirthdayElapsed = dateObject.getTime();
console.log(myBirthdayElapsed);
```

זה נראה מעט אבסטרקט. הנה נראה שימוש אמיתי באפשרות הזאת. נניח שמקבלים את גילו של לקוח ורוצים לדעת בן כמה הוא בדוק. ראשית יוצרים אובייקט עם יום הולדתו, ואז יוצרים אובייקט של התאריך הנוכחי ומחסרים ממנו את הזמן של יום הולדתו. זה אפשרי כי הזמן הוא במספר, מילישניות, משנת 1970:

```
const customerDateObject = new Date(1977, 9, 16);
const currentDateObject = new Date();
let gap = currentDateObject - customerDateObject;
console.log(gap);
```

יש כאן את אובייקט התאריך של הלוקה ואת אובייקט התאריך הנוכחי, ופושט מחסרים אחד מהآخر. אם השנה עכשו היא 2018 וולדתם בשנת 1978, חישוב הגיל הוא פשוט: $2018 - 1978 = 40$. CAN במקומות מסוימים סופרים במילישניות, אבל קשה לומר לבן אדם, "היא! אתה בן 1,305,505,055 מילישניות!" – צריך להשתמש בפורמט של בני אדם. פה נכנסת המתמטיקה לשיפור, ונעשה חישוב פשוט שבו מחלקים את הפער במילישניות במספר המילישניות שיש בינה:

```
const customerDateObject = new Date(1977, 9, 16);
const currentDateObject = new Date();
let gap = currentDateObject - customerDateObject;
console.log(gap);
const years = gap / (365 * 24 * 60 * 60 * 1000);
console.log(`You are ${years} old!`);
```

הינה דרך החישוב:

מילישניות בשניה	1000
שניות בדקה	60
דקות בשעה	60
שעות ביום	24
ימים בשנה	365

מספר המילישניות בשנה הוא **לפיכך: 365X24X60X60X1000**, כלומר 31,536,000,000, מילישניות, שזה המונ. אבל אם מחלקים את מספר המילישניות במספר זהה, מගלים את מספר השנהים. מובן שיצא מספר עם הרבה ספרות אחרי הנקודה, אך בדיק לשם כך קיים Math.round.

נוהג לשמר תאריכים כמעט תמיד כמספר המילישניות (או מספר השניות, אז להכפיל באלף כמשמעותו) ולא כמחוזות טקסט. זה נראה מעט מסורבל בהתחלה, אבל בזכות האובייקט Date ממש קל לעשות את זה.

JSON

לא מעט פעמים נדרש ל בעבר נטוניים אל סקריפט מסוים או ממוני, למשל לקבל נתונים משרת ולבוד אם איטם או לטען או לשמר קובץ אם עובדים בג'אווהסקרייפט לצד השרת. ג'אווהסקרייפט בינה לעבוד היטב עם פורמט נתונים שנקרא **JSON**, ראשי תיבות של JavaScript Object Notation. בגדול, הפורט זהה הוא אובייקט ג'אווהסקרייפטי שעליינו למדנו כבר בעבר, אך בשינויים קלים:

1. מירכאות כפולות בלבד.
2. מירכאות סביב כל מחרוזת טקסט, גם סביב התוכנות.
3. אין פסיק בסוף התוכנה الأخيرة.
4. אסור לכתוב העורות בתוך JSON.
5. אין מתודות.

למשל, הקוד הבא הוא קוד ג'אווהסקרייפטי תקין המגדיר אובייקט, אך הוא אינו JSON תקין:

```
let myObject = {
  name: 'Ran',
  birthdate: 245849420500,
}
```

הקוד הבא הוא אותו קוד, אבל בפורט JSON:

```
{
  "name": "John",
  "birthdate": 245849420500
}
```

מה הבדל? כפי שנכתב לעיל – מירכאות כפולות סביב כל מחרוזת טקסט וגם סביב כל התוכנות ואין פסיק בסוף התוכנה الأخيرة.

כאמור, JSON אמור להיות אינטואיטיבי לכל מי שמכיר ג'אווהסקרייפט מספיק זמן. למעשה השינויים הקלים הללו, מדובר בפורט פשוט וקל, ומעט כל השפות בשרתים השונים עובדות אותו, מה שהופך אותו למשגוח. אבל כדי לעבוד אותו צריך להמיר אובייקט ג'אווהסקרייפטי ל-JSON ולהפוך. עושים את זה בקלות באמצעות אובייקט JSON הגלובלי, שיש לו שתי מתודות – האחת הופכת אובייקט ג'אווהסקרייפט ל-JSON תקין והאחרת ממירה JSON תקין לאובייקט ג'אווהסקרייפטי שאפשר להשתמש בו:

```
let someObject = {
  name: 'Ran',
  birthdate: 245849420500
}
let JSONObject = JSON.stringify(someObject);
console.log(JSONObject) // {"name": "Ran", "birthdate": 245849420500}
```

בדוגמה שלעיל לוקחים אובייקט ג'אווהסקרייפט והופכים אותו באמצעות `JSON.stringify` ל-JSON.JSON מופיע המניין בפורט של מחרוזת, ומכאן השם של המתודה: `stringify`:

להפוך למחוזת. את האובייקט הזה אפשר לשגר לשרת באמצעות פקודת AJAX, לשומר על השרת או להעביר אותו בכל צורה אחרת. הוא נחשב מחוזת טקסט לכל דבר. בצד השרת, כל שפה יודעת להתמודד עם JSON, שהפך דה פקטו לפורמט המרכז של הרשת.

שים לב שאובייקט JSON הוא מחוזת טקסט לכל דבר. כלומר דבר זהה:

```
let someObject = {
  name: 'Ran',
  birthdate: 245849420500
}
let JSONObject = JSON.stringify(someObject);
JSONObject.name = 'Moshe'; // Will get Uncaught SyntaxError: Unexpected
                           token o in JSON at position 1
```

לא יעבד ויזרק הודעה שגיאה. על מנת לעבוד עם מחוזת טקסט שהוא אובייקט JSON כשר למהדרין, צריך להמיר אותה לג'אווהסקריפט. את זה עושים באמצעות `JSON.parse` באופן הבא:

```
const JSONObject = '{"name": "Ran", "birthdate": 245849420500}';
let regularObject = JSON.parse(JSONObject);
regularObject.name = 'Moshe';
console.log(regularObject); // Object {name: "Moshe", birthdate:
                           245849420500}
```

כאן היה אובייקט JSON שהוא בעצם מחוזת טקסט. במקרה הזה הוא הוקלד ידנית, אבל במקרה הוא יגיע משרת, מממד נתונים או בכל צורה אחרת. באמצעות `JSON.parse` הוא הומר לאובייקט רגיל שלו ואפשר לעבוד כרגיל, למשל להחליף את שם הליקוט ל-Moshe כפי שנעשה כאן.

setTimeout

מדובר בפונקציה גלובלית שמאפשרת לעכב את הקוד או לזמן אותו. בדרך כלל בקוד רגיל אין בה שימוש, אבל בלי מעט דוגמאות משתמשים בקוד הזה. כמו כן, בקוד ג'אווהסקריפט שבודק אם יש חשיבות רבה לזמן או לעיכוב, אבל בדרך כלל השימוש בפונקציה הזאת נחשב לתוכנות גרוע. אם התוכנה שלכם צריכה עיכוב (`timeout`), סימן שיש בעיה בסקריפט. אז למה לומדים את הפונקציה הזאת? בעיקר כי צריכה לשם לימוד קוד אסינכרוני בהמשך.

נוסף על כך, זו פונקציה שכמעט נמצאת בדגימות רבות בראש כל מה הקשור לאספקטים מתקדמים יותר של ג'אווהסקריפט (כמו גנרטורים — שהם חלק מתקדם בשפה שלא נגיא לאילו בספר הזה, אבל אפשר לגשת ל-MDN שהוזכר קודם כדי לבדוק), אחרת לא הייתה שום סיבה לכתוב עליה במיוחד. אז מומלץ להתodium לסינטקס שלו ולהכיר אותה, אבל מואוד לא מומלץ להשתמש בה בקוד אמיתי בלי סיבה ממש- ממש טובה (סיבה טובה היא יצירת אינימציות או משחקים, למשל).

חמורים באזהרות האלו, בואו נלמד את הפונקציה של `timeout`. מדובר בפונקציה גלובלית שזמיןנה כמעט בכל הסביבות — דפדפן או סביבת שרת. איך היא עובדת? זו פונקציה שמקבלת שני ארגומנטים, פונקציה אונומית (המוחברת כפונקציית חץ) שרצה מיד אחרי זמן העיכוב, וזמן העיכוב ב밀ישניות.

הינה הפונקציה:

```
console.log('start the code');
setTimeout(() => {
  console.log('After 5 seconds');
}, 5000);
console.log('end the code');
```

פונקציית החץ, שמוחברת כารוגמנט הראשון, תרוץ מיד לאחר זמן העיכוב, שמוחבר כארוגמנט השני, יסתיים. הקוד לפיך ידפיס את מה שיש בקונסולה בשורה הראשונה — "start", וירץ את פונקציית העיכוב שתחכה חמיש שניות. הסקריפט לא יעצור וירץ את השורה الأخيرة "end of code". רק לאחר חמיש שניות, שהן 5,000 מילישניות, יוצג "After 5 seconds".

```
start the code
end the code
undefined
After 5 seconds
```

סדר הזרה

```
1 console.log('start the code');

  ↗️ ארוגמנט ראשון - פונקציה אונומית
2 setTimeout(() => {
  ↗️ ארוגמנט שני - מספר המילישניות
  3   console.log('After 5 seconds');
}, 5000);

2 console.log('end the code');
```

מה שחשוב להבין הוא ששאר הסקריפט רץ, וברגע שהזמן שהוקצב יסתיים, מה שהועבר לפונקציית החץ ירוץ מיד. לא כל הסקריפט עוצר; הזרה של שאר הסקריפט מתקיימת כרגע. פונקציית `setTimeout` מחדירה מספר חיובי שבו אפשר להשתמש כדי לזרה אותה בהמשך הקוד. לאיזה צורך זה שימושי? אם רוצים מסיבה כלשהי בהמשך להשמיד את ה-`setTimeout`. ההשמדה של `setTimeout` נעשית באמצעות פונקציה מובנית אחרת שנקראת `clearTimeout`, והיא מקבלת כารוגמנט את המספר המזהה של ה-`setTimeout` אותו רוצים להרוג:

```
console.log('start the code');
const timeoutID = setTimeout(() => {
  console.log('After 5 seconds');
}, 5000);
console.log('end the code');
console.log(timeoutID); // Some positive number
clearTimeout(timeoutID); // setTimeout will never happen
```

בדוגמה שלעיל יוצרים setTimeout בדיקן כמו בדוגמה הקודמת, אבל הפעם מקבלים את ה-ID שלה ומשתמשים בו כדי להרוג מידית את ה-setTimeout, שלעולם לא יתבצע.

תרגיל:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד 100 (לא כולל 100).

פתרון:

```
function getRandom() {  
    let result = Math.random();  
    result *= 100;  
    result = Math.round(result);  
    return result;  
}  
const randomNumber = getRandom();  
console.log(randomNumber);
```

הסבר:

יוצרים פונקציה עם שם כלשהו. ראשית יוצרים מספר רנדומלי באמצעות `Math.random`. המספר הוא בין 0 ל-1 (לא כולל 1). מכפילים את התוצאה ב-100. כך למשל אם המספר הרנדומלי שיצא הוא 0.512, התוצאה תהיה 51.2. כיוון שחייבים להחזיר מספרים שלמים, לוקחים את המספר, מעגלים אותו באמצעות `Math.round` ומוחזירים אותו כמו כל פונקציה רגילה.

תרגיל:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד הארגומנט שהפונקציה מקבלת

פתרונות:

```
function getRandom(span = 10) {  
    let result = Math.random();  
    result *= span;  
    result = Math.round(result);  
    return result;  
}  
const randomNumber = getRandom(1000);  
console.log(randomNumber);
```

הסבר:

כמו בפתרון הקודם, רק שכאן מכפילים את המספר הרנדומלי בארגומנט שמקבלים מהפונקציה, כך שהטוווח הוא בין 0 לארגומנט.

תרגיל:

צרו פונקציה שמקבלת גיל משתמש בשנים (למשל 1980) ובחודש (למשל 1) ומחשבת אם המשתמש בן יותר מ-21. אם כן, היא מחזירה true. אם לא, היא מחזירה false.

פתרונות:

```
function verify21(costumerYear, costumerMonth) {
    const currentDate = new Date();
    const costumDate = new Date(costumerYear, costumerMonth);
    const age = currentDate - costumDate;
    const years = age / (1000 * 60 * 60 * 24 * 365);
    if (years > 21) {
        return true;
    } else {
        return false;
    }
}
console.log(verify21(1977, 9)); // true
console.log(verify21(2017, 1)); // false
```

הסבר:

יוצרים פונקציה שמקבלת שני ארגומנטים, הראשון של שנה והשני של חודש. ראשית יוצרים אובייקט תאריך של התאריך הנוכחי, ללא ארגומנט. שנית יוצרים אובייקט תאריך של שנת הלידה והחודש. כל מה שנותר לעשות הוא לחשב את הפעור. הוא מתќבל במילישניות, וכך להסב אותו לשנים מחלוקת אותו במספר המילישניות שיש בשנה. oczywiście יש את מספר השנים שהוא הפעור בין גיל המשתמש לתאריך הנוכחי. אם מספר השנים גדול מ-21 מוחזרים true, ואם מספר השנים קטן מ-21 מוחזרים false. איזה יופי.

תרגיל:

צרו פונקציה שמקבלת מספר X ומדפיסה בקונסולה את המשפט "רצתי לאחר X שניות".

פתרון:

```
function delayFunction(int) {
  const delaySeconds = int * 1000;
  setTimeout(() => {
    console.log(`After ${int} seconds`);
  }, delaySeconds);
}
delayFunction(2);
```

הסבר:

יצרים פונקציה בשם delayFunction שיש לה ארגומנט זהה מכפילים ב-1,000 כדי לקבל מספר במילישניות ומוכניסים אותו לקבוע delaySeconds. קוראים לפונקציה הגלובלית setTimeout. הפונקציה הזאת קיימת בכל ג'אווהסקריפט חלק מהשפה, לא צריך ליצור אותה. היא מקבלת שני ארגומנטים. הראשון הוא פונקציה שריצה אחריו מספר המילישניות שמעבירים בארגומנט השני. בארגומנט הראשון מעבירים פונקציה אונונימית מסווג חז, שעליה למדנו בפרק על הפונקציות, שכל מה שהיא עשוה הוא להדפיס בקונסולה "אחרי X שניות". שימושו לב שההדפסה נעשית עם תבנית שגם עליה כבר למדנו, בפרק על תבנית טקסט.

בארגומנט השני שהוא מקבלת מעבירים את delaySeconds – מספר המילישניות. זה הכל. נותר רק לבדוק את הקוד ולראות שהוא עובד.

ביטויים רגולריים

בביטוי רגולרי משתמשים על מנת לעבוד עם טקסטים, והוא פשוט טקסט שמאפשר לבצע חיפוש בטקסט אחר. לא מעט פעמים מקבלים לתוכה משתנה כמוות גודלה של טקסט שרצים לעבוד איתה – למשל לבדוק אם מילים מסוימות נמצאות או לא נמצאות שם וכו'. אם רצים לדוגמה לבצע אימות של אימייל, אפשר להשתמש בביטוי רגולרי פשוט כדי להבין אם מדובר באימייל או לא. הדרך הכח הטובה למצוא מילים, משפטים או כל דבר אחר בטקסט היא באמצעות ביטוי רגולרי.

קשה לתכנת בכל שפה ובמיוחד בג'אווהסקריפט ולא להיתקל בביטוי רגולרי. למשל, אם משתמש בוחר סיסמה שלא מציאות לכל מוסים או לחיפוש והחלפה של טקסט. בכל עם שעושים פעולה מורכבת בטקסט או מנתחים אותו, צריך להשתמש בביטוי רגולרי. ביטויים רגולריים נמצאים בהמון שפות ולא רק בג'אווהסקריפט.

בג'אווהסקריפט, ביטוי רגולרי הוא מוקף בסלאשים (לוכסנים) בלבד, אף על פי שהוא מסווג אובייקט, וככה אפשר לזהות אותו. ביטוי רגולרי יכול להיראות כך:

```
const regularExpression = /abc/;
```

כל מה שיש בין הסלאשים הוא הביטוי הרגולרי. מיד אחרי הסלאש האחרון יש אופציות (modifiers). למשל:

```
const regularExpression = /abc/ig;
```

מقبول מאד להציב מיד אחרי הביטוי הרגולרי אופציות. במקרה זהה האותיות `i` ו-`g` מסמלות אופציות – גם הביטוי זהה תקין לחלווטין. יש לא מעט אפשרויות שונות שאפשר להציג. הינה טבלה קצרה של אפשרויות לדוגמה:

מה הוא עושה	התו המיצג את ה-modifier
הביטוי הרגולרי לא יתייחס להבדל בין אותיות הראשיות לרגילות. abc יהיה זהה ל-ABC או ל-ABc וכן הלאה	i
הביטוי הרגולרי יתחשב בריבוי שורות בסימן של התחלת או סוף	m
כאשר יש כמה התאמות בחיפוש, הביטוי הרגולרי יחזיר את כלן במקומות אחד בלבד	g

אפשר ליצור ביטוי רגולרי גם באמצעות פונקציה בנאית:

```
const regularExpression = new RegExp('abc', 'ig');
```

אם משתמשים בפונקציה בנאית, אפשר להשתמש במשתנים בביטוי הרגולרי. כל הדוגמאות להלן ייעשו בדרך הראשונה והפשוטה יותר, אבל אתם מוזמנים לנסות גם את הדרך השנייה, באמצעות הפונקציה הבנאית, כאשר אתם מתרגלים.

הביטוי הרגולרי הראשון הוא /abc/ שאומר פשוטו כמשמעו – כל ביטוי שכולל את האותיות abc ללא רוח ביןיהן ולא אותיות אחרות, כולל המחרוזת הפשטota abc. נראה איך ג'אווהסקרייפט עובדת עם ביטויים רגולריים ואחר כך נעמיך לטור הביטויים ומשמעותם, ולשם כך הביטוי הבסיסי הזה יספיק. כאשר יוצרים ביטוי רגולרי, יש לו באופן אוטומטי את המתוודות של הביטוי הרגולרי (בדיקה כמו שלמערך יש את המתוודות של forEach למשל).

הmethodה הראשונה שנלמד היא `test`, שמאפשרת לבדוק שהביטוי הרגולרי נמצא במחוזת טקסט כלשהי. נניח שורצים לבדוק אם יש בביטוי `abcdef` את הביטוי הרגולרי `/abc/`:

```
const regularExpression = /abc/;
const textString = 'abcdef';
const result = regularExpression.test(textString);
console.log(result); // true
```

ראשית, יוצרים משתנה המכיל ביטוי רגולרי. הביטוי הרגולרי מוקף בסלאשים בלבד, ללא מירכאות. זה מראה שהוא ביטוי רגולרי. שנית, יוצרים מחוזת טקסט שאותה בודקים. בזמנים האמיטיים, מן הסתם, מחוזת הטקסט זה תועבר על ידי המשמש או על ידי שירות חיצוני כלשהו, אבל כאן, לשם הדוגמה, נכתב אותה לתוך המשתנה קבוע.

כיוון שהמשתנה `regularExpression` הוא ביטוי רגולרי, יהיו לו מתודות של ביטוי זהה. מתודת `test` מקבלת כารוגמנט מחוזת טקסט ואם הביטוי הרגולרי נמצא במחוזת הטקסט, מחזירה `true`. זה בדיק מה שהיא עשו כאן, כיוון שיש `abc` ב-`abcdef`.

אם ננסה את זה במחוזת אחרת, שלא כוללת את הצירוף `abc`, יתקבל `false`:

```
const regularExpression = /abc/;
const textString = 'Nothing is here';
const result = regularExpression.test(textString);
console.log(result); // false
```

ובן שביטוי רגולרי יכול להיות הרבה יותר מורכב מזה. אפשר לסמן, למשל, שורצים שהביטוי יהיה נכון רק אם מחוזת הטקסט מתחילה ב-`abc`. לדוגמה, הביטוי `abcdef` יעבור את הבדיקה, אבל `defabc` לא יעבור. את זה עושים באמצעות התו `^` שמשמעותו "התחלת", אם הוא לא נמצא בתוך סוגרים מרובעים. הביטוי `/^abc/` יהיה נכון רק לגבי ביטויים:

כאמו:

```
const regularExpression = /^abc/;
const textString = 'abcdef';
const result = regularExpression.test(textString);
console.log(result); // true
```

אתם מוזמנים ללקח את הדוגמה זו ולנסות לשנות את מחזורת הטקסט לכל דבר אחר שמכיל abc אבל לא בתחילת המחרוזת. הביטוי הרגולרי לא יהיה נכון ויתקבל false:

```
const regularExpression = /^abc/;
const textString = 'defabcdef';
const result = regularExpression.test(textString);
console.log(result); // false
```

כמו שיש התחלת, יש גם סוף. התו \$ מסמל את סוף המשפט שהביטוי הרגולרי מסתיים בו:

```
const regularExpression = /abc$/;
const textString = 'defabc';
const result = regularExpression.test(textString);
console.log(result); // true
```

הביטוי הרגולרי זהה יהיה תואם אך ורק את מה שנגמר ב-abc. אפשר לשלב ולסמן גם התחלת ו גם סוף, מה שאומר שرك הביטוי abc בלבד יתאים:

```
const regularExpression = /^abc$/;
const textString = 'abc';
const result = regularExpression.test(textString);
console.log(result); // true
```

כל ביטוי אחר לא יתאים, כי תחמננו את abc בהתחלה ובסוף.

אפשר גם לציין אותיות חופשיות (wildcards). למשל, אם רוצים ביטוי שמתחל ב-a ומסתיים ב-bc ובאמצע יש רק אות אחת, אפשר להשתמש בביטוי ". ". (נקודה), שמשמעותותו תו אחד (בלבד) מכל סוג:

```
const regularExpression = /^a.bc$/;
const textString = 'a1bc';
const result = regularExpression.test(textString);
console.log(result); // true
```

אפשר לציין גם ביטוי תו אחד לפחות באמצעות הסימן +. + משמעותו 1 או יותר. או יותר ממה? ממה שמשמעותו לפניו. למשל:

+a – הכוונה היא לאות a אחת או יותר.
+ – הכוונה היא לכל סימן אחד או יותר.

כלומר הביטוי:

`^a.+bc$`

תקף גם ל-a1bc ו גם ל-a12bc ו גם ל-123bc a וכך הלאה. ולא רק מספרים נכילים כאלו, אלא גם אותיות, רווחים, סימנים מיוחדים וכל דבר אחר. נקודה () היא סוגתו כללי.

אפשר גם לציין את סוג התו. למשל, אם רוצים רק מספרים, ללא אותיות או סימנים מיוחדים, לא משתמשים בביטוי שמשמעותו "כל אות" אלא בביטוי:

`^a[0-9]+bc$`

מה הביטוי הזה מציין? שהטקסט חייב להתחיל ב-a ולהסתיים ב-bc. בamuן יש מספר [0-9] אחד או יותר (זו שמעות ה-+).
אפשר להשתמש גם בתווים. למשל, CANאפשרות קטנות גם מספרים:

`^a[0-9a-z]+bc$`

הביטוי הזה מתחילה ב-a ומיד אחריו מופיע הביטוי [z-9a-0], שמשמעותו כל אות בין 0 ל-9 או בין a ל-z, כלומר אותיות קטנות ומספרים. כמה פעמים? "+" – כלומר פעמי אחת או יותר.

אפשר להשתמש באיזה טווח שורצים, למשל A-Z או 0-5.

לא חייבים להשתמש ב-+, יש עוד כמה ביטויים כמו:

* – כוכבית, 0 עד אינסוף.

? – סימן שאלה, 0 עד 1.

אפשר לסקם את הביטויים הרגולריים בטבלה הבאה:

סימן	פירוש הסימן
^	התחלת – מה שאמור להגיע בתחילת הביטוי
\$	סוף – מה שיש בסוף הביטוי
.	כל תו
[]	בתוך הסוגרים המרובעים יהיו טווחים של אותיות
+	אחד לפחות מהביטוי המופיע לפני
?	אפס או יותר אחד מהביטוי המופיע לפני
*	אפס או יותר מהביטוי המופיע לפני

תחום הביטויים הרגולריים הוא עולם ומלאו, וספרים שלמים מוקדשים לו. חשוב לדעת שיש דבר זה ומשתמשים בו לניתוח טקסט. שליטה בביטויים רגולריים היא מעבר לתחומו של ספר זה, אבל אתם חייבים לדעת שיש דבר זה ושלוט בו באופן בסיסי ביותר, מכיוון שמדובר בחלק משפט ג'אווהסקריפט שמתכונתים משתמשים בו.

בעברית הביטויים הרגולריים קשים יותר. אי-אפשר לכתוב טקסט בעברית אלא צריך להשתמש בייצוג היוניקוד שלו. יוניקוד הוא הקידוד שבו מקובל להשתמש בשנים האחרונות והוא מונע את כל הגיבריש. כל צורות הכתב בעולם נמצאות ביוניקוד, וכל אות ואות בכל מערכת כתוב מיוצגת על ידי מספר סידורי. כך למשל האות א' ביוניקוד היא 0D05+U, וזה מה שחייב לכתוב, וכן, יש גם יוניקוד ל... ניקוד. כאמור, הנושא זהה לא מכוסה בספר זהה.

תרגילים:

נתונה מחרוזת הטקסט "ahla bahla". כתבו ביטוי רגולרי הבודק אם היא כוללת את המילה ahla.

פתרונות:

```
const regularExpression = /ahla/;
const textString = 'ahla bahla';
const result = regularExpression.test(textString);
console.log(result); // true
```

הסבר:

יצרים קבוע בשם regularExpression ומכוונים אליו את הביטוי הרגולרי ahla, שמשמעותו מחרוזת טקסט של ahla. הביטוי הרגולרי מוקף ב-/. קר מסמנים ביטויים רגולריים. לאחר מכן יוצרים קבוע שבו יש את מחרוזת הטקסט שאותה בודקים.

השלב הבא הוא להשתמש במתודות. כיוון שלכל ביטוי רגולרי יש אותה באופן אוטומטי, יש את המתודה הזו גם ל-regularExpression שמכיל ביטוי רגולרי. המתודה הזו מקבלת טקסט כARGINENT, זהה בבדיקה מה שהועבר לה.

תרגילים:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אות גדולה אחת לפחות.

פתרונות:

```
const regularExpression = /[A-Z]+/;
const textString = 'passworD';
const result = regularExpression.test(textString);
console.log(result); // true
```

הסבר:

הביטוי הרגולרי הוא מה שחשוב פה. הסוגרים המרובעים מצינים טווח: [A-Z]. [A-Z] משמעו כל אות גדולה שהיא. כמה אותיות גדולות אנחנו רוצים? אחת או יותר. לפיכך משתמשים ב-+.

הביטוי:

[A-Z]+

משמעותו אחת לפחות מהטוווח A-Z, כלומר אחת גדולה לפחות. אחרי שמנסחים את הביטוי הרגולרי, אפשר לבדוק אותו בכל מחרוזת טקסט. יוצרים אותו באמצעות // ומכוונים אותו לקבוע בשם regularExpression. regularExpression מחרוזת טקסט (כמו בדוגמה: D) (passworD) ומשתמשים במתודה test, שקיימת באופן אוטומטי בכל ביטוי רגולרי, על מחרוזת הטקסט הזו. המתודה מחזירה true אם המחרוזת מצויה לפחות לביטוי הרגולרי. כיוון שיש בה אחת גדולה, היא מצויה ויתקבל true.

תרגילים:

נתונה סיסמה. כתבו ביטוי רגולרי הבזק אם היא כוללת אות גדולה אחת לפחות ומספר אחד לפחות. כשהאות היא לפני המספר. למשל `passworD1`.

פתרון:

```
const regularExpression = /[A-Z][0-9]/;
const textString = 'passworD1';
const result = regularExpression.test(textString);
console.log(result); // true
```

הסבר:

כמו בתרגיל הקודם, עיקר הבעיה פה הוא לנסח את הביטוי הרגולרי. במקרה זהה:

[A-Z]

הטוויך הוא A-Z (כליומר אות גדולה).

[0-9]

הטוויך הוא 0-9 (כליומר מספר).

הביטוי המשלב משמעו מחרוזת טקסט שיש בה אות גדולה אחת לפחות ומיד אחריה מספר אחד. אחרי שמנסחים את הביטוי הרגולרי, יוצרים אותו בג'אווהסקרייפט בין // ומכניסים אותו לקבוע `regularExpression`. לקבוע זה, מחריג שיש בו ביטוי רגולרי, יש את מתודת `test` שמקבלת כารגוומנט מחרוזת טקסט, במקרה זהה את הסיסמה שהזונה, ובזק אם היא מציינת לתנאי.

טיפול בשגיאות

כמתכנת ג'אווהסקריפט מנוסים, בוודאי יצא לכם לראות שגיאות בקונסולה במהלך העבודה. השגיאות הללו נוצרות בדרך כלל אם טועים בסינטקטו, למשל בקריאה למשתנה שלא ממש הוגדר. אם למשל כתובים:

```
console.log(myVar);
```

ולא טורחים להגדיר קודם קודם את `myVar`, מקבלים שגיאה כזו:

Uncaught ReferenceError: myVar is not defined

אם משתמשים בדפדפן, רואים את השגיאה בקונסולה באופן הבא:

 **Uncaught ReferenceError: myVar is not defined**

הודעות השגיאה מכילה מידע גם על השגיאה וגם על המיקום שלה. עד עכšíו הת'יחסתי אל שגיאות כל שהוא בלתי רצוי, אבל הרבה פעמים כן רוצים שגיאות. שגיאות והטיפול בהן הם חלק מכל מערכת ופונקציה חשובה. אם למשל כתובים פונקציה שבודקת סיסמה והמשתמש מכניס סיסמה בעברית בלבד, זו שגיאה שחייבים לTrap ולטפל בה. אם כתובים פונקציה שגוראת קובץ לא שם, חייבים ליצור שגיאה עבור מי שכתב את הפונקציה.

יצירת שגיאה היא ממש פשוטה. שגיאה היא אובייקט שנוצר מפונקציה בנית ויצרים אותו כך:

```
const myError = new Error('Oy Vey');
throw myError;
```

ראשית יוצרים את השגיאה עם הטקסט המיחד לה. במקרה זהה "Oy Vey". אחרי שיש אובייקט שגיאה אפשר בכל שלב של הסקריפט לזרוק אותו. ברגע שיש שגיאה, כל קוד שירוץ אחרת יפסיק לrozץ מידית, ממש כמו שגיאת הרצה מהסוג המוכר (והאהוב?), כלומר הקוד הזה:

```
function testMe() {
  const myError = new Error('Oy Vey');
  throw myError;
}
testMe();
console.log('Hello'); // Will never run
```

ההדפסה של Hello בקונסולה לא תרוץ לעולם, אלא אם כן מדובר במקרה נדיר שבו חיבבים להפיל מערכת. בדרך כלל יוצרים שגיאה כחלק מהניסיונות לנהל משהו. שגיאה שמפילה את כל הסקריפט לא ממש תעזר כאן. בגלל זה לרוב זורקים את השגיאות לתוך בлок שנקרא `.try-catch`.

בלוק של `try-catch` הוא בעצם דרך לזרוק שגיאות ולטפל בהן בלי השבתה של הסקריפט כלל. בגדול הוא מורכב מהמילה השמורה `try` שאחריה סוגרים מסולסים, וביהם הקוד רץ. אם לא

תהייה שגיאה — טוב ויפה. אם תהיה שגיאה, קוד השגיאה ייכנס לבlok catch, שבו יש את ארגומנט השגיאה:

```
function testMe() {
  const myError = new Error('Oy Vey');
  throw myError;
}

try {
  testMe();
} catch (error) {
  console.log(error);
}
console.log('Hello');
```

```
function testMe() {
  const myError = new Error('Oy Vey');
  throw myError;
}

try {
  testMe();
} catch (error) {
  console.log(error);
}
console.log('Hello');
```

הפלט שיציר בקונסולה

Error: Oy Vey
at testMe (VM552 pen.js:2)
at VM552 pen.js:7

Hello

אם יש שגיאה היא תגיע ל catch

ה- catch מקבל את ארגומנט השגיאה שנזרקה

הקוד הזה יירץ כרגיל

בבלוק של ה-try קוראים לפונקציה. אם הפונקציה רצתה ללא שגיאה, הכל תקין והקוד מתנהל כרגיל. אם יש שגיאה, הבלוק של catch נכנס לפעולה. כל קוד אחר שמחוץ ל-try-catch רץ כרגיל והסקריפט לא קורס. במקורה הזה, testMe, מוחץ ל-try-catch, תודפס כרגיל. ההדפסה של ה-Hello, מוחץ ל-Hello, מוחץ ל-try-catch, תודפס כרגיל. כל קוד שעטוף ב-try ויזרק שגיאה, אפילו שגיאת קומפילציה, לא יפריע למהלך התקין של הקוד ויהי אפשר לנוהל את דרך הטיפול בשגיאה באמצעות ה- catch. האם תדפיסו הודעה למשתמש? האם תנסו להריץ מחדש מוחץ את הפונקציה? הכל אפשרי.

בדרך כלל מקובל שמודולים זורקים שגיאות וסומכים על מי שמאפיין אותם שיטפל בהן לפי מיטב הבנותו. נניח למשל שיש אובייקט זהה:

```
const passwordTester = function (password) {
  if (/[^A-Z]+/.test(password) === false) {
    const error = new Error('No capital in password');
    throw error;
  }
  if (password.length < 8) {
    const error = new Error('Password is shorter than 8');
    throw error;
}
```

```
    this.password = password;
};
```

האובייקט בודק סיסמאות והוא אובייקט מורכב שנבנה בעזרת פונקציה בנית, מהסוג שכבר למדנו עליו. במקרה זהה, בתוך האובייקט עצמו בודקים את הסיסמה באמצעות ביטוי רגולרי. אם הסיסמה לא מכילה אות גדולה, זורקים שגיאה מסוג אחד. אם הסיסמה קצרה מדי, זורקים שגיאה מסוג אחר. כל זריקה של הסיסמה עוצרת את הסקריפט בתוך האובייקט.

אם מישמים את האובייקט כמו שצרכיך, צריך לעטוף אותו ב-try ו-`catch`:

```
try {
    new passwordTester('12345678');
} catch (error) {
    console.log(error);
}
```

בתוך ה-`catch` אפשר לנוהל את השגיאה, כלומר להדפיס אותה למשתמש, למשל. מקבלים את האובייקט של השגיאה, והטקסט מופיע בתוך אובייקט השגיאה תחת השדה `message`. אפשר להדפיס את הטקסט של השגיאה, כמובן, או לעשות בו מה שרצים.

כמעט לכל מודול חיצוני או אפילו פנימי בג'אווהסקריפט יש מדיניות שגיאות שאפשר לTrappear ולנהל. לא את כל השגיאות אפשר לנוהל בצורה ממש טובה, והבוחירה מה לעשות היא של המתכונת; לפעמים מעדיפים להשבית את כל הסקריפט, לפעמים מציגים הודעה שגיאה, לפעמים מתקנים את הקלט ומריצים שוב — מה שיש בתוך ה-`catch` הוא באחריותכם.

זו גם הסיבה שבגללה זה לא נחשב להברקה גדולה לעטוף את כל הקוד ב-try-`catch`. כהה רק מונעים הדפסת שגיאות, אבל מפספסים את הנוקודות החשובות של ניהול שגיאות — ניהול השגיאה הוא חשוב אם יודעים מאייה מגיעה ומה לעשות אליה. אם הסקריפט מורכב, בשלב העליון (try-catch) אי-אפשר לעשות כלום עם השגיאות וקשה לדעת מהין הן הגיינו. לפיכך, מומלץ שתעטפו את חלקו הקוד שלכם ב-try-`catch` רק אם יש לכם תוכנית מסודרת בוגריה למה שצריך לעשות במקרה של קריישה או בנסיבות שאתם צופים שבהם יופיעו שגיאות.

finally

כתחספת ל-`try` יש את `finally`. זהו בלוק קוד נוסף שניתן להוסיף לביטוי ה--`try` והוא תמיד יrotch אחריו, בין `try` יעבד ובין `try`-`catch`:

```
const passwordTester = function (password) {
    if (/[^A-Z]+/.test(password) === false) {
        const error = new Error('No capital in password');
        throw error;
    }
    if (password.length < 8) {
        const error = new Error('Password is shorter than 8');
        throw error;
    }
}
```

```

    }
    this.password = password;
};

try {
    new passwordTester('12345678');
} catch (error) {
    console.log(error);
} finally {
    console.log('Try again!');
}

```

אם תרצו את הקוד הזה, תראו `sh-hello.js` בקורס זהה, גם אם תשים את הסימנה למשהו שעובר. בדרך כלל משתמשים ב-`finally` על מנת לבצע עבודות ניקיון. למשל אם מבצעים טעינה של משאב, לפני ה-`try-catch` תהיר הצגה של איקון טעינה וב-`finally` תהיר מחיקה שלו.

תרגילים:

כתבו פונקציה שמקבלת שני ארגומנטים שאמורים להיות מספרים, מחברת אותם ומחזירה את התשובה. אם אחד הארגומנטים הוא לא מספר (אפשר לבדוק את סוג הארגומנט באמצעות `typeof`), צרו שגיאה.

פתרונות:

```

function addNumbers(arg1, arg2) {
    if (typeof arg1 !== 'number' || typeof arg2 !== 'number') {
        const numberError = new Error('What?!? no number?? :( ');
        throw numberError;
    } else {
        return arg1 + arg2;
    }
}

console.log(addNumbers(3, 2)); // 5
console.log(addNumbers('3', 2)); // Uncaught Error: What?!? no number?? :( 

```

הסבר:

על הפונקציה עצמה אין מה להזכיר מילימ. יש כאן פונקציה שמקבלת שני ארגומנטים ובודקת אם שפט תנאי פשוט אם אחד מהם הוא לא מספר. אם הוא לא מספר, יוצרים שגיאה וזרקים אותה. שתי השורות האלו הן לב התרגילים:

```

const numberError = new Error('What?!? no number?? :( ');
throw numberError;

```

כאן זורקים את השגיאה שיש בפונקציה. אפשר לראות שהשגיאה הזה תודפס גם אם אין כאן כל שגיאת קומפילציה. ג'אווה סקריפט בהחלט יכולה לחבר מחרוזת טקסט ומספר (היא פשוט תמיר את המספר לטקסט), אבל במקרה הזה, הפונקציה לא מאפשרת זה ותזרוק שגיאה.

תרגיל:

נתון משתנה שהוא מספר. כתבו פונקציה שמקבלת אותו, ואם מדובר במספר שלילי הוא זורק תשגיאה. עטפו את הפונקציה שכתבתם ב-try-catch. אם הפונקציה תזרוק תשגיאה, הציבו 0 במשתנה.

פתרונות:

```
function checkNegative(arg1) {
  if (arg1 < 0) {
    const numberError = new Error('Negative Number');
    throw numberError;
  }
}
let number = -1;
try {
  checkNegative(number);
} catch (e) {
  number = 0;
}
console.log(number); // 0
```

הסביר:

הfonקציה פשוטה למדי, היא מקבלת ארגומנט. אם הארגומנט הוא שלילי, זורקים תשגיאה באמצעות:

```
const numberError = new Error('Negative Number');
throw numberError;
```

כדי לנהל את השגיאה, עוטפים את הקוד ב-try-catch. אם יש תשגיאה, מופיעים את המשתנה. זה הכל. מה שיש בתוך try יזרוק את השגיאה אך לא יפריע למלאר התקין של שאר הסקריפט. מה שיש בתוך catch ינהל את השגיאה. במקרה זהה דרך ניהול השגיאה היא לאפס את המשתנה.

פרק 16

מבנה נתונים מסובב SET-1



מבנה נתונים מסוג **Set**-**Map**

בפרק על מערכים למדנו שאפשר לאחסן בהם נתונים. אבל יש בעיה – במערכות, המפתחה הוא מספר ואי-אפשר לקבוע אותו. אפשר לאחסן מידע באובייקטים, אך לא נוח לעשות זאת כיון שקשה לעבוד עם אטרציות באובייקטים, קשה למחוק ערך לפי המפתח שלו וקשה למספר את מספר הערכים. ג'אויסקייפ מספקת שני מבני נתונים מותחנים יותר לאחסן נתונים, וכן אפשר להנות מהתכונות של אובייקטים אבל גם מהפונקציות של המערכים.

Map

מבנה הנתונים הראשון מאפשר להכניס ולאחסן ערכים בפורמט `key value` פשוט שבפשטוטים. ברגע לאובייקט, אפשר לעשות עליו לולאות פשוטות. אין ערכים כפולים. בוגד, הוא דומה לערך, אלא שבמקום מספרים יש שם מפתח וערך. על מנת ליצור `Map` צריכים להשתמש בפונקציה הבנאית `Map` `new`, שמחזירה אובייקט שיש לו את התכונות ואת המתודות של `Map`.

למשל, כך יוצרים `Map` בסיסי עם המפתחות `username`, `city` ו-`phone`:

```
let myMap = new Map();
myMap.set('username', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
```

הfonקציה `set` מאפשרת להכניס ערכים ל-`Map` והערכים יכולים להיות, כמובן, כל דבר, לא רק מחרוזות טקסט כמו גם מספרים, מערכים ואפילו `Map` אחר. בזמן היצירה אפשר ליצור את `Map` עם מפתחות. למשל:

```
let myMap = new Map([['username', 'Ran'], ['city', 'Petah Tiqwa']]));
```

איך מוחזרים את השמות? באמצעות `get` שיש בה ארגומנט של המפתח המתאים. ממש כך:

```
const phone = myMap.get('phone');
console.log(phone); // 6382020
```

אם רוצים לדעת כמה איברים יש במערך, משתמשים בתכונת `size`. שימוש לב שבשונה מערכים, כאן משתמשים ב-`size` ולא ב-`length`.

הינה דוגמה:

```
const size = myMap.size;
console.log(size); // 3
```

מחיקה נעשית באמצעות המתודה `delete`, שמקבלת ארגומנט את המפתח של הערך שרצים למחוק. למשל:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
console.log(myMap.size); // 3
myMap.delete('city');
console.log(myMap.size); // 2
```

שים לב שבניגוד למערך, שם אם מוחקים איבר האורך שלו נותר כשהיה, כאן האורך משתנה בהתאם לערכיהם.

על מנת למחוק את כל המפה, צריכים להשתמש במתודה `clear` שלא מקבלת שום ארגומנט:

```
console.log(myMap.size); // 3
myMap.clear();
console.log(myMap.size); // 0
```

אפשר להשתמש בולולאת `forEach` ממש פשוטה על מנת לעבור על כל `Map`, ממש כמו מערך, אלא שכאן המפתחות הם לא מספרים:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
myMap.forEach((value, key) => {
  console.log(key);
  console.log(value);
})
```

גם לולאת `for` עובדת יפה עם `Map`.

בגدول, מבנה הנתונים `Map` אמור להכיל כל מבנה נתונים שהמפתחות שלו הם לא מספרים, יש המון כלו, מאובייקט של משתמש, דרך אובייקט של פעולה שהמשתמש עושה ועוד כל דבר עצמו. היתרון האדריר של `Map` הוא שאפשר להיות בטוחים שיש רק מפתח אחד מכל סוג. אין חזרות ולא צריך לנתח את הדופליקציות. נוסף על כך מרווחים יופי של איטרציות, ממש כמו במערכות.

Set

מדובר במבנה נתונים דומה ל-`Map`, אך מעט פשוט יותר, שיש בו ערכים בלבד ללא מפתחות. בנויגוד למערכים, שבהם המפתחות הם מספרים, או לאובייקטים ול-`Map`, שבהם יש מפתחות

שיכולים להיות מחרוזת טקסט רגילה, ב-Set יש רק ערכים - כלומר אין ערכים כפולים. אם מנסים להכניס ערך ל-Set והוא כבר קיים, הוא פשוט "ידرس". Set נוח מאד לאחסן נתונים כמו תגיות, רשימת מצרכים וכל נתונים שהוא שוחש בו הוא רק הערך שלו.

יצירת Set נעשית גם היא באמצעות פונקציה בנית, כמו יצירת Map:

```
const tags = new Set();
```

מהרגע זהה יש שאפשר להכניס לתוכו ערכים. שימוש לב שהכנסתי את ה-Set לתוכו קבוע. מהנקודה הזאת אינ-אפשר להכניסו לאחר מכן הסתם אפשר להכניס ערכים לתוכו ה-Set.

איך מכניסים ערכים? באמצעות המתודה add, שקיימת לכל משתנה מסווג Set. הארגומנט היחידי שיש בה הוא הערך:

```
tags.add('tag1');
tags.add('tag2');
tags.add('tag3');
console.log(tags); // Set(3) {"tag1", "tag2", "tag3"}
```

ל-Set יש לבדוק אותן האיות שיש למערכים ול-Map, כלומר אפשר בנותה להשתמש ב-`forEach`. בפונקציית הקולבך של ה-`forEach` יש כמובן רק את הערך, כי אין מפתחות ב-Set:

```
tags.forEach((value) => {
  console.log(value); // tag1, tag2, tag3
});
```

נחמוד, נכון? אפשר לבדוק אם יש ערך מסוים ב-Set `has` בעזרת מתודת `has`, שמחזירה או `true` או `false`:

```
tags.has('tag1'); // true
tags.has('Not here'); // false
```

כמו ב-`Map`, גם כאן מוחקם ערכי במתודת `delete` ובודקים את גודל ה-`Set` בעזרת תכונת `size`. מובן שאין מפתח אלא משתמשים בערך בלבד:

```
console.log(tags.size); // 3
tags.delete('tag1');
console.log(tags.size); // 2
```

כל ה-`Set` מתנקה אוטומטית בעזרת מתודת `clear`. כן, גם כאן יש דמיון ל-`Map`:

```
tags.clear();
console.log(tags.size); // 0
```

כאמור, `Set` שימושי מאד לאחסנת ערכים שאין צורך במפתחות שלהם, כגון רכיבי מתכוון, תגיוט, שמות וכו'. `Set` יכול להכיל כל נתון, ולא רק מחרוזות טקסט.

תרגיל:

צרו Map שיכיל עיר, רחוב, מספר בית ומיקוד.

פתרון:

```
const address = new Map();
address.set('city', 'Petah Tiqwa');
address.set('street', 'Kaplan');
address.set('streetNumber', 10);
address.set('zip', '6382020');
console.log(address);
```

הסבר:

יצרים קבוע בשם address מסוג Map. שימושו לב שהשתמשתי במילה השמורה `new` על מנת להפעיל את הפונקציה הבנائية `Map`. מחרגע שיש `Map` בתוך הקבוע, אי-אפשר לשנות את הסוג שלו, אבל כמו אובייקט שנכנס לתוכו קבוע – אפשר להוסיף ל-`Map` ולהוריד ממנו. זהה בדיק מה שעושים – ברגע שהקבע `address` הוגדר כ-`Map`, אפשר להשתמש במתודה `set` על מנת להכניס לתוכו מפתחות וערכים. הדפסה של האובייקט בקונסולה תראה את ה-`Map` במלואו.

תרגיל:

צרו Set של רשימה משתמשים שמכילה את המשתמשים `moshe`, `yaakov` ו-`itzhak`. הציגו גם את הגודל שלו.

פתרון:

```
const names = new Set();
names.add('moshe');
names.add('yaakov');
names.add('itzhak');
console.log(names.size); // 3
```

הסבר:

יצירת ה-`Set` נעשית בעזרת פונקציה בנית מסוג `Set`. מכניסים את ה-`Set` החדש לתוכו. זה לא אומר שאי-אפשר להכניס דברים ל-`Set` או למחוק דברים מתוכו, אבל אי-אפשר לשנות את `names` לשוג אחר. קל ונחמד. משתמשים במתודת `add`, שקיימת בכל `Set` להכנסתערכים בלבד. אתם זוכרים שב-`Set` יש אר וركערכים ולא מפתחות? זו הייחודיות של `Set` לעומת `Map`. הציגת מספר הערכים נעשית באמצעות תכונת `size`. זה הכל.

פרק 17

תכנות אסינכרוני - הולבחים



תכנות אסינכרוני – קולבקים

כדי להבין מה זה תכנות אסינכרוני, כדאי להבין מה זו אסינכרוניות. כשחובבים על תוכנה, בעצם חובבים על תוכנה סינכרונית. למשל התוכנה הבאה:

1. לטען את מערך A.
2. לטען את מערך B.
3. למיין את מערך A.
4. למיין את מערך B.

מה קורה אם שלב 1 ושלב 2 (טעינת המרכיבים) הם ארוכים? בכל הדוגמאות, מנווע הג'אווהסקריפט פשוט חיכה. שלב 1 מתבצע, אחרי זה שלב 2, אחרי שלב 3 ואז שלב 4. תכנות אסינכרוני עובד קצת אחרת. בעצם כותבים את התוכנה לפי שלבים. הרי אין היגיון לחכות לטעינת מערך B אם מערך A כבר נמצא בזיכרון ואפשר למיין אותו בזמן שמערך B נטען.

תוכנה אסינכרונית נראה כך:

1. טוענים את מערך A. כשהוא נטען (אם אין פעולה אחרת שמתבצעת) – ממיינים אותו.
2. טוענים את מערך B. כשהוא נטען (אם אין פעולה אחרת שמתבצעת) – ממיינים אותו.

קל לראות שהתהליך ייעיל בהרבה. טוענים את מערך A ואת מערך B. ברגע שהראשון נטען, ממיינים אותו. זה לוקח הרבה פחות זמן. לא מוצאים כלום במקביל – אבל בזמן שמתיניכים למשהו שיטען, יכולים לעשות דברים אחרים. כדאי לשים לב שזה שונה מפעולות דברים במקביל.

תכנות אסינכרוני הוא אחד מהפתרונות החזקים של הפלטפורמות השונות שMRI צוות ג'אווהסקריפט (מדפסן ועד שרת), וג'אווהסקריפט תומכת באסינכרוניות באופן מושלם, בטח ובטח בשימושים מודרניים. כשכתבים סקריפט, יש המון פועלות שMRI צוות והתוכאה שלhn מגיעה מאוחר יותר. הבה נציגים זאת ונשתמש שוב בדוגמה של תוכנה פשוטה של המרת מטבחות. הסקריפט פשוט הזה מקבל קלט מהמשתמש (למשל סוג המטבח וכמותו), ניגש לשירות חיצוני כלשהו באמצעות האינטרנט, על מנת לקבל את שער ההמרה, מכפיל את שער ההמרה בכמות המטבח ומציג את התוצאה למשתמש.

סדר פעולה	מה קורה בפעולה
1	קבלת קלט מהמשתמש – שם המטבח וכמותו
2	גיישה לשרת חיצוני כדי לקבל את שער ההמרה
3	קבלת הנתן מהשרת החיצוני, חישוב הסכום והציגו למשתמש
4	מוכנים לקלט נוספת מהמשתמש

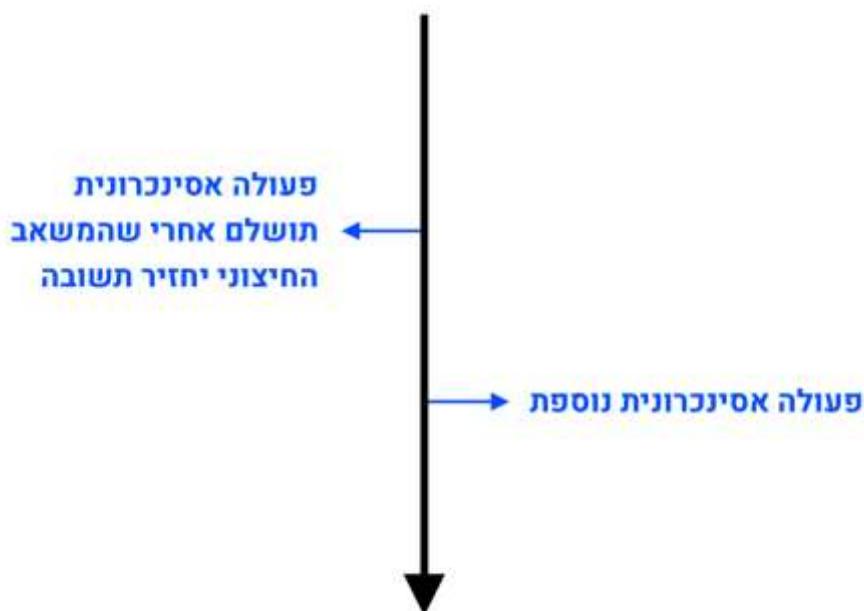
פעולה 2 דורשת זמן. בעוד פעולה 1 ופעולה 3 הן מיידיות, עבור פעולה 2 צריכים לבצע קריאה לשרת מרוחק ולחכות עד שתגיע תשובה. גם אם השרת המרוחק הוא מאד-מאוד מהיר והרשת של המשתמש סופר-יעילה, עדין הפעולה הזאת תיקח כמה מילישניות לפחות, ואז יש בעיה. אי-אפשר לעבור לשלב 4. האם עוזרים את פעולה הסקריפט והמשתמש "געגע" או שמשיכים את פעולה הסקריפט לדברים אחרים? למשל, המשתמש אולי רוצה גם לבצע המرة לשער אחר או להפעיל סקריפט אחר.

יש שתי אפשרויות:

האפשרות הסינכרונית – כל שלב נועל את השלב הבא. מ-1 עוברים ל-2. עד ש-2 לא מושלם לא עוברים ל-3 ובטח ובטח שלא עוברים ל-4. זו בעצם האפשרות הקלואסית שאותה למדנו עד עכשוו. בקוד הבא:

```
console.log('1');
console.log('2');
console.log('3');
console.log('4');
```

2 יודפס אחר ורך אחרי 1 ו-3 תמיד יודפס אחרי 2. הקוד רץ לפי סדר הכתיבה/קריאה. זה נקרא סדר סינכרוני. כל פעולה בג'אווהסקריפט היא סינכרונית וקוד תמיד ירוץ מההתחלת עד הסוף. האפשרות האסינכרונית היא מעט מורכבת יותר. אם שלב 2 חינוי רק לשלב 3, אפשר לומר לסקריפט, "שמע נא, מר בחור, שלח את הקריאה לשרת. ברגע שהשרת יחזיר עム תשובה, תציג את הפלט למשתמש. אבל עד שזה יגיע, אל תחכה לי. רוץ הלהה לשלב 4 ולשלבים שאחריו". כלומר הקוד האסינכרוני לא חוסם ולא בולם את המשך הרצת הסקריפט. הוא כן חוסם את מה שציר.



קוד אסינכרוני מופיע סקריפטים מודרניים. באפליקציות או באתרים כיום יש המונ פועלות כאלה. למשל, רישום לדיסק קשיח, קריאה למשאב חיצוני, הפעלת מצלמת ידיאו או מיקרופון. במקום

לנעלם את המשטמש,אפשרים לו לעبور הלהה. בדיקן כמו שבגוגל דוקס, למשל, השמירה נישית ברקע ואפשר לכתוב מסמך בלי הפרעה או "געילה". בדיקן כפי שבפייסבוק, לדוגמה, אם רוצים ליזום שיחת וידיאו, עד שהמשטמש השני מקבל את הבקשה אפשר לגלו בפיד או לעשות לייק. כל מה שכתבתי לעיל נכתב בג'אווהסקריפט ומשתמש בקוד אסינכריוני.

בתיאוריה זה טוב ויפה, אבל איך מתרגמים את זה למציאות? יש כמה שיטות ליישום אסינכראניות. השיטה הראשונה היא שיטת הקולבקים. זו שיטה שקל ללמידה ולהבין, ולמדנו עליה בפרק על הפונקציות. בואו נחזור ונסביר מה זה קולבק.

על מנת להמחיש את השיטה ניצור הדמיה של פונקציה שקורסית לשירות. הפונקציה הזו תשתמש ב-setTimeout, שאוטו כבר הכרנו:

```
function getService() {
  setTimeout(() => {
    return 'Result from remote service';
  }, 1);
}
const answer = getService();
console.log(answer); // undefined
```

פונקציית `getService` היא פונקציה פשוטה. בתוכה יש `setTimeout`. מה שהוא עושה זה להחזיר תשובה עם עיקוב של מילישניהם אחת. אפשר להעמיד פנים שהתשובה הזו הגיע משרת מרוחק. בפונקציה אמיתית, לא יהיה `setTimeout` אלא תהיה קריאה אחרת, אבל ככל מקרה הקריאה הזו תיקח זמן. אפילו מילישניהם אחת (שזה מעולה) היא עדין זמן.

אם מנסים לקרוא לפונקציה הזו, מקבלים `undefined`. למה? כי הפונקציה עדין לא רצתה בכלל, אף על פי שמדובר בעיקוב של מילישניהם אחת בלבד!

```
2 function getService() {
  setTimeout(() => {
    return 'Result from remote service';
  }, 1);
}
1 const answer = getService();
3 console.log(answer); // undefined
```

לא מספיק לחזור

כמובן, שנמצאים בשלב 3, בשלב 2 עדין לא הספיק להתרחש.

יש דרכים סינכרוניות לטפל בכך, אבל הנה נטפל בכך באופן אסינכרוני. אם רוצים בשלב 3 יתרחש רק לאחר שלב 2 יושלם, הדרך לעשות זאת היא להכניס את שלב 3 לפונקציה ולגרום לפונקציה בשלב 2 לקרוא לה באופן אוטומטי. הדרך הזו נקראת "קולבק".

איך גורמים לפונקציה `getService` להריץ פונקציה אחרת? משבירים לה ארגומנט שהוא בעצם פונקציה וקוראים לפונקציה זו עם מה שמגיע מהשירות המרוחק:

```
function getService(cb) {
  setTimeout(() => {
    cb('Result from remote service');
  }, 1);
}
```

למדנו את זה לעומק בפרק על הפונקציות, אבל ארכיטיב שוב. ארגומנטים יכולים להיות פונקציות. בג'אווהסקריפט פונקציות הן אובייקטים. כמו שאפשר להעביר מחרוזת טקסט, מספר, מערך וכל דבר אחר, אפשר להעביר פונקציה. ברגע שהוא מושתנה, קוראים לה כמו פונקציה רגילה ומשבירים לה איזה ארגומנט שורצים. במקרה זהה, הארגומנט הוא סתם מחרוזת טקסט מומצאת. בקריאה לשרת הוא יכול להיות מחרוזת טקסט משמעותית או אובייקט מייד.

הפונקציה מקבלת ארגומנט של פונקציה וקוראת לו אך ורק כשהפעולה מסתיימת, ובמקרה הזה, אחרי מילישנייה אחת:

```
function getService(cb) {
  setTimeout(() => {
    cb('Result from remote service');
  }, 1);
}
```

ה콜בק מופעל רק בסוף הפעולה

עכשו מפעילים את הפונקציה. צריך רק לקרוא ל-`getService`, כשהארוגמנט הוא הפונקציה שורצים שתעבד מיד לאחר שהפעולה של `getService` תושלם. משבירים ארגומנט פונקציה אונונימית מסווג חוץ שתדפיס את מה שמעבירים אליה, זה הכל. שימוש לב שה-`cb` זה הארגומנט שאותו משבירים והוא אמור להיות פונקציה, כמובן:

```
function getService(cb) {
  setTimeout(() => {
    cb('Result from remote service');
  }, 1);
}
getService(
  (answer) => {
    console.log(answer)
  }
);
```

```

function getService(cb) {
  setTimeout(() => {
    2 cb('Result from remote service');
  }, 1);
}

1 getService(
  3 (answer) => {
    console.log(answer)
  }
);

```

הfonקציית האנוונימית
שאנו מעבירים נקראת
בסיום הפעולה

ככה עובד קוד אסינכרוני עם קולבקים. הפעולה מתחילה כשייצאים ממחוזות הדוגמה אל מחוזות המוצאות. נניח שיש אפליקציה שבה הלקוח מכניס מספר בשקלים ובודק כמה מנויות של חברה זרה הוא יכול לKNות. צריך לעשות את הפעולות הבאות:

1. לקבל את הקלט מהלקוח – מחיר בשקלים וסוג מניה.
2. לראות מה שער הדולר באמצעות קרייה לשירות חיצוני ולבדק כמה דולרים השקלים האלו שווים.
3. לקבל את מחיר המניה באמצעות שירות חיצוני אחר וראות כמה מנויות אפשר לKNות.
4. להציג את המידע ללקוח.

כלומר, אם הלקוח רוצה לרכוש ב-40 שקל מנויות של חברת אינטל, קוראים לשירות מסויים שיראה מה שער הדולר. נניח שהוא 4, וללקוח יש 10 דולרים. אחרי כן קוראים לשירות אחר שיראה את מחיר המניה, נניח שהוא 5 דולרים. הפלט שיוצג ללקוח הוא 2 מנויות. 40 לחלק ל-40:4:5, או בתרגילים: 40:4:5.

איך ממשים את זה? הינה השירותים שمدמים קרייה לשער הדולר וקרייה למחיר המניה:

```
function getCurrencyRate(cb) {
    setTimeout(() => {
        cb(4);
    }, 1000);
}

function getStockPrice(stockSymbol, cb) {
    setTimeout(() => {
        cb(5);
    }, 1000);
}
```

השירות המזוייף הראשון `getCurrencyRate` דומה לזה שתרגלו נו – הוא קורא לקובלבק שמעבירים לו ומחזיר 4. השירות המזוייף השני הוא `getStockPrice`. מעבירים לו שני ארגומנטים – הארגומנט הראשון הוא סימן המניה והשני הוא קובלבק. השירות המזוייף לא משתמש בסימן המניה, אבל הוא כן קורא לקובלבק עם 5:

```
function getCurrencyRate(cb) {
    setTimeout(() => {
        cb(4);
    }, 1000);
}

function getStockPrice(cb) {
    setTimeout(() => {
        cb(5);
    }, 1000);
}

const amount = 40;

getCurrencyRate(
    (rate) => {
        getStockPrice((price) => {
            const finalResult = amount / rate / price;
            console.log(finalResult); // 2
        });
    }
);
```

از מבצעים קרייה ל-`getCurrencyRate` כשמיירים פונקציית חץ כארוגמנט. פונקציית החץ מקבל את שער הדולר מהשירות המזוייף ותקרא מיד לשירות שמחזיר את מחיר המניה. שער הדולר ומחיר המניה יסיעו לחשב כמה מנויות אפשר לרכוש. במקרה זה – שתיים. אפשר לראות שהמבנה זה מסובך למדי. קובלבק בתוך קובלבק.

```

function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}

function getStockPrice(cb) {
  setTimeout(() => {
    cb(5);
  }, 1000);
}

const amount = 40;

getCurrencyRate(
  (rate) => {
    getStockPrice((price) => {
      const finalResult = amount / rate / price;
      console.log(finalResult); // 2
    });
  }
);

```

קולבק ראשון

קולבק שני

המצב הזה עוד יכול להסתבר ככל שהסקרייפט מורכב יותר, וاز בהחלט אפשר לראות שלושה וארבעה קולבקים ואףילו יותר מקננים זה בתוך זה. מצב זה נקרא **"גיהינום הקולבקים"** (callback hell). איך פותרים את זה? בעזרת promises, כפי שנלמד בפרק הבא.

תרגיל:

נתונה פונקציית `getCurrencyRate`, שמדמה החזרה של שער הדולר באמצעות שרת חיצוני.

```
function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}
```

קראו לפונקציה זו וחזירו את התוצאה שלה בקונסולה.

פתרונות:

```
getCurrencyRate(
  (answer) => {
    console.log(answer)
  }
);
```

הסביר:

קוראים לפונקציה `getCurrencyRate` עם ארגומנט אחד. איזה ארגומנט? פונקציה אוניבימית מסוג `חץ` שמקבלת ארגומנט. הארגומנט זהה יודפס. איך זה עובד בפועל? פונקציית `getCurrencyRate` מcepts לקלל ארגומנט שהוא קולבך. היא מדמה קרייה לשרת ומפעילה את הקולבך עם ארגומנט שהוא התשובה. הפונקציה האוניבימית שיצרתם מקבלת את התשובה ומחזירה אותה.

תרגיל:

צרו שירות מזויף המדמֶה קרייה לדיסק הקשיח וקריאת קובץ. השירות המזויף יקבל שם קובץ וקובלבק. הוא תמיד יחזיר טקסט שבו כתוב "this is from filename", שם הקובץ יהיה השם שמועבר לו. קראו לשירות המזויף והדפיסו את התוצאה בקונסולה.

פתרונות:

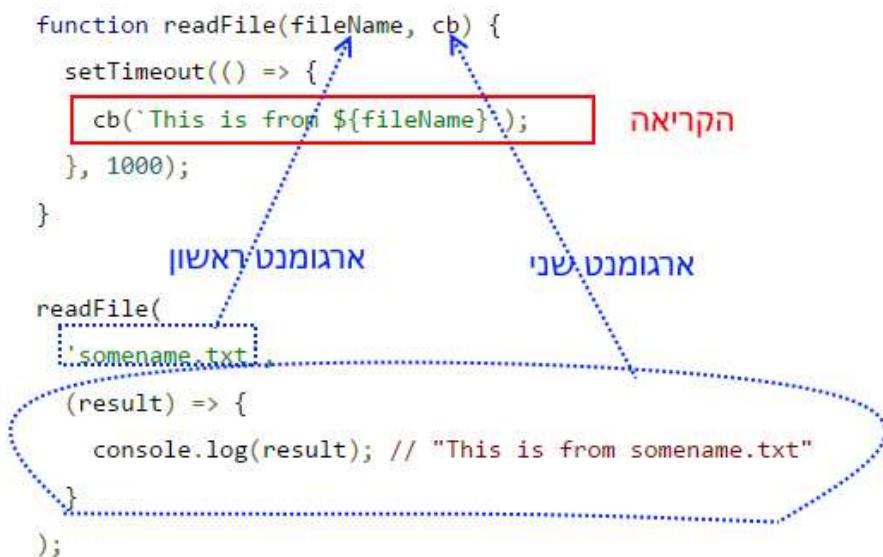
```
function readFile(fileName, cb) {
  setTimeout(() => {
    cb(`This is from ${fileName}`);
  }, 1000);
}

readFile(
  'somename.txt',
  (result) => {
    console.log(result); // "This is from somename.txt"
  }
);
```

הסביר:

השירות `readFile` הוא שירות מזויף מהסוג שכבר יצרנו. הוא מקבל שני ארגומנטים, שם הקובץ וקובלבק. את הקובלבק הוא מפעיל לאחר עיכוב של שנייה ומשתמש בשם הקובץ כדי להעביר אותו כארוגמנט לקובלבק.

עכשו רק צריך לקרוא לשירות ולהעביר אליו שם קובץ מזויף ופונקציית חץ אונומית שתופעל בסיום הפעולה. בפונקציית החץ האונומית רק מבצעים הדפסה של התוצאה:



Promises

כפי שראינו, קולבקים הם שיטה ייעילה מאוד לעבוד עם קוד אסינכרוני, אבל יש להם חסרונות, ובראשם הסרבלן הגדול מאוד של הקוד או "גיהינום הקולבקים", וגם חסרונות אחרים — כמו למשל חוסר היכולת לתפוא שגיאות (את עניין ה-try-catch למדנו באחד הפרויקים הקודמים). כדי לנהל את השגיאות קיימים פרומיסים (promises, "הבטחות" באנגלית).

העיקרון מאחורים הוא פשוט: פונקציה אסינכרונית מחזירה "הבטחה" שהיא יכולה להפר או לKEEP. אם למשל יש פונקציה שקוראת לשירות חיצוני, היא יכולה לKEEP את הבטחה בכך שתחזיר את התוצאה מהשירות המרוחק או להפר אותה אם השירות המרוחק נכשל. בקוד שקורא לפונקציה אפשר להחליט מה קורה אם הבטחה מתקיימת ומה קורה אם לא.

איך יוצרים promises? הנה נשתמש בשירות המזוייף מהפרק הקודם. השירות מחייב שנייה ואז מוחזר 4. הוא מדמה שירות שמחשב שערי מטבעות וקורא לשרת חיצוני. הקוד הוא פשוט מאוד:

```
setTimeout(() => {
  return 4; // Return 4
}, 1000);
```

ובן שזו יכולה להיות קרייה לשרת אחר, קרייה למאגר נתונים או לכל מקום שהוא. בסופו של דבר, מדובר בפקודה שמקבלת קולבק שפועל אחרי 1,000 מילישניות. למדנו על כך בפרק על setTimeout ועם בפרק הקודם.

מה שעושים הוא ליצור אובייקט הבטחה בעזרת פונקציה בנית. האובייקט מקבל ארגומנט אחד שהוא קולבק, שיש לו שני ארגומנטים — פונקציית resolve, שלה קוראים כשההבטחה מצלילה, ופונקציית reject, שלה קוראים אם רציתם שההבטחה תיכשל. יצירת ההבטחה נראה כך:

```
let myPromise = new Promise((resolve, reject) {
});
```

כאן ממש אפשר לראות את יצרת ה-Promise במציאות פונקציה בנית שמקבלת כארגומנט פונקציית חץ אונומית, שלה יש שני ארגומנטים שהם בעצם קולבקים.

כל מה שנשאר לעשות הוא להכניס את השירות המזוייף ולקרא ל-`resolve` שבתוכו. כאן רוצים למש את הבדיקה עם 4:

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

אפשר לראות שהפונקציה הבנאית מקבלת פונקציית חץ אונומית שיש לה שני ארגומנטים – `reject` ו-`resolve`. מדובר בשני קולבקים שאפשר להפעיל. הראשון הוא בעצם הפעלה במקרה של הצלחה, וזה בדיק מה שעושים בשירות המזוייף.

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

השירות המזוייף מחייב שנייה אחת (1,000 מילישניות) וaz קורא ל-`resolve` ו מעביר את הארגומנט 4. הכל אrox בתוך פונקציית מחזירה הבדיקה. אפשר לקרוא ל-`reject` לפונקציה זו ולקבל את הבדיקה.

מה עושים עם הבדיקה? אפשר בעצם לבוא ולומר, "תשמעי, הבדיקה, ברגע שתתמלאי, תריצי את הקוד זהה". איך עושים את זה? לכל הבדיקה יש מתודת `then` שמקבלת כารוגומנט ראשון פונקציה שモפעלת כשההבדיקה מתממשת:

```
readFile().then((result) => { console.log(result); })
```

הקוד הזה רץ בכל פעם שההבדיקה הנמצאת ב-`readFile` מתממשת. בואו נמחיש את זה באמצעות דוגמה נוספת, הפעם אפיו בלי `setTimeout`: יוצרים פונקציה שמחזירה הבדיקה. שמה `readFile`, והוא לא תקבל שום ארגומנט. מה שהוא עושה הוא ליצור הבדיקה. בתוך הבדיקה, במקומ לקרוא לשירות זה או אחר, פשוט עושים `resolve` ומקיימים את הבדיקה.

כשקוראים לפונקציה, משתמשים ב-`then` ו מעבירים כารוגומנט פונקציה שמאזינה להבדיקה. ברגע שהוא תתקיים, היא תפעל. כך נראה הקוד:

```
function readFile() {
  let myPromise = new Promise(
```

```

        (resolve, reject) => {
            resolve('Promise resolved!')
        }
    );
    return myPromise;
}
readFile().then(
    (result) => {
        console.log(result);
    }
);

```

הشرطוט הבא ימחיש טוב יותר את העניין. בפונקציית `readFile` יוצרים את הבדיקה. הבדיקה מקבלת ארגומנט של פונקציית חץ שבתוכו מקיימים אותה ומחזירים תגובה. בחלק השני קוראים לפונקציה:

```

function readFile() {
    let myPromise = new Promise(
        (resolve, reject) => {
            resolve('Promise resolved! ')
        }
    );
    return myPromise;
}

```

הזרת הבדיקה למי שקורא לפונקציה הבדיקה שיצרת!

קריאה לפונקציה המחזירה הבדיקה
הפונקציה הראונה המועברת בארגומנט תקרא
כasher הבדיקה תתקיים

ירוץ כasher הבדיקה תתקיים

מה שחשוב להבין הוא שיש לשימוש בהבדיקות שני חלקים – החלק הראשון הוא ייצירת הפונקציה שבה יש את הבדיקה וכתיבת הבדיקה, כולל החלק שבו מבצעים `resolve` או `reject` את הבדיקה. החלק השני הוא ההזנה, באמצעות `then`, לפונקציה שמחזירה הבדיקה.

כמו שאפשר לקיים הבדיקה, אפשר גם להפר אותה. למשל אם השירות מחזיר שגיאה כי השרת לא נמצא, כי מסד הנתונים נפל או מכל סיבה אחרת. הפרת הבדיקה נעשית באמצעות `reject`. זה נראה כך:

```

function waitTwoSeconds() {
    let secondsPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('ERROR! ERROR!');
        }, 2000);
    });
}

```

```

    });
    return secondsPromise;
}
waitTwoSeconds().then(
  (result) => { console.log(result) }, // Will never happen
  (error) => { console.log("Bad error:" + error) } // "Bad
error:ERROR! ERROR!"
);

```

כאן יש את ה-setTimeout שראינו קודם, אבל במקום לו קיים את ההבטחה אחרי שתי שניות, מפרים אותה בgesot. שימו לב שהכל נראה אותו דבר: בניית הפונקציה שעתופת את ההבטחה והשימוש ב-setTimeout. השוני העיקרי כאן הוא השימוש ב-reject. כך מפרים את ההבטחה. גם פה אפשר לשלוח ארגומנט שייתפס בפונקציה שקולתת את ה-reject.

הפונקציה שתופסת את הפרת ההבטחה נכנסת כารוגומנט השני של פונקציית then. העריה: בקוד מורכב מומלץ לעשות reject עם אובייקט שגיאה מסווג, אחרת לא מקבלים לוג שגיאות מסווג וזה עלול לבלבל.

```

function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('ERROR! ERROR!');
    }, 2000);
  });
  return secondsPromise;
}

```

הפונקציה הראשונה תרוץ במקרה וההבטחה תקיים

```

waitTwoSeconds().then(
  (result) => {console.log(result)}, // Will never happen
  (error) => {console.log("Bad error:" + error)} // "Bad error:ERROR! ERROR!"
);


הפונקציה השני תרוץ במקרה וההבטחה תופר


```

הפונקציה הראשונה מתממשת רק כשההבטחה מקיימת. במקרה זה – זה לעולם לא יקרה. הפונקציה השנייה היא החידוש פה: מعتبرים אותה כารוגומנט השני של ה-then והוא תפעל אך ורק כאשר ההבטחה תופר. במקרה זה היא מדפיסה את מה שהוא מקבלת בקונסולה בתוספת המילים Bad error. כיוון שההבטחה מחייבים את המילים ERROR!, ERROR!, מקבלים בקונסולה את המשפט:

"Bad error: ERROR! ERROR!"

בשימוש בהבטחות צריך תמיד לזכור לסגור את כל המקרים, כדי שלא יקרה שהקוד פשוט יפסיק לרוץ מפני שכחיתם להשתמש ב-`reject` באחד ממשפטיו התנאי.

שרשור הבטחות

אחד הפיצ'רים החזקים ביותר בהבטחות הוא יכולת לשרשור אותן — כמו `then` יכול להחזיר `promise` שיתפס על ידי `then` המשורשר אליו. ראשית נשאלת השאלה — למה צריך את זה? התשובה פשוטה: פעמים רבות פונקים לשרת, ואת התגובה שמקבלים ממנה שולחים לשרת אחר או ששמורים במסד הנתונים או כל דבר אחר. שרשור הבטחות או פונקציות אסינכרוניות הוא נפוץ מאד.

בדוגמה הבאה נראה איך עושים את זה. יש שתי פונקציות שemmמשות הבטחה. אחת היא `writeNumber` ש תמיד מקיימת את הבטחה ומחזירה 100, והאחרת היא `getNumber` שבודקת את המספר. אם המספר הוא 100, היא מקיימת את הבטחה ומחזירה OK. ניתן לראות את התלות — הפונקציה `writeNumber` תלוי בפונקציית `getNumber` שתחזיר לה את המספר. הן מממשות כמו כל הבטחה שכבר ראיינו, אבל מה שמיוחד בשרשור הבטחות הוא מי שקורא לפונקציות הללו:

```
function getNumber() {
  let myPromise = new Promise((resolve, reject) => {
    resolve(100);
  });
  return myPromise;
}

function writeNumber(number) {
  let myPromise = new Promise((resolve, reject) => {
    if (number === 100) {
      resolve('OK');
    }
  });
  return myPromise;
}

getNumber()
  .then(
    (number) => { return writeNumber(number); }
  )
  .then(
    (result) => { console.log(result) }
  )
}
```

אפשר לראות שemmמשים את הקריאה ל-`getNumber` כרגע ומשתמשים ב-`then` כדי לתפוס את הקריאה הראשונה. מה שמיוחדפה הוא שלוקחים את התוצאה של קיום הבטחה הראשונה (שתהיה מן הסתם 100) ובמקומ להחזיר אותה לפונקציה, מבצעים קרייה של הפונקציה השנייה עם התוצאה (ש כאמור היא 100) ומהזירים אותה! זו הבטחה לכל דבר עניין. אפשר לתפוס אותה עם `then`, שם מדפיסים את התוצאה, מן הסתם היא OK. אממש זאת בעזרת דוגמה נוספת:

```

function myPromise() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise resolved');
  });
  return promise;
}
myPromise().then((data) => {
  return data + ' 1 ';
})
  .then((data) => {
    return data + ' 2 ';
})
  .then((data) => {
    console.log(data); // promise resolved 1 2
  });

```

```

function myPromise() {
  let promise = new Promise(function(resolve, reject) {
    resolve('promise resolved');
  });
  return promise;
}

myPromise().then((data) => {
  return data + ' 1 ';
})
  .then((data) => {
    return data + ' 2 ';
})
  .then((data) => {
    console.log(data); //promise resolved 1 2
  });

```

אפשר לראות איך `then` מוחזיר תמיד הבטחה, גם אם לא קוראים לפונקציה שמחזירה את הבטחה יותר מפעם אחת.

ואיך תופסים בשיטה זו הבטחה שנכשלת? בעזרת מתודת `catch` שמקבלת לתוכה את מה ששלוחים ב-`reject`, בדיק כmo במתודה השנייה ב-`then` הרגיל שלמדנו עליי בסעיף הקודם. הינה דוגמה ל-`catch`:

```
function myPromise() {
  let promise = new Promise(function (resolve, reject) {
    reject('promise rejected!');
  });
  return promise;
}
myPromise().then((data) => {
  return data + ' 1 ';
})
  .then((data) => {
    return data + ' 2 ';
})
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.log(error); // promise rejected!
  });
}
```

הדוגמה הזאת לדוגמה הקודמת למעט שני דברים: נוסף catch שתפקידו ומדפיס כל error ואת ה-promise מוחזרים כ-reject. הוספת ה-catch-ה-allowת מאפשרת שגיאות בכלל רגע נתון כמשמעותם then.

קיבוץ הבטחות

אם יש כמה הבטחות שאין תלויות זו בזו ורוצים לשלוח את כלן יחד ולקבל את התוצאות של כלן בזורה מסודרת, אפשר להשתמש ב-`Promise.all`. מדובר בפונקציה גלובלית של האובייקט `Promise` שמקבלת מערך של הבטחות. אפשר לשרר לה `then` שיטף במצב שבו כלן עוברות או שאחת מהן לפחות נכשלה, אף על פי שסדר השילוח וההפעלה אינם מובטחים.

כלומר אם יש שלוש הבטחות, אפשר להאזין לקיום של שלושתן בביטחון. אםColon עבורות, הפעונקציה הראשונה ב-then מופעלת. אם אחת נכשלה, הפעונקציה השנייה מופעלת, ממש כמו בהזנה להבטחה בודדת, ואין זה משנה אם שאר הבטחות קיימו או לא:

```
function myPromiseA() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 1 resolved');
  });
  return promise;
}

function myPromiseB() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 2 resolved');
  });
  return promise;
}

function myPromiseC() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 3 resolved');
  });
  return promise;
}

Promise.all([myPromiseA(), myPromiseB(), myPromiseC()]).then(
  (results) => { console.log(results) }
  // ["promise # 1 resolved", "promise # 2 resolved", "promise # 3 resolved"]
)
```

כאן נוצרו שלוש פונקציות שמחזירות הבטחה פשוטה. יוצרים מערך של שלוש הבטחות שלhn. שימושו לבודם מערך קוראים לפונקציות כדי לקבל את הבטחות שלhn. המערך הוא של הבטחות ולא של פונקציות. את המערך מכנים allPromise.all ואפשר להשתמש עכשו ב-then. כיוון שיש כמה הבטחות וכמה תוצאות לקיום הבטחות, הכו נכנס לערך של results. זה השוני היחיד.

קיבוץ הבטחות מתקדם

הבעיה ב-`Promise.all` היא שאם הבטחה אחת נכשלת, כל הבטחות נכשלות. ה-`Promise.all` מחזיר שגיאה.

```
function myPromiseA() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 1 resolved');
  });
  return promise;
}

function myPromiseB() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 2 resolved');
  });
  return promise;
}

function myPromiseC() {
  let promise = new Promise(function (resolve, reject) {
    reject('promise # 3 resolved');
  });
  return promise;
}

Promise.all([myPromiseA(), myPromiseB(), myPromiseC()]).then(
  (results) => { console.log(results) } // Uncaught (in promise)
promise # 3 resolved
```

אם אני רוצה שלא יהיה כשלון, גם אם הבטחה אחת, שתיים או יותר לא מתקיימות, אני משתמש בפקודת `allSettled` שזאה להלוטין ל-`all` או אף בניגוד לו, היא מחזירה דיווח מלא על כל הבטחות – בין אם יש כשלון ובין אם הבטחה מתקיימת. ניתן לראות את זה בקוווט:

אם תיקחו את הדוגמה הקודמת ותחליפו בין `all` ל-`allSettled`:

```

function myPromiseA() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 1 resolved');
  });
  return promise;
}

function myPromiseB() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 2 resolved');
  });
  return promise;
}

function myPromiseC() {
  let promise = new Promise(function (resolve, reject) {
    reject('promise # 3 resolved');
  });
  return promise;
}

Promise.allSettled([myPromiseA(), myPromiseB(), myPromiseC()]).then(
  (results) => { console.log(results) }
  /*[{"status": "fulfilled", value: "promise # 1 resolved"},  

   {"status": "fulfilled", value: "promise # 2 resolved"},  

   {"status": "rejected", reason: "promise # 3 resolved"}, */  

)

```

ישנו עוד פקודות של קיבוץ הבטחות. `any` היא פקודה שמחזירה מ בין שורת ההבטחות את ההבטחה הראשונה שמתקיימת ואלה בלבד. `race` היא פקודה המחזירה את ההבטחה הראשונה שמתקיימת או נכשלת.

תרגילים:

צרו שירות, באמצעות `setTimeout`, שמחזיר לאחר שתי שניות תגובה עם המילים `Two seconds passed`. הכניסו אותו להבטחה והכניסו את הבטחה לתוכה פונקציה שתחזיר אותה.

בדקו שהפונקציה עובדת באמצעות האזנה עם `then` לקיום הבטחה.

פתרונות:

```

function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Two seconds passed!');
    })
  });
  return secondsPromise;
}

```

```

        }, 2000);
    });
    return secondsPromise;
}
waitTwoSeconds().then((result) => { console.log(result) });

```

הסביר:

החלק הראשון הוא ייצרת ה-`setTimeout`, שמקבל שני ארגומנטים. הראשון הוא מה שיש לעשות אחרי פרק הזמן שהוגדר לו והשני הוא פרק הזמן במלישיות. 2,000 מילישניות הן שתי שניות.

החלק השני הוא ייצרת פונקציה שמחזירה הבטחה. יוצרים פונקציה בשם `secondsPromise` בתוכה יש הבטחה בשם `secondsPromise`.

החלק האחרון והקשה ביותר הוא ליזוק תוכן בתוך הבטחה. הבטחה נוצרת באמצעות פונקציה בנאיית של `Promise`. היא מקבלת ארגומנט אחד שהוא פונקציית קולבק עם שני ארגומנטים, `resolve` ו-`reject`. בתוך הקולבק מכנים את ה-`setTimeout` וקובעים שההבטחה תקיים ברגע שתי השניות יעברו. הבדיקה פשוטה יותר. קוראים לפונקציה ואז מażינים עם `then`. היא מקבלת ארגומנט אחד, פונקציה אחת שפועלת ברגע שההבטחה מתקינה. כל מה שצריך לעשות הוא להזין למה שמחזיר ב-`result` ולהדפיס אותו בקונסולה:

```

function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Two seconds passed!');
    }, 2000);
  });
  return secondsPromise;
}
waitTwoSeconds().then((result) => { console.log(result) });

```

תרגיל:

צרו פונקציה `checkIfNumberOK` שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזיר הבטחה מקוימת עם מחרוזת הטקסט `OK`. אם המספר קטן מ-10 או שווה לו, תחזיר הבטחה מופרת עם מחרוזת הטקסט `bad Number`.

פתרונות:

```
function checkIfNumberOK(number) {
  let myPromise = new Promise((resolve, reject) => {
    if (number > 10) {
      resolve('Number is OK');
    } else {
      reject('Bad Number');
    }
  });
  return myPromise;
}
checkIfNumberOK(11).then(
  (result) => { console.log(result) }, // Number is OK
  (error) => { console.log('error: ' + error) } // Will never happen
);
checkIfNumberOK(5).then(
  (result) => { console.log(result) }, // Will never happen
  (error) => { console.log('error: ' + error) } // error: Bad Number
);
```

הסבר:

הfonקציה `checkIfNumberOK` היא פונקציה רגילה המקבלת מספר כารוגמנט. בתוך הפונקציה יוצרים הבטחה כאובייקט `Promise` עם הפונקציה הבנאית. הפונקציה מקבלת ארוגמנט שהוא פונקציה אונימית מסוג `cz`, ובה יש שני ארוגמנטים, `reject` ו-`resolve`. משתמשים בשנייהם. מפעילים משפט תנאי פשוט ובודקם: אם המספר גדול מ-10, ההבטחה תקיים באמצעות `resolve`, ואם המספר קטן מ-10 היא תופר באמצעות `reject` - כמובן עם הטקסטים הרלוונטיים.

הבדיקות יי'ו פשוטות למדי – הריצה של הפונקציה עם המספר 11 ותפישת הצלחה או השגיאה באמצעות `then`. כיוון ש-`checkIfNumberOK` מוחזירה הבטחה, אפשר להשתמש ב-`then` שמקבל שני ארוגמנטים, ארוגמנט של הפרה וארוגמנט של הצלחה. מן הסתם, כשמעבירים 11, ההבטחה מקוימת והfonקציה הראשונה שמעבירים ב-`then` תופעל. כשמעבירים מספר קטן יותר מ-10, ההבטחה מופרת והfonקציה השנייה שמעבירים ב-`then` תופעל.

פונקציית `async`

השיטה השלישית לתוכנות אסינכרוני לא שונה כל כך מהשיטה של הבטחה. למען האמת מדובר בהרבהה שלה. פונקציית `async` מאפשרת לטפל בהבטחות ללא ה-`then` או ליתר דיווק מאפשרת לכתוב קוד אסינכרוני בצורה סינכרונית. אם יש שירות שמחזיר `promise`, אפשר

לבחור אם לubit עם then הקודם או בפונקציה אסינכרונית עם המילה השמורה .async

לשם הדגמה משתמש בדוגמה של הקריאה המזויפת שהשתמשה בה קודם. הקריאה זו לא שונה כל ממה שלמדו בפרק הקודם:

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

מדובר בקריאה שמחזירה תשובה מזויפת של 4 אחרי שנייה אחת. שימוש ליב שהוא מחזירה promise. אם היינו רוצים להתחבר אליה, היינו צריכים לקרוא לה וזו משתמש ב-then כדי לתפוא את ההצלחה. אבל אפשר לעשות את זה עם `chuny` באופן פשוט. ראשית מגדירים פונקציה אסינכרונית, כלומר הפונקציה זו מרים קוד שיכל לזרע בזמן אחר. ההגדרה נעשית באמצעות המילה השמורה `chuny`. בתוך הפונקציה זו אפשר לקרוא לפונקציות המוחזקות במאזעות הקריאה `as`. פונקציות המוחזקות הבטחה יקבלו `await` ליד הקריאה שלהן, והסקריפט בתוכה כפי שרצים. פונקציות המוחזקות הבטחה יקחנה `await` ליד הקריאה שלהן, והסקריפט בתוכה `as` יעצור וימתין להן. קוד מוחזק לפונקציה יוץ כרגע. כך מרווחים אסינכרוניתות אבל עם קוד ברור יותר. למשל:

```
async function main() {
  let result = await readFile();
  console.log('result' + result);
}
main();
```

בדוגמה יוצרים פונקציה שמודרת אסינכרונית. שימוש ליב למילה השמורה `async` לפני `main` `function`. בתוך הפונקציה זו, שיש בה `chuny`, אפשר להשתמש במילה השמורה `await` שקוראת לפונקציה המוחזיקה `promise`. שימוש ליב ש-`readFile` לא שונה מפונקציה אחרת שמחזירה הבטחה, בדיק כפי שלמדו בפרק הקודם. פונקציות `chuny` פשוטណו לניהל את הbettוחות, רק ללא ה-`then`.

```

function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}

async function main() {
  let result = await readFile();
  console.log('result' + result);
}

main();

```

עד שההבטחה זו לא חזרה הפונקציה לא ממשיכה

לא ייחס עד שה await לא יושלם

כפי שב-chunk יש שני ארגומנטים – אחד שמתפקידו מאשר ההבטחה מקוימת ואחד שמתפקידו מאשר ההבטחה נכשלה – גם ב-chunk אפשר לתפוס את הכשלון. איך? באמצעות try-catch. בדוגמה הבאה יוצרים שירות ש תמיד נכשל ותופס את השגיאות:

```

function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('error :(');
    }, 1000);
  });
  return myPromise;
}

async function main() {
  try {
    let result = await readFile();
  } catch (error) {
    console.log('An error occurred: ' + error); // An error occurred:
  }
}

main();

```

שירות readFile הוא שירות ש תמיד נכשל. הוא מחרכה שנייה ועושה reject. היחסום שלו הוא בדיק כמה כל שירות אחר שמחזיר הבטחה ומקיימים אותה או לא מקיימים אותה. מה שמעניין הוא שה-chunk שנקרא main זהה לדוגמה הקודמת, למעט ה-try-catch. ה-try יקרה כאשר ההבטחה תקינה (ובדוגמה זו לעולם לא) וה-catch יתקיים אם ההבטחה לא תקינה (בדוגמה

הזה תמיד כי תמיד עושים reject). מה שקרה הוא שמודפסת שגיאה בקונסולה. אפשר כמובן לקבוע כל התנהגות אחרת.

ובן שאין בעיה לקבוע כמה `await` בטור פונקציית `async` אחת. מה שצריך לזכור, הוא שבסוףו של דבר מדובר בזיכרון סופר על גבי ה-`promises`. במקרה להשתמש ב-`then` שעלול להיות קצר מסורבל, משתמשים ב-`catch`. אלו שתי דרכים שונות לעבוד עם שירות זהה שמחזיר הבטחה.

תרגיל:

בדיוק כמו בפרק הקודם, צרו פונקציה `checkIfNumberOK` שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזיר הבטחה מקוימת עם מחוזת הטקסט `Number is OK`. אם המספר קטן מ-10, תחזיר הבטחה מופרת עם מחוזת הטקסט `Bad Number`. אם המספר אמיתי הוא למש את הקריאות עם `async`. צרו פונקציית `async` שימושת השירות, וקראו לה פעם אחת באופן צזה שהוא תדף בקונסולה את מה שהשירות מחזיר אם הוא מצליח ופעם אחת באופן צזה שהוא תדף בקונסולה את מה שהשירות מחזיר אם הוא נכשל.

פתרונות:

```
function checkIfNumberOK(number) {
  let myPromise = new Promise((resolve, reject) => {
    if (number > 10) {
      resolve('Number is OK');
    } else {
      reject('Bad Number');
    }
  });
  return myPromise;
}
async function checkMyNumber(number) {
  try {
    let result = await checkIfNumberOK(number);
    console.log(`success! ${result}`);
  } catch (error) {
    console.log(`error! ${error}`);
  }
}
checkMyNumber(11); // success! Number is OK
checkMyNumber(5); // error! Bad Number
```

הסבר:

על השירות שנוצר כאן אין מה להסביר יותר מדי כי למדנו אותו בפרק הקודם. זה שירות שמקבל מספר ומחזיר הבטחה. אם המספר גדול מ-10, הוא מקיים את ההבטחה. אם לא, הוא מפרק אותה.

מה שמעניין הוא איך שקוראים לשירות: יוצרים פונקציית `async` בשם `checkMyNumber` שמקבלת מספר. בתוכה יש `try-catch`. בתוך ה-`try` קוראים לשירות ולא שוכחים להשתמש במילה השמורה `await` כדי להראות שמדובר בשירות אסינכרוני שיש לחכות לו. אם ההבטחה מקייםת, מה שיש בתוך ה-`try` רץ. אם לא, מה שיש בתוך ה-`catch` רץ. עכשו כל מה שנוצר הוא לקרוא לפונקציה `checkMyNumber` עם המספר המתאים.

תרגיל:

נתונים שלושה שירותים שונים המחזירים הבטחות:

```

function myPromiseA(number) {
  let myPromise = new Promise((resolve, reject) => {
    resolve('Promise A success!');
  });
  return myPromise;
}
function myPromiseB(number) {
  let myPromise = new Promise((resolve, reject) => {
    resolve('Promise B success!');
  });
  return myPromise;
}
function myPromiseC(number) {
  let myPromise = new Promise((resolve, reject) => {
    resolve('Promise C success!');
  });
  return myPromise;
}

```

כתבו פונקציה `chasy` הקוראת להם.

פתרונות:

```

async function callMyPromises() {
  let results = [];
  results[0] = await myPromiseA();
  results[1] = await myPromiseB();
  results[2] = await myPromiseC();
  console.log(results);
}
callMyPromises(); // ["Promise A success!", "Promise B success!",
"Promise C success!"]

```

הסביר:

ראשית יוצרים פונקציה בשם `callMyPromises`. כדי לסמך שהיא אסינכרונית משתמשים במילה השמורה `await`. ברגע שימושם במילה הזו, אפשר להשתמש ב-`await` לפני כל פונקציה שמחזירה הבטחה. שימוש כזה יבטיח שהשורה הבאה בפונקציה תחכה. כך למשל:

```

results[1] = await myPromiseB();
תרוץ רק אחרי שההבטחה של myPromiseA תتمלא.

```

```
results[0] = await myPromiseA();
```

נותר רק לקרוא לשירותים השונים לפי הסדר ולהדפיס את התוצאה. שימוש לב שזרימת הקוד מחוץ ל-`callMyPromises` מתנהלת כרגיל, אבל זה לא מאד מעניין.

פרק 18

AJAX



AJAX

ההגדירה היבשה קובעת ש-AJAX הוא ראשית תיבות של Asynchronous JavaScript and XML. בעבר זה גם היה נכון. בגדול מדובר בשם כולל לדרך לתקשר עם שרת אינטרנט באמצעות ג'אווהסקריפט ופרוטוקול HTTP. זה נשמע מעט מッチיק, כי ג'אווהסקריפט היא שפה שחיה בראשת. אבל כשוחשבים על כך לעומק, ג'אווהסקריפט לא חיה בראשת. ג'אווהסקריפט היא שפה שחיה בדף ונטענת מיד כשהדף נטען. אם התקנתם את סביבת הבדיקה שהסבירתי עליה באחד הפרקים הראשוניים, אתם יודעים שבכל פעם שמבצעים טעינת דף, קוד הג'אווהסקריפט רץ זהה. הקוד רץ עם המידע שמכניסים לו. הוא יכול ללקחת את המידע מה-DOM או משתנים שגדירים לו, אבל מקור המידע אחד – האתר שהדף טוען.

עם AJAX אפשר לגשת לכל אתר שהוא ולקחת ממנו מידע. לצורך העניין אפשר אפילו להיכנס לאתר חדש, לשאוב את ה-HTML שלו ואז, באמצעות ביטוי רגולרי (שעלוי למדנו בפרק המוקדש לביטויים כאלה), ללקחת את הcotract ולהציגו למשתמש. מה שדף או, נכון יותר, המשתמש בדף יכול לעשות, גם ג'אווהסקריפט יכול לעשות. אבל בדרך כלל לא ממשים ב-AJAX כדי לפנות לאתרים המיעדים לבני אדם, אלא כדי לפנות לכתובות אינטרנט שמחזירות אובייקטי JSON, שקל לעבוד איתם בג'אווהסקריפט. אם תפעילו את הדף ותנסו להיכנס לכתובת החזו, גם אתם תראו את אובייקט ה-JSON. כאמור, AJAX הוא בדיקון כמו דף שמשגר נתונים, אבל עושים את זה באמצעות קוד.

הבה נדגים. יש בראשת שירוטים רבים המספקים מידע, למשל אתר המספק נתונים מזג אוויר או שערים של מטבחות חוץ. אלו אתרים המספקים API, זהה ראשית התיבות של Application Interface Programming, ובעצם המשמעות היא שמי שימצא בהם תועלת הם בעיקר סקורייפטים של AJAX. אפשר למצאו רשימות של API כלו בראש בchiposh public API data sites. אחד האתרים המרכזים רשימה של שירותים כאלה נמצא בכתובת הבאה:

<https://github.com/toddmotto/public-apis>

אפשר למצוא שם עשרות אתרים המספקים API. כאן נטרגל בשירות שנקרא "בדיקות צ'אק נוריס". השירות אינו דורש הזרחות, רישום או CAB ראש אחר ובעצם כל מה שצריך לעשות הוא לשלוח בקשה אל:

<https://api.chucknorris.io/jokes/random>

כדי לקבל בדיחה רנדומלית של צ'אק נוריס.

קל לבדוק אותו! פשטן היכנסו ל קישור באמצעות דף וריגל. תוכלו לראות שאתם מקבלים ממש JSON עם מידע. יתכן שהעיצוב יטעה אתכם (בכrome התוצאה מוצגת כמחוזת טקסט) אך זה נראה כך:

```
{
  "category": null,
  "icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
  "id": "uDjkIgHpQhe7kv8xoyJm6g",
  "url": "http://api.chucknorris.io/jokes/uDjkIgHpQhe7kv8xoyJm6g",
```

```

  "value": "Chuck Norris is currently suing NBC, claiming Law and Order
are trademarked names for his left and right legs."
}

```

ובן שעובד משתמש רגיל זה קצת בעיתוי, אבל לסקריפט שמשגר בקשת AJAX זה מעולה. יוצרים בקשת AJAX אל האתר הזה ומציגים את המידע. נשאלת השאלה "אייר?" בג'אווהסקריפט המודרנית יותר מקובל להשתמש ב-`fetch`, פונקציה גלובלית שקל להשתמש בה וקיימת בסביבת הדפדפן. הפונקציה זו ממחזירה promise שהוא התוצאה. מהתוצאה בוררים את סוג הנתון שרצו ומחזירים אותו, ואיתו עושים מה שרצו, ובמקרה של שירות הבדיקות — מדפיסים את הבדיקה.

השימוש ב-`fetch`:

```
fetch('https://api.chucknorris.io/jokes/random')
```

פשוט מאד, נכון? הפונקציה זו ממחזירה promise עם התגובה מהשרת. ב-`promise` למדנו להשתמש בפרק הקודמים. עכשו צריך רק להחליט מה עושים אם ה-`promise` מתקים. במרקחה זה אנחנו רוצים את `JSON`, סוג המידע שה-`API` עבד אליו. זו הסיבה שאנו משתמשים במתודת `JSON` הקיימת לאובייקט התגובה `response`. יש סוגים מתודות שונים לסוגי מידע שונים שמתקבלים מה-`response`. אם למשל ה-`API` היה מחזיר תמונה, היה נדרש `BLOB` שהוא סוג מידע אחר שלא נדון בו כאן:

```
fetch('https://api.chucknorris.io/jokes/random')
  .then((response) => {
    return response.json();
})
```

פשוט, נכון? עכשיו אפשר לשרשר עוד `then`, שבו מחליטים מה לעשות עם ה-JSON. במקרה הזה, מופיעים את ערך הבדיקה:

```
fetch('https://api.chucknorris.io/jokes/random')
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    document.write(jsonObject.value);
  });
});
```

זה הכל! אם אתם שולטים בתכנות אסינכרוני בכלל ובתכנות מבוסס promises, לא צריכה להיות לכם בעיה בהבנת הסינטקס הזה: fetch מחזירה promise עם סוג המידע, ויש לבחור את סוג ולהציג אותו, אז לעשות בו מה שרצים.

מетодות של HTTP וארוגומנטים נוספים

השימוש שהראיתי עד כה הוא שימוש בסיסי בפורמט GET HTTP. פרוטוקול הרשות לא נלמד בספר זה, אך אפשר לציין בקירה שפרוטוקול אינטרנט יש כמה סוגים בקשות. הבקשה הנפוצה היא בקשה GET, לקבל נתונים. זו הבקשה שמבצעים כאשר נכנסים לאתר אינטרנט למשל. מетодות אחרות הן למשל POST – שבה משתמשים כדי לשלוח נתונים (למשל כאשר מלאים טופס באינטרנט, הטופס נשלח בבקשת POST), PUT (לעדכן נתונים) או DELETE (למחיקת נתונים).

אם רוצים לשלוח בקשה POST, אין קל מזה. צריך להעביר ארגומנט נוסף ל-`fetch`, שבו מגדירים את הנתונים שישלחים כשהמתודה מבצעת את הקוריאה לשרת. במקרה הזה `method` ו-`body`. זה מה שנשלח לשרת. מה שמתקיים הוא `response` וכן מחליטים את ה-`response` באמצעות המתודה `then` שמחזירה הבטחה שאונה תופסים ב-`then` השלישי והאחרון.

```
fetch('https://jsonplaceholder.typicode.com/posts/',
{
  method: 'POST',
  body: { title: 'MyTitle' }
})
.then((response) => {
  return response.json();
})
.then((jsonObject) => {
  console.log(jsonObject); // { id: 101 }
});
```

בדוגמה זו שלבים בבקשת POST עם אובייקט אל הכתובת של השירות: GET, אלא שכאן נדרש להעביר ארגומנט שני שmaps את סוג הבקשה ומכובן את המידע. באותו אובייקט אפשר גם להגדיר headers למשל. על מנת להבין את סוגי הבקשות יש צורך בידע על HTTP שאינו נלמד בספר זה, אבל הפרקטייה היא די פשוטה. כאמור, fetch הוא פשוט מאוד להבנה אם מכירים היטב תכונות אסינכרוני ו-promise.

תרגיל:

נתון ה-**API** שכתובתו ה'יא <http://jsonplaceholder.typicode.com/posts/1> – הדףו את הcotרת המתקבלת מהגישה לכתובת זו.

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    document.write(jsonObject.title);
  });
}
```

הסבר:

אם נכנסים אל ה-**API** זהה, רואים שמתתקבל אובייקט **JSON** מהסוג זהה:

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi
  optio reprehenderit",
  "body": "quia et suscipit suscipit recusandae consequuntur expedita
  et cum reprehenderit molestiae ut ut quas totam nostrum rerum est autem
  sunt rem eveniet architecto"
}
```

כעת רוצים את ה-`title`.

ראשית יוצרים `fetch` אל הכתובת. זה קל ופשוט. הfonקציה `fetch` ממחישה `promise` שאוטומטית מוחזר `promise` עם ארגומנט התגובה וצריך תמיד להחזיר את מה שרצים מהתגובה זו, לנוינו – **JSON**. מוחזרים את:

`response.json`

כיוון שאפשר לשרשר `promise`, צריך להשתמש בעוד `then` על מנת להציג בו את מה שרצים, לנוינו – הcotרת `title`, שיש להדיפה.

תרגיל:

שלחו בקשה מסוג DELETE אל <http://jsonplaceholder.typicode.com/posts/1> והדפיסו "Deletion successful" במקרה של הצלחה.

פתרון:

```
fetch('https://jsonplaceholder.typicode.com/posts/1',
{
  method: 'DELETE'
})
.then((response) => {
  return response.json();
})
.then((jsonObject) => {
  document.write('Delete successful');
});
```

הסביר:

כפי שציינו קודם, יש כמה סוג בקשות. מי שஅחראי לפעול לפי הבקשות הוא השירות ובעל השירות, או המתכוון שכותב את השירות שנמצא על השירות, אומר לנו מה לשלוח. במקרה זה אנו צריכים לשלוח בקשה מסוג DELETE. מה השירות יעשה איתה? זה תלוי בו (במקרה הזה, שירות הדוגמה לא יעשה דבר עם DELETE) אבל במקרה אנו צריכים לדעת לשלוח בקשות שונות זהה המהוות שלו.

על מנת לשגר בקשה מסוג DELETE צריך להעביר ארגומנט שני אל fetch. הארגומנט הראשון הוא הכתובת. הארגומנט השני הוא אובייקט שיכל להכיל כמה תוכנות אבל במקרה זה מכיל רק תוכנה אחת: method, שהערך שלה הוא DELETE. מפה ממשיכים בדיק כמו ב-`fetch` רגיל. מקבלים את התגובה ומעבירים את ה-JSON המתקיים ממנה הלהה. אם מקבלים תגובה מסוג כשלחו, סימן שהמחיקה הצלחה.

ES6 Classes

מחלקות, קלאסים, או `classes` באנגלית, הן דרך מצוינת לארגן קוד בג'אווהסקריפט. יש לא מעט שפות תכנות שימושות במחלקות. באופן עיקוני, בג'אווהסקריפט מחלקות הן פשוט ציפוי של סוכר מעל אובייקט רגיל ופונקציה בנית, שעליה הוסבר בפרק על האובייקטים. אבל כיוון שמחלקות הן כל כך נפוצות, אסביר עליהם כאן, כולל איך הן נראות כאובייקט רגיל. מחלקה היא אובייקט שיש לו מפתחות. חלק מהפתחות הם סוג מידע פרימיטיבים וחלקים פונקציות (ואז קוראים להן מתודות של המחלקה). הנה נבנה קלאס פשוט כדי להדגים איך זה עובד:

```
class Book {
  constructor(title) {
    this.title = title;
    this.isCurrentlyReading = false;
  }
  start() { // Public method.
    this.isCurrentlyReading = true;
    return this.isCurrentlyReading;
  }
  stop() { // Public property.
    this.isCurrentlyReading = false;
  }
}
const theShining = new Book('The Shining'); // Instance of the object.
console.log(theShining.title); // Return "The Shining".
const result = theShining.start();
console.log(result); // Return true.
```

אם זה מזכיר לכם פונקציה בנית מהפרק על האובייקטים, אתם צודקים. זה בדוק אותו דבר והפונקציות היא אותה פונקציונליות, רק שהיכול ארוזיפה יותר. ב-`constructor` יש את הפונקציה הבנית עצמה, זו שרצה כאשר מרים את `new`. בתוך המחלקה מגדירים גם את המתודות.

זה הכל. בדוגמה אפשר לראות איך יוצרים מהבתנית, שהיא פונקציה בנית עצם (שימו לב שגם שם המחלקה הוא עם אות גדולה), את האובייקט הפרט. מחלקה היא פשוט ציפוי של סוכר או "סוכר סינטקטי" על הפונקציה הבנית הרגילה שכבר למדנו עליה. אין צורך להיבהל מהamilים השמורים `class` או `constructor` – זה בדוק מה שלמדנו קודם.

המחלקה המפורטת לעיל בעצם נראה כך מתחמי הקלעים:

```
function Book(title) {
  this.title = title;
  this.isCurrentlyReading = false;
  this.start = () => {
    this.isCurrentlyReading = true;
    return this.isCurrentlyReading;
  }
  this.stop = () => {
    this.isCurrentlyReading = false;
  }
}
const theShining = new Book('The Shining'); // Instance of the object.
console.log(theShining.title); // Return "The Shining".
const result = theShining.start();
console.log(result); // Return true.
```

אבל כשם שיש לנו יש עוד מתודות מוצלחות, שכאמוּר עוזרות לסדר את הקוד בצורה טובה ונעימה יותר לעין.

בפרק על האובייקטים דיברתי על `get` ועל `set`, ובטח תשمخו לדעת שאפשר להגדיר אותם גם פה. אלא מה? ציריך להיזהר מWOOD מהפניות מעגליות. הנה דוגמה שמתבססת על הדוגמה הקודמת. במקורה הזה יש `get` שambil את התוכנה הנדרשת (`title`) וועטף אותה בתבנית טקסט. ה-`set` הוא פשוט יותר:

```
class Book {
  constructor(title) {
    this._title = title;
    this.isCurrentlyReading = false;
  }
  get title() {
    return `Great Book: ${this._title}`;
  }
  set title(newTitle) {
    this._title = newTitle;
  }
}
const theShining = new Book('The Shining'); // Instance of the object.
theShining.title = 'The bible';
console.log(theShining.title); // Great Book: The bible
```

כאמוּר, את הקונספט של `get` ו-`set` כבר למדנו, וכך מוצגת פשטוט דרך פשטוטה יותר ונעימה יותר לעין למשם את מה שאותם כבר יודעים. אין פה משחו חדש. זו הסיבה שמייקמת את הפרק הזה בסוף הספר. מחלוקת היא לא וודו אלא פשטוט דרך שונה ונעימה לכתיבת פונקציה בנית, ואת הפונקציה הבנית שיזכרת אובייקט עם תוכנות ונתודות אתם מכירים.

אם יש כמה פונקציות בנהיות שחוזרות על עצמן, אפשר ליצור פונקציה בנהיות שומרישה לפונקציות הבנאיות את כל התכונות שלהן. אם למדתם תכנות מונחה עצמים, אז זה דומה — אבל רק קצת, כיון שג'אווהסקריפט היא לא שפה מונחית עצמים.

הבה נמחיש זאת באמצעות דוגמה מהח'ים. יש שני סוגים לключи - רגילים ופרימיטים. לשני סוגים הלקחות יש תכונות דומות: לשניהם יש שמות וכתובות ותודת `set` ו-`get`, אבל ללקוח פרימיטים יש גם תכונת "התבות". איך ממשיכם זהה דבר? אפשר ליצור שתי מחלקות באופן זהה:

```
class RegularClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class PremiumClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe", address: "Tel Aviv"}
const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address: "Petah Tiqwa"}
console.log(premium.getBenefits()); // 'A Lots of premium stuff';
```

מגדירים שתי מחלקות, `PremiumClient` ו-`RegularClient` (בתחתיות מודגם שימוש בכל אחד מהן). שתי המחלקות זהות, למעט המתודה `getBenefits` שנמצאת רק במחלקה `PremiumClient`.

באופן עקרוני אין בעיה עם הקוד הזה והוא עובד. אבל מה כן הבעיה? שכפול קוד. יש כאן קוד שוחזר על עצמו, וזה בעיה. למה? כי אם יבקשו מכם שינויים במבנה הלקוח (למשל להוסיף תכונה נוספת), תצטרכו לשנות את המבנה פערם. ואם יש יותר משני סוגים לключи, תצטרכו לעשות שינויים במקומות נוספים. שכפול קוד הוא ממשו שבדאי במידת האפשר להימנע ממנו.

בדוק בשבייל זה קיימ-h-extend, שמאפשר ליצור מחלקה-על ולרשת ממנה את כל התכונות. במקרה זהה – ליצור מחלקה client כללית, ש-*client* ו-*PremiumClient* יעתיקו ממנה הכל. במחלקה *PremiumClient* צריך ליצור את *getBenefits*. כך זה נראה:

```
class Client {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class RegularClient extends Client {
}
class PremiumClient extends Client {
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe",
address: "Tel Aviv"}
const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address:
"Petah Tiqwa"}
console.log(premium.getBenefits()); // 'A lots of premium stuff';
```

אפשר לראות שהקוד נראה הרבה יותר אלגנטית ושאין חזרות. בעצם, לוקח את *Client* ומضاف לו. ברגע שעושים extend, כל המתודות והתכונות של *Client* נכנסות ל-*RegularClient* ול-*PremiumClient* באופן אוטומטי. אפשר להוסיף להן מתודות חדשות. זה אולי נראה לכם כמו קסם, אבל זה לא. זה בדוק מה שלמדנו בפרק על אובייקטים – רק בariezhah חדשה. אם אתם מתקשים בתרגילים, כדאי לחזור שוב לפרק על האובייקטים.

תרגילים:

כתבו מחלקה של מכונית המקבלת פונקציה במבנה שם, צבע וונוף מנوع. לכל מכונית יש מספר זיהוי ייחודי המורכב מחיבור של הדגם, הצבע והונוף. למשל, אם המכונית היא opel, הצבע הוא white והונוף הוא 1,200, מספר הזיהוי יהיה 1200whiteopel. צרו למחלקה "מכונית" פונקציה המחזירה את מספר הזיהוי.

פתרונות:

```
class Car {
  constructor(name, color, engine) {
```

```
this.name = name;
this.color = color;
this.engine = engine;
this.modelNumber = this.name + this.color + this.engine;
}
getModelNumber() {
    return this.modelNumber;
}
}
let opelObject = new Car('opel', 'white', '1200');
let id = opelObject.getModelNumber();
console.log(id); // opelwhite1200
```

הסביר:

באמצעות המילה השמורה `Class` יוצרים מחלקה בשם `Car`. בפונקציה הבנאית מקבלים את שלושת הארגומנטים הנדרשים ונוצר `this`.`modelNumber`. שימו לב ל-`this` – ברגע שימושים בו, `this.modelNumber` זמין לכל המחלקה. יוצרים מתודת `get` מסודרת כדי שתחזיר אותו. למטה משתמשים במחלקה עם הנתונים. אפשר ליצור עכשו איזו מכונית שאתם רוצים. לשם השוויה – הסתכלו על התרגום הראשון בפרק על `new` ו-`this`. זה אותו תרגיל בדיק והתשובה דומה מאד לתשובה הזו, אלא שכן השתמשנו בקלאס. כאמור, המילה `klass` והטכנית האלגנטית יותר מבלבולות מתכנתים שמסתכלים על כל העניין של המחלקות ועל ידו, אבל ברגע שימושים זהה ציפוי מעל מההו שכבר היה קיים בשפה – הכל נראה פשוט יותר.

פרק 19

ומה ענשין?



ומה עכשו?

למדנו לא מעט על ג'אווהסקריפט בספר זהה. התחלנו בעצם מושתנים בסיסיים ומסוגי מידע פרימיטיביים; המשכנו הלאה אל פעולות ואל זרימת קוד באמצעות משפטי תנאי ופונקציות; למדנו גם על מערכים ואובייקטים ועל לולאות ושינויי מערכים ואיך לארגן את המידע; העמינו הלאה בקוד באמצעות לימוד שימושותי של אובייקטים מובנים ואיך ג'אווהסקריפט מתנהלת בסביבת דפדפן; ובפרק האחרון למדנו איך עובדים בקוד אסינכרוני. זה לא מעט למד, במיוחד אם לא רأיתם שפת תכנות קודם לכן.

אם עברתם על כל פרקי הספר סדרם, יש לכם ידע תיאורטי מbasico על עקרונות ג'אווהסקריפט והתחביר שלו. אתם יכולים לדעת לכתוב קוד פשוט, לנתח קוד מורכב יותר ולפתור בעיות. על אף שהביסיס שלכם יציב וטוב, עדין נדרש תרגול על מנת לחזק את הידע שלכם בשפה ולהעшир אותה. אף אחד לא הופך למתכנת בעקבות קראת ספר, טוב ככל שהיא.

ניסו מושי טוב ניתן למצוא בנספח על [Vel](#), שנכתב על ידי מור גלעד, מתכנתת בחברת [Wi](#). בנספח זהה, אפשר למדוד איך בונים רכיב באתר [Wi](#). רכיב שניית להשתמש בו בכל אתר ואך למכור אותו ללקוחות.

אם אתם רוצים להמשיך ולהרחיב את הידע התיאורטי שלכם, אתם מוזמנים להמשיך לספר [הדרכה הנוספים שכתבתי](#). היתי מתחילה בקריאת הספר "לימוד Node.js בעברית" שילמד אתכם ג'אווהסקריפט בצד שרת. ומיד לאחריו בספר "לימוד ריאקט בעברית" שילמד אתכם בניית ממשקים בג'אווהסקריפט. הספרים הללו יותר מעשיים ויסיעו לכם למן את הידע החדש שלכם לתחום מעשי ממש. אם אתם לא יכולים לרכוש את הספרים הללו, המדריכים החינמיים שכתבתי ומופיעים באתר שלי: [internet-israel.com](#) יעזרו לכם. כדי לעبور על הספרים או על המדריכים עד שתתעדו את התיאוריה טוב מספיק.

אבל מהתיאוריה צריכים לעبور למעשה ולכתיבת קוד שעושה משהו, זהה הצעד הכי קשה. איך בדיק עוברים ממצב שבו אני יודע (תיאורטי) לכתוב קוד ג'אווהסקריפט למצב שבו אני כותב משהו שעבוד ושאנשים משתמשים בו?

הדרך הכי טובה היא לבנות ולכתוב. השאלה היא מה. פה שום ספר או מדריך כבר לא יעזרו לכם וזה תלוי בכם ובצריכים שלכם. ג'אווהסקריפט היא השפה הפופולרית ביותר בעולם, ומשתמשים בה בהרבה מקומות: באתר אינטראקט, באפליקציות [דסקטופ](#), [móvel](#), בשירותים ואפיו לזרבי אבטחת מידע. אחרי שלמדתם את אחד מהנושאים הללו או את כלם, כדי להתנסות בקוד ממש.

המקום הטוב ביותר להתחילה הוא באמצעות תרומות קוד לפרויקט קוד פתוח שאפשר למצוא ב-GitHub. עבודה במסגרת פרויקט קוד פתוח זהה יכולה להעшир ולהעמיק מאוד את הידע שלכם. בחרו פרויקט מbasico ג'אווהסקריפט ונוסו לתרום לקוד שלהם. באמצעות פתרון באג שמתועד בפרויקט, באמצעות תוספת לדוקומנטציה, לבדיקות אוטומטיות או אפילו באמצעות הוספת תכונה מסוימת. נסו להתקין קומפוננטת ריאקט על אתר של חבר, לבנות תוסף קטן ל-Query עבור האתר של העמותה החביבה עליהם, לתרום לדוקומנטציה של קומפוננטה שאתם

אהובים במיוחד, להגיע להאקטון ולהציג לוצאות קיימ או אפילו רק לשבת ולהסתכל על מתכנתים אחרים. אפשר אף רצוי להתנדב בעמותה ולהתנסות בעבודה בפרויקט ממש שאנשיים השתמשו בו. יש הרבה דרכים – איזו דרך תבחרו? זה תלוי בכם.

יש לא מעט מיטאפים ופגישות של מתכנתים שכדי להגיע אליהם. במפגשים האלו פוגשים מתכנתים שעובדים בחברות מגוונות, והאווירה נעימה מאוד. אפשר ורצוי לשאול את האנשים האלו איך מתקדמיים הללו בלימוד. יש גם קבוצות פיסבוק בנושא, שהאנשים הנחמדים והמקצועיים בהן יכולים לסייע לגבי המשך הדרך.

באתר זהה יש רשימה של מקורות ושל קבוצות שניים משל עצמם בקשר להשתתפותם:

<https://github.com/barzik/web-dev-il-resources>

כך או אחרת, חשוב מאוד להמשיך לכתוב בשפה. כאמור, שפת תכנות אינה שונה ממהותית משפה מדוברת; אם לא תשתמש בה, תשכח אותה. אפשר להשתמש בג'אווהסקריפט במגוון עצום של פרויקטים – רק צריך לבחור פרויקט ולהמשיך לתרגל. להשתלב בתחום ההיבט זה לא בשמיים זהה אפשרי. אם אני הצלחתי – כל אחד יכול.

נספח: Best Practices

מחברים: שחר טל, רוני אורבר, דניז רוזג, נופר ברנס, חברת **Really Good** — בוטיק Front End, מאז 2012. ReallyGood.co.il

מה זה Best Practices ולמה כדאי לישם אותם?

השאלה Best Practices הם כלליים, המלצות ומוסכמות שנעודו לסייע לייצור תוכנות איכותיות, בין השאר באמצעות כתיבת קוד בצורה טובה ונכונה שמצוירת טעויות ובלבול.

תוכנות הוא פעולה חברתית. כשוחרים על תוכנות, לרוב מדריכים האker בודד שיישב בחושך בקופץ'ון ומקליד במרץ בלי לחשוב שנייה, כאילו הקוד עף מתוכו. הסיטואציה הזאת היא נדירה, ובינינו, זה די לא נוח לשבת לתוכנת כהה. לרוב תוכנות נכתבות עם קולגות במקומות העבודה או עם חברים או אפילו עם אנשים זרים באינטרנט — והיצירה שלהם דורשת קצת יותר סבלנות ושיתול דעת. מתכנתים אחרים יסתכלו על הקוד לפני שהוא יוכנס באופן סופי לפרויקט (אדרב על *Code Review* בהמשך), ומתכנתים אחרים יצטרכו להבין ולבוד איתו בעתיד כשירצטו להוציא, לשנות או לתקן את התוכנה.

לאורך הזמן מתכנתים עוזבים צוותים אחרים צריכים להתמודד עם קוד שהכתבו שלו כבר לא בסביבה, וגם אתם בטח תרצו להשאיר אחרים קוד שמתכנתים אחרים יוכלו ואפילו ישמשו לעבד איתהו. אפילו בפרויקט שכלו שלכם, ככל שהזמן עובר גדל הסיכוי שיום אחד תסתכלו על קטע קוד ותשאלו את עצמכם "מי לעזאזל כתב את זה?" והתשובה הטריגית-קומית תהיה — אתם בעצםם. لكن חשוב לשומר על קוד מסודר ומארגון שנכתב בדרך מסוימת. כך מוצאיםם באגים וטעויות, מKİלים על כולם את קריית והבנת הקוד במהירות וחושכים רعش רקע ויכוח שרק על שטויות — מחליטים פעם אחת על סגנון ומדיניות ונמנעים מ"מלחמות עריכה" אינסופיota שבחן כל אחד מסדר את הקוד איך שהוא והוא אחריו מבטל את השינויים.

אפשר להסביר על כלליים כאלו בעל פה, אפשר בכתב, ויש כלים שадון בהם בהרבה ממש כאן, שיכולים לעזור לאכוף לפחות חלק מהכללים באופן אוטומטי.

במיוחד בשפה גמישה כמו ג'אווהסקרייפט, שמאפשרת לעשות המון דברים יצירתיים ובתוכם המון שטויות, מומלץ לעקוב אחרי Best Practices מקובלים בתעשייה, שאולי לא נכתבו בdam אבל בהחלט עלו ביעז ובדמות. מעבר למשמעותם ולשימוש מושכל באפשרויות שהשפה מציעה, חלק מה-Best Practices הם העדפות סטנדרטיות של המתכנתים בפרויקט, שהתקבעו והמצוות רוצה לשמר — בשני הסוגים יש מקום לשיקול דעת בריא, ולא צריך לחתת הכלול כתורה מסיני בלי להבין ולבוחן מה מתאים לכם.

דוגמה להמלצה שמנועת טעויות היא תמיד להשתמש בהשוואה קפדיות, כמו `==` או `!=` במקומם של `==` המתירנית יותר, שמבצעת המرة לסוג המשתנים לפני השוואה (`type coercion`).

לא מומלץ:

```
if (numberOfThings == possiblyNotANumber) { /* code */ }
```

מומלץ:

```
if (numberOfThings === possiblyNotANumber) { /* code */ }
```

דוגמה לכל סובייקטיבי יותר, פשוט מבטא העדפה סגנונית ועובד לשמר על אחידות, היא ריווח בין חלקי קוד שונים, למשל ריווח בין שם פונקציה לבין הסוגרים שגדירים את הארגומנטים:

אופציה א':

```
function jump(x) { }
```

אופציה ב':

```
{ } (x)
```

אופציה ב' היא המומלצת עליינו כי היא מאפשרת להבדיל בחיפוש בין הגדרת הפונקציה לкриאות אליה.

סת הכללים והמלצות יכול להיות שונה מאוד בין פרויקטים שונים, ועובדות יותר יש חשיבות בשם העקביות והשפיות להתגבר על העדפה האישית שלכם ולכבד את מה שכבר נקבע – מומלץ לדבר בפתיחות ולבחון ביחד צורך בשינויים.

שכל מפתח מקצועי צריך להכיר Best Practices

בחירה שמות

משתנה או פונקציה עם שם טוב יאפשרו לכם להבין מיד מה התפקיד שלהם. אם השם לא מצליח תCENTER לראות איפה הגדרו את המשתנה ומה הוא אחסון בתוכו או לקרוא את הפונקציה ולפעננה מה ניסו לעשות בה. פתרון אפשרי הוא להוסיף הערה, אך עדיף פשוט לבהיר שם טוב. שם טוב הוא קצר וקובל יותר מהערה, מעבר לכך שモטב לא לפזר הערות ליד כל מקום שבו משתמשים במשתנה או בפונקציה.

איך בוחרים שם טוב? שיטה מקובלת עבור פונקציות היא שיטת ה-חסוך-verb (פועל-שם עצם).

לדוגמה:

- `getStudents`
- `sortColorsByBrightness`
- `setLunchBreakTimer`

וכשנתקלים בו צריך להזכיר באיזה טימר מדובר. `setLunchBreakTimer`

יש גם יוצאי דופן בולטים – שמות משתנים סטנדרטיים שצורך להכיר ואין צורך להסביר או לפרט אותם. דוגמה לשמות אלה הם `x` ו-`y` בהקשר של פיקסלים או ציונים על המסר, או `z` בשימוש הקלאסי שלהם בלוולאות. אבל חוץ מהם, כדאי לזכור שמעט תמיד תמיד משתנה של אותן או שתיהן זה רעיון רע. אין שום יתרון בראשי תיבות מסוורים על פני מילים מובנות.

שם המשתנה הוא לא הדבר היחיד שיעזר לך להבין הקוד. אפשר לכתוב אותו שם בכמה דרכים בעזרת אותיות גדולות וקטנות. כל דרך צו נקראת "קִיּוּס", ומיעודת לייצג סוג המשתנה אחר.

אומנם השם זהה, אבל על כל אחד מהמשתנים האלה נוסף רויבד של משמעות שונרמז רק מבחןת הקיון:

camelCase

lunchBreakTimer

כל המילים מלבד הראשון מתחילה באות גדולה; אין רווחים בין המילים.
קיים זה נמצא בשימוש עבור כמעט כל המשתנים והפונקציות בג'אווהסקרייפט.

PascalCase / TitleCase

LunchBreakTimer

כל המילים מתחילה באות גדולה; אין רווחים בין המילים.
קיים זה נמצא בשימוש כמעט כל אוטומט, וככה מקל את הבדיקה בין קלאו ל-`instance`.

SCREAMING_SNAKE_CASE / UPPER_CASE

LUNCH_BREAK_TIMER

כל האותיות גדולות; רווחים בין המילים הופכים לקו תחתון.
קיים זה נמצא בשימוש בעיקר קבועים, שבדרכן כל יוצבו במשתני `const`.

kebab-case

lunch-break-timer

כל האותיות קטנות; הרוחים בין המילים הופכים לקו מפץ.
נמצא בשימוש ב-HTML וב-CSS עבור שמות של `p`, קלאסים ואלמנטים.
קיים לא נוח לשימוש בג'אווהסקרייפט כי קו מפץ אינו תואם חוקי בשמות משתנים, אך שקרה עד בלתי אפשרי להשתמש בו ברוב המקרים.

snake_case

lunch_break_timer

לא בשימוש בג'אווהסקרייפט בדרך כלל.

KISS

"Keep It Simple, Stupid", אמירה שקלעה בול כשנtabעה לפני יותר מחמשים שנה על ידי מהנדס המטוסים קלי ג'ונסון בלוקהיד, ועודין מתאימה כהנחיה למכונטים של 2020 והלאה. כתבו את הקוד בדרך פשוטה והקצרה ביותר, כל עוד הקיראות נשמרת. למשל, הימנו מסינטקס איזוטרי שלא נמצא בשימוש רחב ומהתחכחות יתר בכלל.

DRY

Don't Repeat Yourself. אין סיבה לכתוב את אותו קוד יותר מפעם אחת. לעיתים זקנים לאותה פונקציונליות בכמה מקומות אבל מהססים להפוך אותה לפונקציה מכיוון שמדובר בקוד פשוט וקצר או בקוד שדורש שינוי קטן בכל שימוש שלו.

אל התססו. כשבוטפים את הקוד בפונקציה גם נתונים לו שם, שהופך את השימוש בו לקלירא יותר. ככל שימושים יותר קוד שלא לצורך, מגדילים את הסיכון שהמוכנת הבא יפספס חלק מהמקומות שבהם יש קוד כפול, והתחזקה התהופר לקשה ורוצפת תקלות.

זכרו – התהופר מ-DRY הוא WET, ראשיתibus של **Write Everything Twice** (-;

לא להמציא את הגלגל

למרות שתכנות זהEIF ומעניין, לא בכל דבר שבונים צריך להקימם הכל מאפס. כמו שכנראה ברור כיום שתשתמשו בספרייה כלשהי שתעזר לרender את הממשק, נניח ריאקט, כהה חשוב להשתמש גם בספריות ובכלים קטנים ומוקדים יותר כזהה הגיוני. מהווצאת בקשوت לשרת עם `axios`, דרך ספריות לפעולות נפוצות על מערכים אובייקטיבים כמו `lodash` ועד לאלפי הרכיבים המוכנים לרוב חלק הממשק שתרצו לבנות – תהיה בטוחים שימושו כבר פיתח פתרון טוב כמעט לכל צורך. הבה נודה באמת – אתם לא הראשונים שcottובים `Color Picker` או `Dropdown` עם השלמה אוטומטית, וגם אלגוריתמים לערבוב רשימות כבר נ כתבו בעשרות שקדמו להצטרפות לכם לתחום.

יש שלושה מקרים שבהם דוחק הגיוני להיכנס לעובי הקורה ולכתוב בעצמכם משהו לא אלמנטרי:

1. אם אין פתרון מוכן שעונה על כל הדרישות שלכם (עדין זכרו שלעתים יש רכיבים קטנים יותר בתוך המכלול שיוכלו לעזור לכם או שתוכלו למוד מהם, אז תמיד טוב להסתכל עליהם).
2. אם זה ממש התחום שלכם, מה שבשבילו קמתם בבודק ואתם תרצו לקחת אותו כל כר רחוק שכנראה אף כי מוכן לא יספק לכם – למשל במסגרת העבודה ב- *Really Good* אנחנו עובדים עם פלטפורמת יידיאו מובייל, אז כשהתבקשנו לפתח נגן יידיאו חדש ייצרנו אותו מאפס על בסיס תגיית ה-video של HTML, בלי אף ספרייה חיצונית לניגון יידיאו.
3. אתם רוצים לתרגל או נבחנים על פיתוח עצמאי של משהו זהה. בהצלחה!

בשלוי הדברים, שימושם לב לרשימות בקוד פתוח – רישיונות שונים מאפשרים שימוש בתנאים שונים, וחלקים מגבילים שימוש מסחרי. אם אתם חלק מארגון גדול ולא בטוחים, כדאי לווודא שרשישון השימוש מקובל על הגורם המשפטי בחברה. בכל מקרה תמיד טוב לקרוא ולהכיר את הרישישון של הקוד שימושתם בו, גם באופן פרטי.

Make it work, make it right, make it fast

כאשר כתבים תוכנה, גם קטנה אבל בעיקר גדולה – אי אפשר לכתוב את כולה בזורה מושלמת במכה אחת. מתחילה מגרסה בסיסית שפשטית ועובדת עם הfonקציונליות הבסיסית. אחר כך חוזרים לעבות ולכ索ת את כל הדרישות, ומשפרים את הקוד כך שהיא קרייה יותר, תמציתית ומובן ככל האפשר. אל תתפתטו להקדים את המאוחר ולחטוא ב- "Premature Optimization or " – אופטימיזציה שומרים לסופ.

תיעוד

כבר למדתם מוקדם יותר בספר על הערות בקדוד. שני השימושים היכי נפוצים בהערות הם:

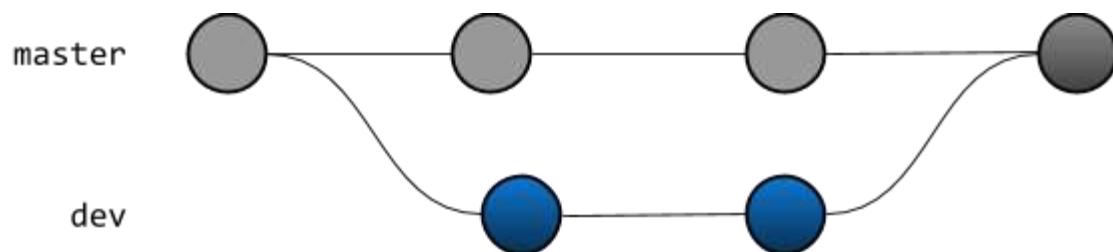
1. כיבוי זמני של חתיכת קוד ביל' לאבד אותה. נוכח תור כד' פיתוח ובדיקות, אבל שמו לבש העורות han לא שיטה לשמר גרסאות של קוד – מייד אצלול לניהול גרסאות נכון.
 2. תיעוד – להסביר מה מאפיים שיקריה או למה פיתחתם משהו בצורה מסוימת. תיעוד טוב מօסיף מידע, ולא סתם חוזר על מה שהקוד עשו. אפילו כמה מילות הסבר עדיפות על כלום. הערה שכתובה טוב יכולה לחסוך למכונת אחר הרבה כאב ראש. כשאתם ניגשים לכתיבת התיעוד, נסו לחשב מה הדבר היכי מורכב או לא ברור בקוד שלכם ושימו את הדגש שם. מתקנות אחרים יודו לכם, ואפילו אתם תודו לעצמכם כשהתחרזו לעבוד על קוד ישן ומתווד היבט שלכם.

ניהול גראסאות

בגאל שקדם הוא בסופו של דבר קובץ טקסט שנערכים על המחשב שלכם, הפתרון הכי פשוט (ונאיבי) לניהול גרסאות הוא יצירת עותקים של תיקיות בנקודות זמן מסוימות. במצב זה – קל לאבד קוד וקשה לעבוד בצוות. המחשב יכול לשבוק חיים או שתצטרכו לשחזר קוד שמחקתם וזה יהיה קשה עד בלתי אפשרי. לא רק זה – אם מישחו מהוצאות ירצה לשנות קוד הוא יצטרך לברר אצל מי הקובץ החדש ביותר ולבקש ממנו שישלח לו את הקובץ. באיזו שנה אונחונו?

כדי להתגבר על כל הבעיות הללו, משתמשים בפתרון ניהול גרסאות (Version Control) או Source Control. הפתרונות הנפוצים ביותר הם Git ו-SVN. לגיט יש כמה יתרונות שימושיים ובראשם מהירות, ביזור שמאפשר יכולת עבודה נוחה ללא חיבור לאינטרנט וקהילה עצומה – כמעט כל עולם הקוד הפתוח היום מתנהל על בסיס Git. לרוב ב-db.qithub.com.

שימוש בניהול גרסאות מאפשר למפתחים רבים לכתב קוד בכמה ענפים (Branches) שונים במקביל ולשתף אותם ביניהם. אפשר לראות איזה מפתח כתוב מה ומתי, אפשר לשלב כמה ענפים מוכנים אל תוך גרסה אחת שאותה משחררים, והכי חשוב — אפשר לחזור אחרה במידת הצורך.



לביקש עזרה

כשנתקעים במהלך פיתוח — לבקש עזרה זו לא בושה. אם אחרי Debugging, בידוד הבעיה ובדיקה הלוגים (אם יש כאלה) עדין לא צלחתם את הבעיה, עוברים לוגול. מנסים לחפש את השגיאה בצורה הכי נקייה שאפשר, בלי שמות קבצים או מספרי שורות מהקוד הספציפי. לרובה המזל, רוב הבעיה נפוצות ומעט תמיד אפשר למצוא פתרון או לפחות כיוון או גישה שיעוכלו לעזור.

אם יש מתכנתים נוספים בצוות אפשר לשאול אותם. מתכנתים מנוסים או כאלה שנמצאים בפרויקט כבר תקופה מכירם את הבעיה הנפוצות והאתגרים שאתם עשויים להיתקל בהם ויכלו להכוין אתכם ולהסביר לכם זמן יקר. שיתוף הבעיה עם חברי הצוות גם גורם לעבר על הקוד פעם נוספת ואפשר למצוא פתרון תוך כדי תיאור הבעיה — מניסיון, זה עובד.

אם בכלל זאת לא מצאתם תשובה בגוגל או אצל המתכנתים האחרים בצוות, זה הזמן לשאול את השאלה שלכם אונליין. האתר הכי פופולרי לשאלות תכנות הוא StackOverflow, שמנגן דירוג התשובות בו עוזר לzechות מה מבין התשובות עובד ורלוונטי בעני הקיילה, בוגר לפורומים שגם הם רלוונטיים אך בחיפוש פתרונות בהם נדרשת קראיה יותר ביקורתית. אין מה להתבזבז, השאלה שתשאלו תהיה כנראה רלוונטית למתכנתים אחרים בעtid שיתקלו באותה בעיה ויחפשו תשובה.

בדרך לפרסום שאלה תצטרכו להכין דוגמה מוקัดת וمبודדת של הבעיה (מכונה בדרך כלל Reduced Test Case), וגם זה תחילך מומלץ שעצם הרשקה בו עשוי להוביל אתכם לפתרון עוד לפני שתפרנסמו משהו.

ביקורת עמיתים — Code Review

סקורתי הרבה kaliim, וזה יכול להיראות מיידיים. האם באמת מקפידים על כל צוות פיתוח?

התשובה היא שאמנם לא מקפידים במאת האחדים, אבל מפתחים וארגוני מקצועיים יתעקשו לעמוד בכמה שיטות מהם, כי מוצרים הולכים וمتפתחים וככל שפרויקט מתuba ונוספות עוד

שכבות של קוד כך נהיה קשה לתחזק אותו ולהבין מי נגד מי. אם מראש היסודות רועעים ואחד לא מקיים על Best Practices, מהר מאוד תמצאו את עצמכם מול "קוד סגטי" (שمعורבב כמו סגטי בצלחת) שקשה מאוד לתחזק.

אם זכיתם לעבוד בצוות ולא בלבד, Code Review עשוי להיות חלק מה אחראיות שלכם, זהה מזוין. העבירו את הקוד שלכם לבדיקת מתקנת אחר והימנו ממה שבאים מיותרים. תנו קצת רקע כללי על ההקשר של הקוד ומה ניסיתם להשיג ואפשרו לקוד לדבר בעד עצמו. כך, אם חסר תיעוד – זה יבלוט. למי שעבוד בלבד, יש אתרי אינטרנט וקובציות שבהם אפשר להתייעץ, לשתף קטעי קוד ולבקש ביקורת.

את העורות על הקוד מקבלים ונוטנים בשתי דרכים. הדרך הראשונה היא בעל פה (פניהם אל פניהם או בטלפון). הדרך השנייה היא בכתב, במצווד לשינויים במערכת ניהול הגרסאות. לדוגמה, אם עובדים עם Git, אפשר להשאיר העורות על Pull Request. בקשה זו היא בעצם בקשה דרך מערכת ניהול הגרסאות לאשר את הקוד ולמzag אותו לתוכן הענף הראשי של הפרויקט. מתקנת שבודקת קוד תוכל להשאיר עליו העורות, ולאחר שהמפתח יבצע את השינויים היא תוכל לאשר הבקשה כדי למzag את שני הענפים.

כשאתם מקבלים ביקורת, קבלו באבבה הצעות, העורות ושאלות. אל תיקחו את הביקורת כתקיפה אישית נגדכם. זכיתם בעוד זוג עניינים שייעברו על הקוד שלכם, לא משנה למי מהם יש יותר ניסיון. קחו אותה בשתי ידיים, הקשיבו באמת ואל תרצו למוגננה – אולי אם עבדתם קשה על משהו ואולי קצת התאהבתם בפתרון מסוים או נקשרתם אליו, או אם הערה שקיבילתם תדרוש שניי יחסית גדול והרבה עבודה מצדכם. זכרו, באתם לבודד ביחד וליצור קוד איקוטי וברור, לא להראות מי יותר חכם וצדוק.

כשאתם עורכים למשהו אחר Code Review – נסו להבין מה קורה בקוד ולהעיר אם היה מסוגלים להרחבת או לשנות אותו במידת הצורך. בחרנו את הקוד אל מול עקרונות מנהיים כמו כל אלו שמוזכרים בפרק זהה. האם המשות הגיוניים וברוריהם? הפונקציות או הקבצים ארוכים מדי? יש קוד ש חוזר על עצמו והיה אפשר להפוך ליותר Y? קוד מסודר ומעומד בצורה עקבית וקריאה? וכן הלאה.

בסוף דבר, בכל יום לומדים משהו חדש. Code Review היא דרך מצוינת ללמידה מהאנשים סביבכם ולהמשיך להתפתח בתור מתקנתים. זה הרגל מעולה שיחד אתכם מתקנתים זההות קוד פחות טוב ולכתוב קוד יותר טוב. כולם מרוויחים.

Tech Design

כשניגשים למאיץ פיתוח רציני, סוף מעשה במחשבה תחילה. כמה שמנסים לקבל באבבה ביקורת, כמעט תמיד מתבאים לשימוש בסוף התהליך שהוא עדיף להשתמש בספריה מסוימת שכבר קיימת ולא יעדתם עליה או שאית אפשר ללכט על פתרון שיש מתחם לפתח משיקולים שונים שלא הכרתם. לעיתים אפילו תיתקעו עם הפתרון הלא-מושלם שלכם כי מאוחר מדי לחזור אחורה ולתקן.

כדי להימנע ממצבים כאלה, ראוי לעשות תכנון טכני, שהוא מסמך בפורמט גמיש. יש איזורי התמחות שבהם הגיוני להשתמש בתרשימי זרימה (למשל תכנון של ארכיטקטורת שרת או מסדי נתונים); כשעובדים גם בצד לקוחות וגם בצד שרת מקובל לתאר את ה-*API* שמצפים לבנות לקריאות בינם — لأن שולחים כל בקשה, אילו פרמטרים מעבירים ואיך בדיקת תיראה תגובה טיפולית.

בעזרת *Tech Design* טוב אפשר לוודא שמבנים זה את זה ולהתחל לפתח את הצדדים במקביל, ולא לגלות בסוף שכל אחד תכנן מבנה נתונים שונה לחולוטין ולהתוווכח מי צריך להתאים את עצמו לאחדר.

את התכנון הזה תעבירו לביקורת עמיתים, דברו עליו וצאו בדרך רק אחרי שהסכמתם על הפרטים — ככה תמנעו לפחות חלק מההפתעות האדירות והיקרות בשלבים מאוחרים יותר.

Best Practices ואוטומציה

از כתבתם *Tech Design* וקיבלתם אישור לצאת בדרך, עבדתם קשה ואז הגיעו ערמה ענקית של הערות-*Code Review*. איך בכלל מתחילה לעבור על זה? עם כל הכאב וההערכה, איך לא לוקחים את זה קשה כהויל אתם מרגישים ש"מחפשים אתכם" ושכמעט על כל שורה יש למשהו מה להגיד?

עבדתם יפה! לדברים יש נטייה להתבלגן כי כולם בסך הכל אנושיים. היו לכם כוונות טובות, אבל יכול להיות שפה שכחتم להיות עקביים בסגנון, שם העדפתם להעמיס יותר מדי על פונקציה אחת, וmdi פעם לא הקפדתם על מהשו מהכללים המומלצים. זה טבעי, אבל כשמכפילים את הנטיה האנושית לעגל פינות ולפספס פרטם במספר אנשי צוות ובכמה חודשי פיתוח, האיכות מידדרת מהר מאוד.

אחד הדברים שהכי עוזרים בRICT המעודד של הביקורת, ובכלל בשמירה על איכות על ידי איתור ומינעת שגיאות מראש, הוא אוטומציה. יש כלים אוטומטיים מעולים, שאציג בהמשך, שעוזרים לשומר על סדר ולהקפיד על הכללים שתקבעו לעצמכם, לבד או כצוות.

כך, כשתגיעו ל-*Review* תדעו שבחלק נכבד מההערות פשוט לא תיתקלו, כי כבר קיבלתם פידבק מיידי או תיקון אוטומטי בזמן אמיתי. זה כבר מאחוריכם, וחוסכם גם מהעמיים וגם מעצמכם דינונים על ריווחים, שורות ארוכות mdi ועוד המונע הערות, מהותיות יותר או פחות. יהיה קל, מהיר ונעים יותר להתמקד ולהקשיב לביקורת שכן תגיע כשהיא מצומצמת. יחסית.

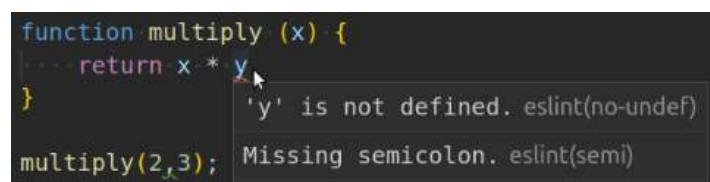
בצווות, מומלץ מאוד להחליט פעם אחת (מוקדם ככל האפשר) על הקווים המנחים שלכם, להיעזר בכלים האוטומטיים שתלמדו עליהם בחלק הזה וליצור מערכת אוטומטית שתפקידה על הקוד שנכנס לפרויקט. לרובה השממה, מתכנתים אחרים כבר בנו מערכות כאלה, ואך נתנו לפועלות האוטומציה של פיקוח על הקוד שם — *Linter*.

מה זה linting?

המילה `lint` (לינט) באנגלית פירושה מזג – הלקוח המctruber בכיס של בגד או הבדורים הקטנים שנוצרים על גבי הסרג' לאחר כמה כביסות. בעת כתיבת ועדרון קוד מתווספים לאט לאט, שגיאות או דפוסים לא רצויים אחרים שנוגדים את-best Practices שקבעתם – זהו המזג, וכן יש לעבור על הקוד עם כל שמשיר את המזג; בהשאלה, קוראים לו `linter`. לינטרים יודעים לזרות את השגיאות והדפוסים שהגדרתם, להתריע על הימצאותם ולפעמים אפילו לתקן אותם אוטומטית.

בעולם התוכנה, הלינט הוא כל שטאקיידו לבחון את הקוד שכתבתם ולהזהיר משגיאות, באגים או טעויות סגנוניות. הלינט הראשון פורסם ב-1978, בדק קוד שנכתב בשפת C ונעשה בו שימוש לבודיקת מערכת הפעלה אונס.

עורכי טקסט מודרניים המיעדים לכתיבת קוד ג'אווהסקריפט מפעילים בודק שגיאות מובנה שמשזהיר את המשתמש משגיאות שימנעו מהקוד לזרז בצורה תקינה בדף. אם כן, למה עדין צריך לינט? קוד יכול לפעול באופן תקין וудין לא לעמוד ב-best Practices ובכללים שקבעתם מסיבות שונות, וכן רוצים להריץ על הקוד גם לינט שזהיר מחריגות וטעויות מכל הסוגים.



```
function multiply (x) {
  return x * y;
}
multiply(2,3);
```

`'y' is not defined. eslint(no-undef)`

`Missing semicolon. eslint(semi)`

لينט יכול להזהיר מפני בעיות כמו שימוש במשתנים שלא הוגדרו, קריאות לפונקציות שאין קיימות, הפרת מוסכמות בנוגע לירוח וסידור של קוד, שימוש בפיצרים של השפה שנוטים להוביל לבעיות אבטחה ועוד. הלינט יסמן בעיות באמצעות סימן בקו מזוגאג אדום לשגיאות וירוק לאזהרות, בדומה לסימן של שגיאות כתיב במעבדי תמלילים. במעבר עכבר מעל הקוד המסומן אפשר יהה לראות את הסיבה ולעתים גם הצעות לשיפור הקוד.

הכל הראשו בעולם הג'אווהסקריפט שעשה זאת היה `JSHint`, שכך סט מוגדר מאוד של כללים בצורה קשוחה – מי שלא אהב את הדעות החותכות של היוצר שלו, אושיתת הג'אווהסקריפט דאגלס קראוקפורד, נאלץ להשתמש בכלי אחר בשם `JSHint`, שכן אפשר יותר גמישות.

עם השנים וההתפתחות המהירה של ג'אווהסקריפט, `ESLint` של ניקולס זאקאו תפס תאוצה, בין היתר בזכות מבנה מאוד גמיש. `ESLint` מאפשר להציג בrama פרטנית את העריכים הרצויים לכללים קיימים (כמו למשל אורך שורה מקסימלי או איפה בדיק מותר או אסור להשאיר שורות ריקות), אבל גם להציג כללים משלכם, ולהשתמש בקלות בכללים אחרים יצרו, כמו שארחיב עוד מעט. נכון לעכשיו, `ESLint` הוא ה-`linter` הפופולרי ביותר עבור ג'אווהסקריפט, וכן אתמקד בו.

ESLint

עורכי ג'אווהסקריפט מודרניים יודעים להציג הערות של ESLint בצורה אוטומטית או בהתקנה פשוטה של תוסף. האזהרות והשגיאות שייצגו נקבעות על פי קובץ הגדרות, בדרך כלל בשם eslintrc, בתיקייה הראשית שלuproject. אם אצלכם עדין אין קובץ זהה, קראו על [this](#) וצרו באמצעותו קובץ הגדרות משלכם — מומלץ להתחיל מאחד הסטיטים הקיימים שיוצעו לכם במהלך האתחול.

אם בחרתם סט חוקים שנוח לכם להשתמש בו אבל יש בו כמה חוקים שלא נראים לכם, אל תהססו להיכנס לקובץ ההגדרות ולשנות אותו.

בחלק הבא עברו על כמה Best Practices, חלקם חדשים וחלקם כבר הזכירתי קודם, לצד הכלל ב-ESLint שמשיע לצוית להם.

רישימה של Best Practices של ESLint והחוק הרלוונטי של ESLint

בלי מספרי קבוע

שם הכלל: no-magic-numbers

"מספרים קבועים" הם מספרים שמופיעים בקוד בלי הסבר מפורש, כמו הצבעה למשתנה שהשם שלו יכול להאיר עיניים. הוצרך בולט בחישובים, וביחדש בחישובים שמשלבים כמה מספרים. ברגע הכתיבה ברור איך מוגעים אליהם ווים או חדש אחר כך כבר מוגדים בראש ולא מבינים מה הסיפור של המספר הזה.

דוגמה לקוד בעיתוי עם מספר קבוע:

```
setTimeout(ringBuzzer, 180000);
```

למה 180,000 בעצם? בלי הערה שתספר למה התכוון המשורר, קשה לקלוט במהירות במה מדובר. אפשר להוסיף הערה כמו "three minutes from now", אבל בנסיבותיו לוקח זמן להבין מה אומר המספר הזה ו איך לשנות אותו אם בקשו מכם להאריך את ההמתנה לחמש דקות. מחשבים מהירים מאוד בחישוב, לא צריך לרחם עליהם וללעוס עבורה את המספרים מראש. חשוב יותר שהקוד יהיה מובן לכם ולשאר בני האדם שיצטרכו להבין את הקוד, אז מפרקם את המספר הלא מוסבר לגורמים:

```
const MS_IN_1_SECOND = 1000;
const SECONDS_IN_1_MINUTE = 60;
const DELAY_MINUTES = 3;
const BUZZER_DELAY = DELAY_MINUTES * SECONDS_IN_1_MINUTE *
MS_IN_1_SECOND;
setTimeout(ringBuzzer, BUZZER_DELAY);
```

השוואה **קסדנית**: `==`

שם הכלל: **eqeqeq**

זה הכלל של לינטרים בג'אווהסקריפט, עוד מ-`JS` המוקורי. ג'אווהסקריפט מאפשרת להשוות בין ערכים מסוימים שונים. זו תכונה מעניינת שמאפשרת גמישות ויכולת להיות שימושית, אבל היא ביכולת לגרום לבלבול ולהטעות. כדי להימנע מטעויות קשות לאיטור כתוצאה מההשוואה המתירנית, מקובל וומלץ להקפיד על שימוש בהשואות הקשוחות יותר. מומלץ להשתמש ב-`==` לבדיקה שווין וב-`!=` לבדיקה אי-שווין. אך משווים בין ערכים בלי להמיר את סוג המשתנים תוך כדי ההשוואה.

קוד לא נגיש

שם הכלל: **no-unreachable**

תוך כדי `debugging` ו-`refactor`, קורה שנשארת פקודה שיצאת מקטע הקוד בשלב מוקדם, כמו `return`, `throw` ואחרות. אם מופיע קוד אחרי פקודה כזו, בדרך כלל מדובר בטעות. בדוגמה הבאה, כל מה שאחרי `return true` לעולם לא ירץ, ו-`ESLint` תdagיש את הקטע הלא נגיש:

```
function validateEmail (email) {
  return true;
  if (email.length) {
    let result = true;
    // if (!email...) {}
    // TODO: complete validation
  }
  return result;
}
```

חולקה לחלקים קטנים

שמות הכללים: `max-len`, `max-lines`, `max-lines-per-function`

קשה לתפוא הרבה לוגיקה ברצף אחד, וכמו שבכתיית ספר או מאמר מומלץ להשתמש בנדיבות בפסקאות, כתורות ועמודים, כך גם בקוד חשוב לא להתפתות להעמיס המון קוד על שורה ארוכה מדי, פונקציה אחת שעשוה הכל או קובץ אחד של אלפי שורות.

החוקים הבאים הם חוקים שאין עליהם מוסכמו, אך הם יכולים לעזור לכם בכך שיישמשו תמרור אזהרה שהקוד שאתה כתובים הפק לאורך ומוסרב. אם תבחרו להשתמש בהם כדאי שתתגלוו להסכמה עם חברי הצוות שלכם בנוגע לגדלים שמקובלים על כולם.

אורך שורה (מספר תוויים מקסימלי בשורה)

יש להזכיר על אורך שורה שאיןנו עוברים מספר תוויים מסוימים. הסיבה לכך היא שרצים שורה שנכנסת ברוחב המסך ללא צורך לגילול הצידה בעת קריאת הקוד. קוד קרייא הוא קוד שאפשר לראות את כלו או את רובו בביטחון אחד. בעבר, כשגודל המסר היה קטן יותר, היה מקובל להגדיר אורך שורה של 80 תוויים, אבל בימינו, כשהמסכים הרבה יותר גדולים, נהוגים אורכי שורה של 120 ואף 140 תוויים.

אפשר להימנע משורות ארוכות בכמה דרכים. הסיבות העיקריות לשורות ארוכות הן רשיימה ארוכה של תנאים וקוד מקוון. רשיימה ארוכה של תנאים קל לשבו לכמה שורות. קוד מקוון יותר קשה לתקן, אך אם דואגים לשמור על פונקציות קצרות ופשטות מצלחים למנוע זאת ברוב המקרים.

אורך פונקציה (מספר שורות מקסימלי בתוך פונקציה)

אין מוסכמה בנוגע לאורך פונקציה למורדות שכולם מסכימים שפונקציות ארוכות זה רע. החוק הזה טוב בעיקר כדי להזכיר לכם שהגזרתם, لكن אם תשתמשו בו כדי לבחור מספר שנותר לכם. אם איןכם בטוחים איזה מספר לבחור, התחילה עם מספר השורות שנכנסות לכם בסיס ללא גלילה.

עוד דרך לשמור על פונקציות קצרות היא לכתוב פונקציות קצרות שיש להן תפקיד אחד בלבד. רמז לכך שפונקציה עשויה יותר מדי אפשר לקבל כתנותו לחת פונקציה שם. אם במהלך הניסיון תגלו בשם הפונקציה את המילה "and", תבינו שניסיתם להכניס יותר מדי לתוך פונקציה אחת. ברגע שזיהיתם מקרה זה, זה הזמן לחלק את הפונקציה לשני חלקים או יותר.

אורך קובץ (מספר שורות מקסימלי בקובץ)

גם כאן אין מוסכמה מקובלת, למרות המתכונטים יסכימו שקובץ שמתקרב ל-1,000 שורות הוא גדול מדי, בעוד אחרים ידברו על מקסימום של 500 שורות או פחות. כאשר הקובץ גדול מדי קל לאלד בו את הידים והרגליים ולכן כדאי להגביל את עצמכם.

אורך מינימלי ומקסימלי לשמות משתנים

שם הכלל: **id-length**

שמות משתנים קצרים מדי או ארוכים מדי הופכים את הקוד ללא קרייה. שמות קצרים מדי מתקשים להבין מה מטרת המשתנה ושמות ארוכים מדי מסרבלים את הקריאה. כשבוחרים שם למשתנה כדאי להתחשב בקיים המנחים שדנוטי בהם תחת הcotraht "בחירה שמות".

בלי **eval**

שם הכלל: **eval-no**

כפי שכבר הוסבר מוקדם יותר בספר, פונקציית `eval` היא פונקציה מסוכנת מבחינה אבטחת מידע. תמיד אפשר וכדאי להימנע משימוש ב-`eval` באמצעות חלופות שונות לפתרון הבעיה.

בלי משתנים שלא הוצאה

שם הכלל: `no-undef`

בג'אווה סקרייפט אפשר להציב ערך לתוכה משתנה שלא הוצאה קודם. במקרה זהה המשתנה ייווצר על האובייקט הגלובלי (`window`, במקרה של ריצה בדפדפן)

```
function greet(name, title) {
  let firstName = name.split(" ")[0]; // משתנה שזמין רק בתחום הפונקציה
  // והירות! המשתנה מוגדר על האובייקט הגלובלי
  prefix = title || "Mr.";
  console.log(`Hello, ${prefix} ${firstName}!`); // "Hello, ser Jaime!"
}
greet("Jaime Lannister", "ser");
console.log(prefix); // "ser"
```

מלבד זיהום האובייקט הגלובלי, דבר זה גם פותח את הקוד לבאים לא צפויים כתוצאה מחוסר תשומת לב ל-`scope` שפועלים בו. מכיוון שהשמה לתוכה משתנה שלא הוצאה מתבצעת כמעט תמיד בטעות, הכלל עוזר לאיתר מקרים כאלו ולהימנע מהם.

לטיכום, כדי ליצור תוכנות איכותיות שהקוד שלهن ברור ונוח לתחזוקה חשוב להקפיד על `Best Practices`, כמו אלו שסקרטרי ברשימה החלקית כאן. הפעילו שיקול דעת בבחירה ובהתאמת הכללים שלפיהם תעבדו, כדי ליצור לעצמכם תהילך עבודה יעיל וקוד ממש טוב לארוך זמן.

נספח: בדיקות, יציבות ואיכות קוד

מחבר: דניאל שטרנלייכט, חברת Outbrain

ברכוט! אם הגיעتم לנספח זהה, נראה לכם כבר מבינים איך ג'אווהסקרייפט עובד ואתם מוכנים להתחיל לבנות אפליקציות ווב. אבל רגע אחד לפני שאתם מתחילהם, רציתי לספר לכם קצת איך לוודא שהקוד שלכם הוא יציב, איקוטי, ויכול לעמוד בבדיקה הזמן כשפתחים שאתם תעבדו יעשו בו שינויים בעתיד.

קצת רקע

באוטבריין, חברת המלצות התוכן הגדולה בעולם שmagisha המלצות למייליארד משתמשים מדי חודש, עובדים יותר מ-200 מפתחים שכותבים אלפי שורות קוד ומחררים שירות גרסאות ביום. האתגר הוא לא קטן, שכן כל שינוי בקוד שפותח עשו ציריך להיבדק ולהיבחן לפני שהוא עובר ללקוחות.

בקנה מידה כל כך גדול, איך מודאים שהקוד חדש שנכתב לא שובר את החוויה שהלקוחות מצפים לקבל? איך אפשרים למפתחים להמשיך לעשות שינויים בקוד בלי ליצור צוואר בקבוק בתהיליך העלאת הגרסאות?

התשובה: בדיקות. המון המון בדיקות. למעשה, לפני כל העלאת גרסה באוטבריין רצוט, באופן אוטומטי, לא פחות מאשר אלף בדיקות!

מבנה בדיקות

כשכותבים בדיקות, אפשר בקלות להציג במצב שבו יש כל כך הרבה עד שהולכים לאיבוד. בדיק בשביל זה קיימים היום כל הספריות והפרוייקטים שעוזרים לסדר את הבדיקות. רוב הספריות מסודרות במתודולוגיה בשם behavior-driven development או **BDD** בקיצור, שמטרתה לסדר את הבדיקות לפי התנהגות כך שייראו פחות או יותר ככה:

```
describe('utils', function () {
  describe('string utils', function () {
    test('should validate strings are strings', function () {
      const foo = 'hey test';
      expect(foo).toBeString();
    })
  })
})
```

הfonקציה **describe** מאפשרת לתאר סדרה של בדיקות מסוימות. תחת הfonקציה הזאת אפשר לקרוא לעוד פוןקציה מסוימת סוג כדי לתאר תת-סדרה. הfonקציה **test** היא הבדיקה עצמה, שתכיל "טענות" (או **Assertions** באנגלית), וfonקציית **expect** היא הטענה עצמה. אם הטענה נफלת, הבדיקה נכשלה.

כ舍דברים על בדיקות, יש לא מעט סוגי: Unit Test, Integration Test, End-to-End Test וכו', וכל בדיקה המטרות שלה.

בנוסף זהה אתמוך בשלושה סוגי בדיקות שונים:

- בדיקות יחידה (Unit Tests)
- בדיקות קצה לקצה (End-to-End או בקיצור E2E)
- בדיקות ממשי משתמש (UI Tests)

בדיקות יחידה

המטרה של בדיקות יחידה (או **Unit Tests** באנגלית) היא לבדוק יחידות קטעות של קוד שעומדות בפני עצמן. למעשה, ככל שהיחידות קטעות יותר, כך טוב יותר. בדיקות יחידה יעבדו מול פונקציות או מול מחלקות, וlorוב יבדקו שקלט מסוים יחזיר פלט מסוים. קחו לדוגמה את הפונקציה הבאה, שידעת לקבל מספר ולהחזיר את המספר מוכפל בעצמו:

```
function multiply(number) {
  return number * number;
}
```

כדי לוודא שהפונקציה תקינה ועושה בדיק את מה שהוא נדרש, מרכיבים כמה בדיקות שונות:

1. בטור התחלת, מעבירים לפונקציה מספר ובודקים שהוא מחזיר את התוצאה הנכונה:

```
expect(multiply(5)).toBe(25);
```

2. בהמשך, מודדים שהפלט המתקיים הוא מהסוג שמצפים לו:

```
expect(typeof multiply(1)).toBe('number');
```

3. אבל מה יקרה אם מעבירים לפונקציה קלט שלא מצפים לו, כמו מחרוזת טקסט למשל?

```
multiply('text');
```

כשכתבתם את הפונקציה לא חשבתם על האפשרות הזה, וכרגע בעקבות הרצאה הזאת היא תחזיר **NaN**, מה שLLLLל לגורם לשגיאות בהמשך. התרחיש הזה, אגב, עלול להתקיים בעיקר בשפות דינמיות כמו ג'אווהסקריפט, שבון משתנים מסוגלים לשנות את סוגם בזמן ריצה. כדי לתקן את הבעיה, אפשר לשדרג את הפונקציה ולהוסיף בדיקה של סוג המשתנה:

```
function multiply(number) {
  if (typeof number !== 'number') {
    throw new Error('Parameter is not a number');
  }
  return number * number;
}
```

4. עכשו אפשר להוסיף בדיקה שמודדת שאם הפרמטר שעובר הוא לא מספר, מצפים לשגיאה:

```
expect(multiply('test')).toThrowError('Parameter is not a number');
```

בדרך כלל רוצים לעשות בדיקות יחידה על יחידות עצמאיות באפליקציה כמו `utilities`, `helpers`, `services`, ודומה, והבדיקות שMRIצים יבדקו שימוש פשוט לצד מקרי קצה – זאת על מנת לכטוט את כל האפשרויות ולהגן עליהם טעויות והתנהגוויות שלא ציפיתם להן

בשימוש בפונקציה. בדיקות ייחודית, נדרש לזכור לשנות אותו אחרי שינויים משמעותיים בקוד.

בדיקות קצרה לקרה (End-to-End)

از יש בבדיקות ייחודית שיפורות לרווח על חתיכות קטנות באפליקציה, אבל זה יכול אחד מהחלקים לעבוד בparelle לא לומר שהם מסונכרנים ועובדים היטב יחד. נוסף על כן, בבדיקות ייחודית לא בודקות תרחישים שבהם מעורבות מערכות אחרות – לצורך העניין בבדיקות ייחודית שרצות על קוד ג'אווהסקריפט בצד הלקוח לא יודעת להגיד אם ממשק המשמשעובד כמו שצריך עם צד השרת.

קחו לדוגמה מפעל לייצור רכב מסווג פורד. יש מחלוקת אחת שאחראית להרכבת המנוע, אחת שאחראית לשולדה ואחת למחשב הרכב. המנוע עובדמצוין, השולדה נראהית מעולה, המחשב עובדמצוין. ההרכבה של המנוע לשולדה עוברת בשלום, אבל כשباءים לחבר את המחשב, מגלים שהוא מיועד לרכיבים של פורמולא 1.

בבדיקה בשליל זה קיימות בבדיקות E2E. בעזרת בדיקות E2E אפשר לבדוק שהאפליקציה עובדת מקרה אחד לקרה שני. בבדיקות E2E יכולים תרחישים שמשלבים מערכות אחרות, ווודאו שהבדיקה מצליחה להגיע ממצב א' למצב ב' ללא הפרעות. לשם הדוגמה אקח אפליקציית חיפוש שכולם משתמשים בה מדי יום: גוגל.

הינה התרחיש שרצים שהבדיקה תבדוק:

1. לר' לאתר google.com.
2. לחץ על אלמנט תיבת חיפוש עם ID בשם "search".
3. הקלד את המילים "אוטבריין" בתיבת החיפוש.
4. לחץ על מקש Enter.
5. בדוק שה-URL הנוכחי הוא google.com/search.
6. בדוק שקיים param query בשם "q" עם הערך "אוטבריין".
7. בדוק שתיבת החיפוש בעלת ID מסווג "search" מכיל את הערך "אוטבריין".

שים לב: הבדיקה נעשית בסביבת דפדפן ומדמה פעילות של משתמש אמיתי באתר אמיתי. הרעיון הוא לבדוק אם האפליקציה עובדת ללא התחשבות במבנה שלה או בטכנולוגיה שעומדת מאחוריה, סוג של "קופסה שחורה".

בדיקות ממשקי משתמש (UI Tests)

בדיקות קצרה לקרה הן מעולות ויודעות לתת מענה לתרחישים שכולים מערכות אחרות, אבל לרוב הן בבדיקות "Happy flow" – מקרים שבהם הכל עובד כמו שצריך. אבל מה לגבי מקרים קצה או מקרים שבהם המרכיבים שהבדיקות תלויות בהן לא עובדות או לא מוחזירות את מה שמשיך המשתמש מצפה לקבל?

הבה ניקח כדוגמה את התרחיש שבודק חיפוש בגוגל. התרחיש יוצא מטור נקודת הנחה שהשרתיים שאחראים על החזרת תוצאות החיפוש עובדים ומוחזרים תשובה מסוימת. אבל מה

יקרה אם תהיה בעיה בשרתים והם לא יחזירו תשובה? או לחולופין הם יחזירו תוצאות ריקות? האם האפליקציה שבניתם תדע להתמודד עם התרחישים הנ"ל ותיראה כמו שאתם מעריכים שהיא תיראה?

אפשר ל כתוב סדרה של בדיקות קצה לenza שעובדות על תרחישים מהסוג הזה, אבל כאמור בדיקות קצה הן יחסית כבדות, והמטרה שלן היא להריץ בדיקות שיצאות מtower נקודת הנחה שהכל עובד.

מפתחי אוטוביין מספרים: נתקלנו בבעיה הזאת ורצינו שתהיה לנו דרך להריץ בדיקות על ממשק המשמש ועל איך הוא מגיב למקורי קצה ולתרחישים מורכבים, אז הוספנו עוד שכבה של בדיקות שבאה לכוסות מקרים מהסוג הזה. אנחנו קוראים לבודיקות האלה "בדיקות ממשק משמש" (UI Tests או UI באנגלית).

כדי להריץ את בדיקות ה-UI צריכים סביבה סגורה שלא תלולה בשרתים (או לפחות לא בשרתים אמיתיים), אז בנוינו כל' בשם "לאונרדו" שידוע להזדהות בקשות לשרת, להשלט عليهן ולדמota תשובה שאנונו מגדירים מראש.

כך למשל אפשר לראות איך האפליקציה מתמודדת עם תשבות ריקות, עם שגיאות שmagiות מהשרת ועם מצבים שבהם השרת לא מגיב. כמו כן אפשר לבדוק מצבים שבהם התגובה מהשרת איטית.

בדיקות מהסוג הזה הן מהירות, יציבות ויעילות מאוד משומשים להן את יכולת לבדוק כל תרחיש שרצים ללא תלות במערכות אחרות.

הכל' לאונרדו הוא כל' open source וזמן לכולם בגיטהאב תחת הכתובת:

<https://github.com/outbrain/Leonardo>

ספריות ופרימורקים מומלצים

יש לא מעט ספריות ופרימורקים לכתיבת בדיקות בג'אווהסקריפט, הינה כמה שאנונו: באוטוביין, משתמשים בהם וממליצים לכם בחום לנסוטו:

- **Jest** – ספריית הבדיקות של פיסבוק. נותנת מענה לרוב סוגי הבדיקות ובעלת API אינטואיטיבי לכתיבת הבדיקות ולהתמודדות עם שגיאות. עובדת מעולה עם ספריות מודרניות כמו `mar`, `React`, `Angular`, `-Vue`.
- **Jasmin** – ספרית בדיקות ותיקה ומאוד פופולרית. טובת מאוד בבדיקות יחידה ועומדת בפני עצמה ללא תלות במערכת כלשהי.
- **Karma** – כל' פשוט שיאפשר לכם להריץ קוד ג'אווהסקריפט ובקביל לבצע בדיקות בדפננים.

סיכום

למדתם מה אומר המושג "בדיקות" בכתיבת קוד, ואיך הוא יעזר לכם לשמר על יציבות ועל קוד נקי. ראייתם דוגמאות לבדיקות יחידה ובדיקות קצה לקצה והבנתם למה הן כל כך חשובות. וגם המלכנו לכם על ספריות ופרוייקטים שייעזרו לכם לכתוב בבדיקות.

עוד משהו קטן שווה לציון: בקבוצות פיתוח מסוימות לקחו את עניין הבדיקות רוחק יותר ואימצו מתודולוגיה בשם "פיתוח מונחה-בדיקות" או באנגלית **Test-Driven Development** (בקיצור TDD). לפי המתודולוגיה הזאת, בדיקת יחידה תיכתב עוד לפני כתיבת הקוד שאוטו היא בודקת. קצת על הקצה, אבל אם תחילתו לכתב עם המתודולוגיה הזאת, אתם יכולים להיות בטוחים שהקוד שתכתבו יעבד בצורה חלקה.

נספח: Velo by Wix

מחברת: מור גלעד, חברת Wix

הקדמה

Velo היא פלטפורמת פיתוח אפליקציות שנבנתה על העורק הוויזואלי של Wix. העורק הוויזואלי של Wix נותן למפתח את יכולת לייצר ממשק ויזואלי מפותח מאוד מוביל להשתמש ב-HTML או ב-CSS. הוא מכיל מאות רכיבים חזותיים (elements) שאפשר לעצב אותם באינסוף אפשרויות. בעזרתו אפשר ליצור אתרים מרהיבים.

הפלטפורמה של Velo מאפשרת:

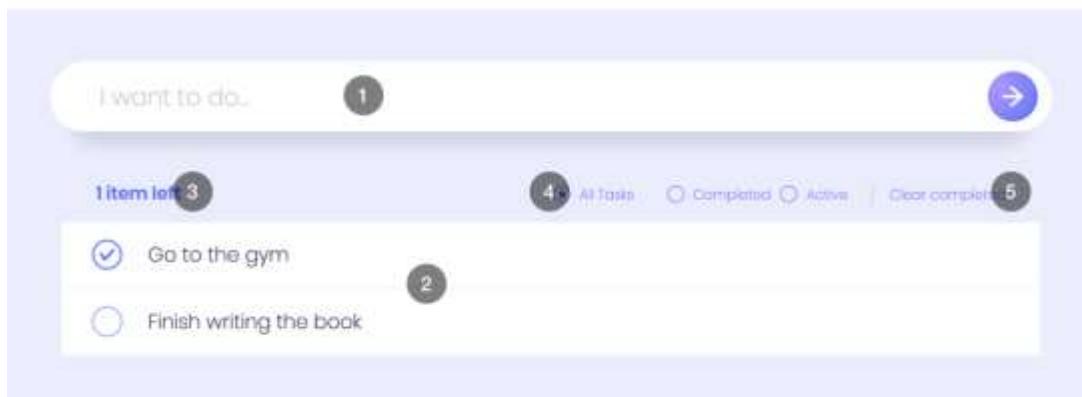
- לכתוב קוד בג'אווהסקריפט עבור הפעולות הנדרשות של האתר
- לשנות פועלות והתנהגות של רכיבים באתר
- לייצר מבני נתונים

מה בונים?

בפרק הנוכחי אלמד אתכם בצעדים קטנים איך לבנות אפליקציה ניהול משימות באמצעות Velo.

אפשר לראות את האפליקציה שנבנה בכתובה <https://www.wix.com/velo-dev/todo-app>

- הקוד בספר נכתב במטרה ללמידה ולהמחיש בצורה ברורה ופושטה ולאו דווקא לגרום למערכת לעבוד בצורה אידיאלית ויעילה.



תיאור כללי של אפליקציית המשימות:

1. תיבת טקסט (text input) וכפתור ההוספה המאפשרים למשתמש להוסיף משימה חדשה.
 2. רשימת המשימות — עבור כל משימה רואים את הסטטוס שלה, אם היא הושלמה או לא.
 3. מספר הרשימות שנוטרו לבצע.
 4. רכיב עם כפתורי בחירה (Radio group) המשמשים לסינון המשימות. אפשר לראות את כל המשימות (All Tasks), את המשימות שהושלמו (Completed) או את המשימות שעדיין לא הושלמו (Active).
 5. כפתור למחיקת כל המשימות אשר הושלמו. בלחיצה על הכפתור תופיע למשתמש חלונית שתשאל אותו אם הוא מעוניין לבצע את הפעולה. אם המשתמש יסכים, כל המשימות אשר הושלמו – יימחקו.
- במהלך הפרק תשתמשו בהרבה פונקציות ש-Velo מספקת עבור מתכנתים ואסביר את כולן. אם ברצונכם להרחיב את הידע שלכם וללמוד פונקציות נוספות מတואות בפרק זהה, אתם מוזמנים לבקר באתר המידע של Velo reference <https://www.wix.com/veo/reference/> עבור כל פונקציה שאלמד פה, אוסיף הפניה לאזור שבו היא מוסברת באתר המידע.

AIR מתחילה?

הכניינו עבורכם תבנית מעוצבת מראש שמננה תוכלו להתחילה => <https://www.wix.com/website-template/view/html/2186>

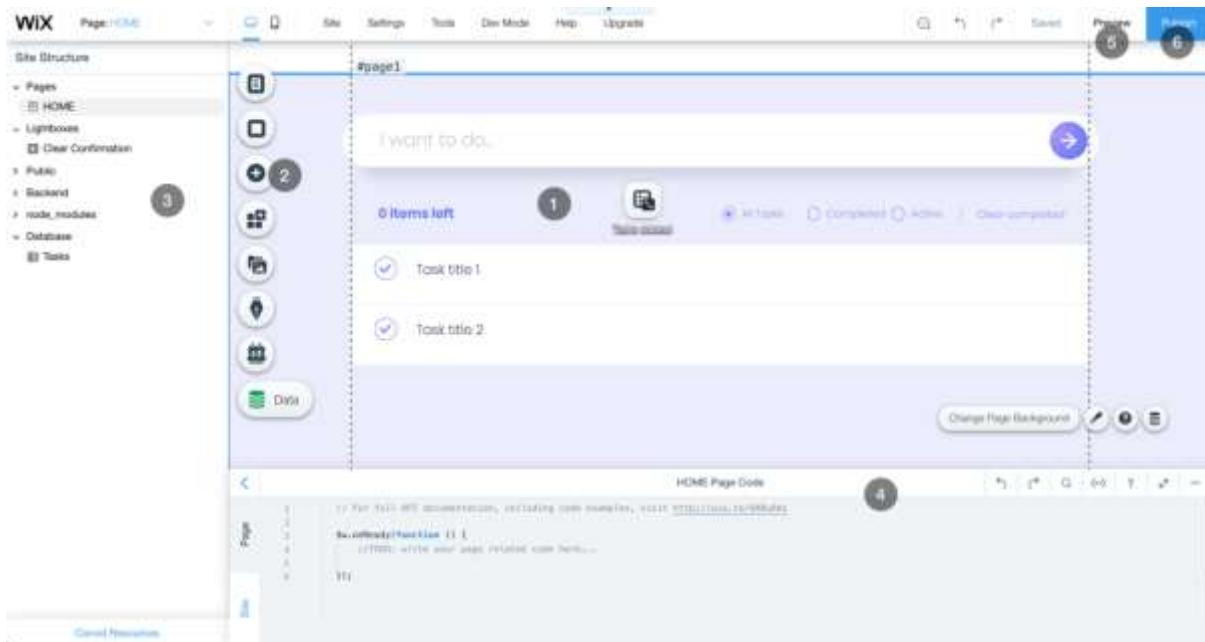
העורק הוויזואלי של Wix עשיר ביכולותיו ויש הרבה מאוד מה ללמידה עליו. אבל מכיוון שאתם לומדים כרגע ג'אווהסקריפט, רציתי לחסוך לכם את זמן העיצוב והבניה של הרכיבים באפליקציה על ידי שימוש בתבנית ולהשאיר לכם רק את החלק המהנה – כתיבת הקוד. כמובן שתאתם יכולים לשנות את העיצוב ולהוסיף רכיבים חדשים.

לאחר שנכנסתם ל התבנית, לחזו על  , הירשמו ל-Wix באמצעות כתובת האימייל שלכם, וcutם נמצאים בעורק הוויזואלי של Wix – ברוכים הבאים 😊 כדי להפעיל את Velo, עברו בעזרת העכבר מעל Dev Mode, בתפריט שנמצא בחלק העליון

 Turn on Dev Mode

של האתר ואז על

אחרי שהפעילם את Velo, תראו את המסר הבא:



אך מה בעצם רואים פה?

1. עורך ויזואלי. האזור שבו עורכים את הצד הוויזואלי של האתר. אל אзор זה מוסיפים רכיבים ומשנים את העיצוב שלהם. אפרט לגבי הרכיבים בהמשך.
2. הכפטור +, להוספה רכיבים חדשים לאתר. תפירט הרכיבים מכיל מאגר של מאות רכיבים שאפשר להוסיף לאתר על ידי גירירה.
3. בניית האתר | **site structure**. פannel שמציג את כל העמודים, חלונות (lightbox), קובצי הקוד ומסדי הנתונים שיש באתר.
4. בדוגמה הנוכחית יש כרגע עמוד אחד שנקרא **HOME**, חלונית שנקראת "Clear Confirmation" ומסד נתונים המכיל טבלה שנקראת **Tasks**.
5. סבירת הפיתוח. המקום שבו כתבים את הקוד. אפשר לכתוב קוד עbor כל עמוד באפליקציה. עברו כל עמוד חדש באפליקציה, מתחילה עם תבנית קוד מוכנה המכילה `$(w.onReady)` – שדבר עליה בהמשך.

- * יכול להיות שסבירת הפיתוח שלכם תהיה מצומצמת בשורה התחתונה. לחצו על כדי להרחיב אותה.
- 6. כפטור הצג | **preview**. מציג את האתר שעבדתם עליו עד עכשיו. לרוב, משתמשים בכפטור זהה תוך כדי עבודה על האפליקציה לפני שורצים לפרסום אותה לעולם. במהלך הפרק, עשו עזרות מדי פעם כדי לראות מה עשיתם עד עכשיו. בuczירות האלו אבקש שתלחצו על **preview**.
- 7. כפטור פרסום | **publish**. מפרסם את האתר שנבניתם עד עכשיו וחשוף אותו לכל העולם.

התבנית שעלייה אתם בונים את האפליקציה מגיעה מוגעה עם מסד נתונים, שבו טבלת **Tasks** שמכילה משימות. תלמדו איך להוסיף לטבלה זהה משימות חדשות, לשנות את סטטוס המשימה, למחוק משימות ועוד כל מיני פעולות שיעזרות לנויל מידע.

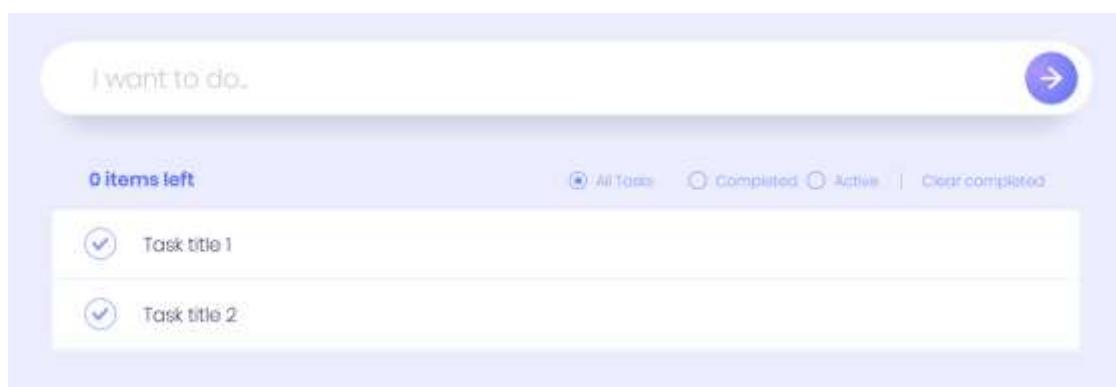
אתם יכולים לראות את המשימות שנמצאות בטבלה על ידי ללחיצה על Tasks ב-site structure.

כל שורה מייצגת משימה וכל משימה יש שני מאפיינים:

- **Title** – תיאור המשימה
- **Completed** – משתנה בוליאני המתאר אם המשימה הושלמה או לא

אם אתם רוצים למדוד איך ליצר את הטבלה בעצמכם, קפצו לסוף הפרק, לנוסף שבו הוספה לכם פירוט ורחבת על בניית טבלאות.

לפני שמתחלים לכתוב קוד, לוחצים על preview ומטבוננים בנקודות הפתיחה:



כפי שאפשר לראות, כל הרכיבים מוכנים ומעוצבים מראש אבל אף כפטור לא מגייב ללחיצה ומספר המשימות שנותרו אינם נכון.

כדי להמשיך לעבוד על האפליקציה, עכשו לוחצים על

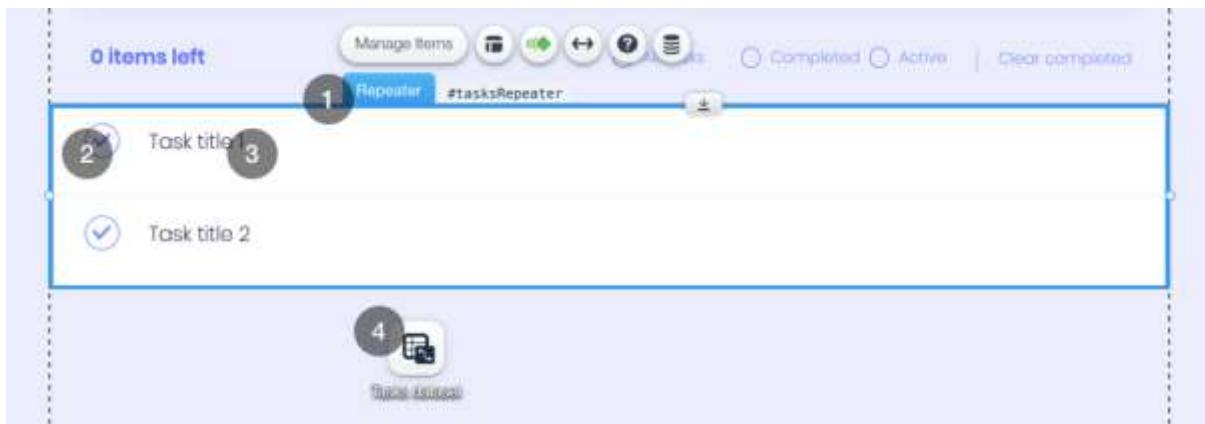
[Back to Editor](#)

הציג נתוניים ממוד הנתונים באתר

בחלק זה אלמד איך אפשר להוסיף נתונים מהטבלה על ידי "חיבור" רכיבים מהאפליקציה לטבלה Tasks שלנו. כך חוסכים כתיבת קוד טריוויאלית. מובן שnitן לעשות את זה גם

באמצעות קוד – אבל אל דאגה, תכתבו הרבה שורות קוד ממש בקרוב 😊

מתחלים בהכרת הרכיבים שיקחו חלק בהציג המשימות שבטבלה Tasks של האפליקציה.



1. **Repeater** – רכיב שיצר בתוכו כמה עותקים של רכיב אחר. כל עותק מכיל עיצוב זהה אך מידע משתנה. מספר העותקים שייצרו הוא משתנה ונקבע לפי מספר הפריטים שעלו להציג. במקורה זהה, מספר המשימות שהוא יציג.

[https://www.wix.com/velo/reference/\\$w.Repeater.html](https://www.wix.com/velo/reference/$w.Repeater.html)

2. **Checkbox** | תיבת סימון – רכיב המאפשר הכנסת ערך בוליאני (true | false).

[https://www.wix.com/velo/reference/\\$w.Checkbox.html](https://www.wix.com/velo/reference/$w.Checkbox.html)

3. **Text** | רכיב טקסטואלי – רכיב שמציג טקסט.

[https://www.wix.com/velo/reference/\\$w.Text.html](https://www.wix.com/velo/reference/$w.Text.html)

4. **Dataset** – הרכיב שמקשר בין המידע בטבלת Tasks לבין מרכיבים אחרים בעמוד.

<https://www.wix.com/velo/reference/wix-dataset.html>

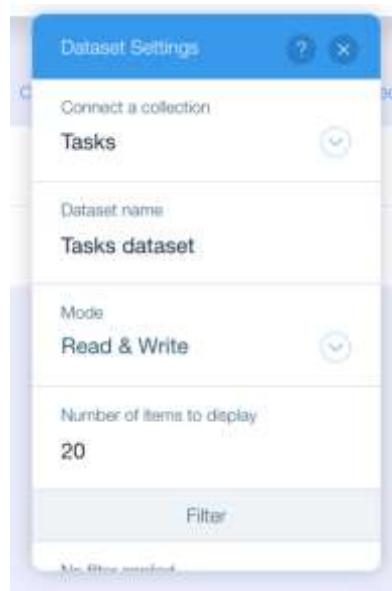
Dataset

ה-component dataset מחבר רכיבים בעמוד למידע מטבליות נתוניים. רכיב זה הוא וירטואלי, כלומר, הוא יעשה את תפקידו בניהול מידע בעמוד אבל מבקרים העמוד לא יראו אותו באפליקציה הסופית.

אפשר להסתכל על ההגדרות של ה-component dataset בעזרת לחיצה על כפתור ה-settings:



از נראה שה-component dataset מחובר לטבלת Tasks וונמצא במצב של קרייה וכתיבה (& write).



הוא צריך להיות מוגדר במצב של קרייה וכתיבה - **Read & Write**, מכיוון שתרצו אפשרות להציג את המשימות אבל גם לשנות את סטטוס ה-*completed* שלהן.

יש עוד שני מצבים ל-**dataset**:
Read Only – מצב שמאפשר רק קריאה של נתונים מהטבלה
Write Only – מצב שמאפשר רק הוספה של נתונים חדשים לטבלה

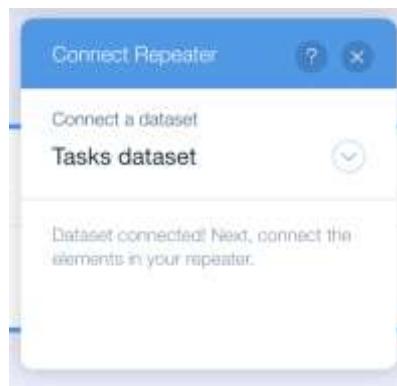
נוסף על כך, **Number of items to display** מהווה את כמות הרשומות המקסימלית שה-**dataset** ישולף מהטבלה. אם אתם רוצים לתרום בցהה של יותר מ-20 משימות, אתם יכולים לשנות את זה בהגדרות.

חיבור רכיבים למידע מהטבלה

כעת, משתמשים ב-**dataset** על מנת להציג את המשימות בכל שורה ב-**repeater**. על מנת לעשות את זה לוחצים על האיקון של מסד הנתונים, שMOVED כאשר לוחצים עם העכבר על ה-**repeater**.

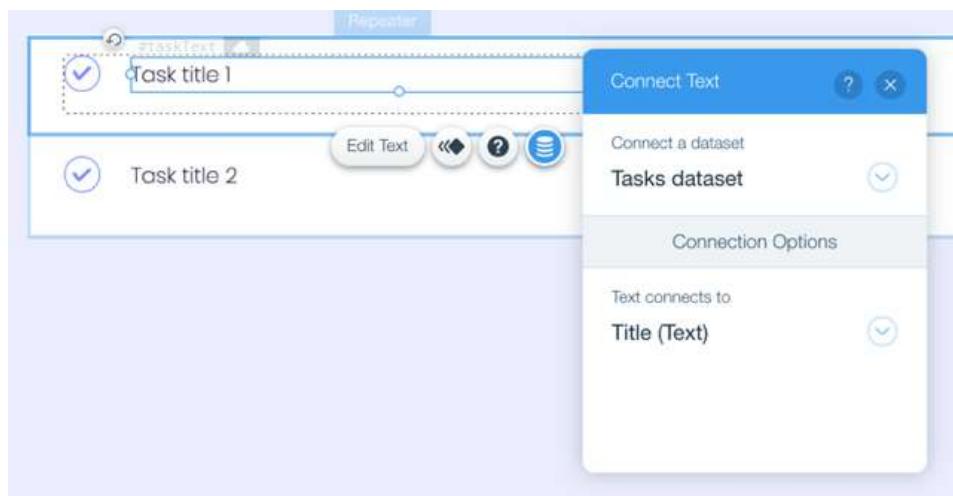


כעת נפתח פannel החיבור (connection panel) של ה-**repeater**. מכיוון שאתם רוצים לחבר את ה-**repeater** ל-**dataset** שיצרתם, בחרו אותו ב-**connect a dataset**.



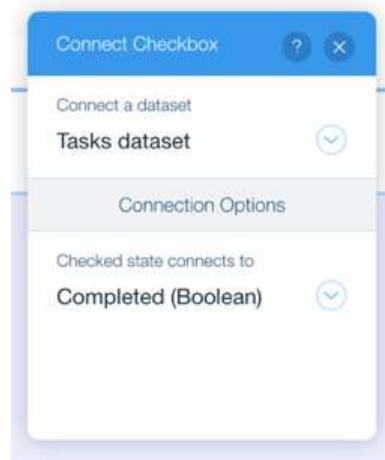
התוצאה של הפעולה זו היא שמספר הקופסאות ב-repeater ישוכפלו כמספר הרשומות שה-
dataset שלפיה מובלט המשימות.

הצעד הבא הוא להציג כל משימה בשני הרכיבים שיש בתוך ה-repeater. מתחילה מרכיב הtekst. פותחים את פאנל החיבור של רכיב הtekst (זה דרוש לחיצה ראשונית על ה-repeater ולאחר מכן לחיצה על רכיב הtekst) ומחברים את הtekst של הרכיב לשדה Title מובלטת ה-Tasks.



- אולי שמתם לב שה-dataset שיצרתם כבר מחובר. זה קורה באופן אוטומטי מכיוון שחברתם את ה-repeater ל-dataset זהה.
- ברשימה השדות שנפתחת יש שדות נוספים שאפשר להתחבר אליהם. אלו שדות שנוצרים באופן אוטומטי עבור כל רשומה בטבלה:
 - ID
 - createdDate
 - updatedDate

לאחר מכן, פותחים את פאנל החיבור של checkbox של ה-repeater (גם זה ידרוש לחיצה על ה-repeater) ולאחר מכן לחיצה על ה-group שמכיל את הcheckbox, ולבסוף על הcheckbox (checkbox על ה-repeater) ומחברים את הערך checked לשדה completed מהטבלה.



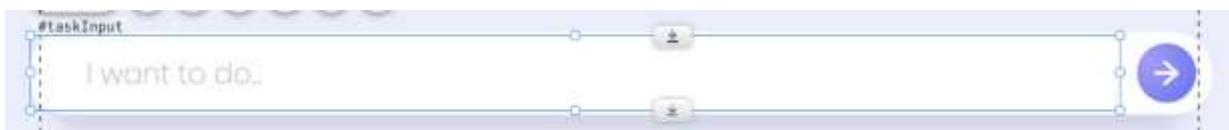
אחרי שהיברתם את הרכיבים שלכם לטבלה, הגיע הזמן לראות את התוצאות!
לחצו על preview ווסתכלו על התוצאות. התוצאות הרצויות הן:



רואים את המשמעות? יש!
עכשו הגיע הזמן להתחיל לכתוב קוד – חגיגה!

הוספת משימה חדשה

הרכיבים שיעזרו לכם להווסף משימה חדשה:



1. **תיבת טקסט | text input** – אפשרותה להכניס את תיאור המשימה.
[https://www.wix.com/velo/reference/\\$w.TextInput.html](https://www.wix.com/velo/reference/$w.TextInput.html)

2. **כפתור | Button** – בלחיצה עליו תתווסף המשימה מתיבת הטקסט לutable Tasks.
[https://www.wix.com/velo/reference/\\$w.Button.html](https://www.wix.com/velo/reference/$w.Button.html)
 אם רצים לאפשר למשתמשים להווסף משימה חדשה, יש ללמידה איך בלחיצה על הכפתור מוסיף משימה חדשה לתוך מסד הנתונים ומציגים אותה ב-repeater.

Events

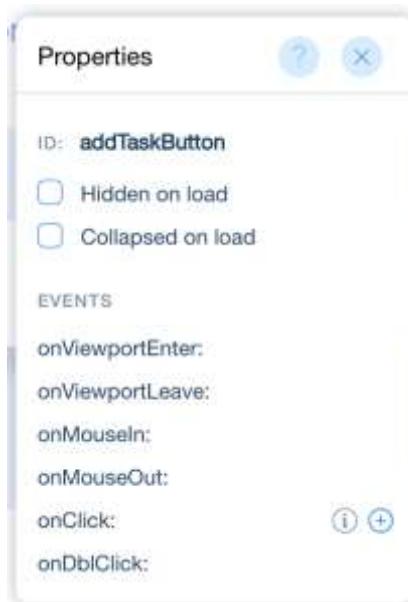
[https://www.wix.com/velo/reference/\\$w.Event.html](https://www.wix.com/velo/reference/$w.Event.html)

בדומה לארועים בג'אווהסקריפט, גם הרכיבים של Velo מאפשרים הפעלה של אירועים (events).

אלמד אתכם שני אירועים על שני רכיבים שונים:

- **button.onClick event** - שנקרו כאשר המouser באפליקציה לוחץ על כפתור.
[https://www.wix.com/velo/reference/\\$w.Button.html#onClick](https://www.wix.com/velo/reference/$w.Button.html#onClick)
- **textInput.onKeyPress** - שנקרו כאשר המשתמש לוחץ על כפתור במקלדת בזמן שהוא נמצא בתוך ריבית ה-text.
[https://www.wix.com/velo/reference/\\$w.TextInput.html#onKeyPress](https://www.wix.com/velo/reference/$w.TextInput.html#onKeyPress)

מתחלים בהוספת event לכפתור . עושים זאת על ידיפתיחה פאנל ה-properties של הכפתור, בקליק ימני על הכפתור ולחיצה על View Properties.



פאנל ה-`properties` מחולק לשני חלקים:

בחלקו העליון אפשר לראות את ה-ID של הרכיב. בעזרתו ה-ID אפשר לבצע פעולות עם הרכיב דרך הקוד. בתבנית שהכניתו לכם, נתתי מראש לכל רכיב ID שם שימושי, אבל אתם יכולים לשנות את זה.

נוסף על כך, מתחת ל-ID מופיעות הגדרות הנראות של הרכיב בזמן העלייה של העמוד.

בחלקו התיכון של הפאנל נמצאת רשימה של כל ה-`events` הנתמכים עבור הרכיב:

- `onViewportEnter` מופעל כאשר הcpfotor מופיע בחלון של הדף
- `onViewportLeave` מופעל כאשר הcpfotor נעלם מחלון הדף
- `onMouseIn` מופעל כאשר המשתמש מעביר את העכבר מעל הcpfotor
- `onMouseOut` מופעל כאשר העכבר יוצא מגבולות הcpfotor
- `onClick` מופעל בלחיצה על הcpfotor
- `onDoubleClick` מופעל בלחיצה כפולה על הcpfotor

* לכל רכיב יש events שונים בהתאם לפונקציונליות שהוא מספק. ניתן לראות הסבר מדויק על כל ה-`events` של הcpfotor בlienק:

[https://www.wix.com/velo/reference/\\$w.Button.html](https://www.wix.com/velo/reference/$w.Button.html)

במקרה זה רוצים להוסיף משנה כאשר המשתמש לוחץ על הcpfotor. לכן, עוברים עם העכבר על הצד ימני של event `onClick` וЛОוחצים על כפטור ה- . ללחיצה על הcpfotor תכניס באופן אוטומטי שם ל-`event` `onEnter` ובלחיצה על `Enter` יתווסף event חדש לקוד בסביבת הפיתוח שלכם.

```
export function addTaskButton_click(event) {
    //Add your code for this event here:
}
```

עכשו, בכל לחיצה על הכפתור, יקרא ה-event שלכם. הנה נראה שזה אכן נקרא בכל לחיצה.
מוסיפים כתיבה ל-console:

```
export function addTaskButton_click(event) {
    console.log('button clicked');
}
```

שימוש לב שהפונקציה שלכם מקבלת ארגומנט שנקרא event. הארגומנט זהה הוא אובייקט המכיל מידע על ה-event שנקרה. האובייקט הזה משתנה בהתאם לסוג ה-event.

* מעתה כתבו ותעתיקו הרבה שורות קוד. אם אתם רוצים שהקוד יעבור הרצה באופן אוטומטי, לחזו על הכפתור , שנמצא בצד ימני של החלק העליון בסביבת הפיתוח.

עוברים ל-view, לוחצים על הכפתור ובחתית הדף יפתח ה-console של Velo ויציג את מה שכתבתם בכל פעם שתלחצו.
עובד לכם? מעולה! ממשיכים להלאה ושמרים את המשימה.

\$w

[https://www.wix.com/velo/reference//\\$w.html](https://www.wix.com/velo/reference//$w.html)

על מנת לשמר את המשימה שהמשתמש כתב, יש "להוציא" את הטקסט שנכתב מתוך רכיב ה-input text. את זה עושים בעזרת שימוש בפונקציה **w**.

w היא פונקציה שזמיןה בקוד ומאפשרת לבחור רכיבים מתוך העמוד באמצעות הקוד, בדומה לפונקציה `document.querySelector`. לכל רכיב סט מאפיינים ופונקציות ייחודי המאפשר אינטראקציה עם הרכיב, לקבל ולשנות את המידע שלו, לשנות את מצבו ולהירשם ל-events שהוא חושף.

איך בוחרים רכיב בעזרת **w**?

זכירים את ה-ID שיש לכל רכיב בפאנל properties? אז על מנת לקבל את המשתנה של הרכיב משתמשים ב-ID שלו. אז, כדי לבחור אותו, קוראים ל-w עם ארגומנט מסווג טקסט, המכיל את ה-ID של הרכיב ומתחיל ב-#.

לדוגמה, אם תפתחו את פאנל properties של תיבת הטקסט שלכם, תראו שה-ID שלה הוא `taskInput`, וaz תבחרו אותה כך:

```
$w('#taskInput').value
```

אחרי שיש תיבת טקסט, רוצים לדעת מה הטקסט שהמשתמש הכניס לתוכה. לכל רכיב יש מספר רב של נתונים שהוא יכול לספק. במקרה הזה צריכים את ה-`value` שהמשתמש יכניס לתיבת:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  console.log('taskTitle:', taskTitle);
}
```

למידע נוסף על משתנה ה-`value` של תיבת הטקסט:

[https://www.wix.com/velo/reference/\\$w.TextInput.html#value](https://www.wix.com/velo/reference/$w.TextInput.html#value)

עוביים ל-`width`, כתובים את המשימה בשורה המשימה, לוחצים על הפתור והידד! רואים את הטקסט שכתבתם.



הצעד הבא => לשמר את המשימה כמשימה חדשה בטבלה ה-`Tasks`.

wix-data

<https://www.wix.com/velo/reference/wix-data.html>

wix-data הוא מודול המאפשר עבודה מול המידע שיש לכם בטבלאות. בעזרתו, אפשר להוסיף רשומות לטבלה, למחוק רשומות וגם לעורוך רשומות קיימות. במהלך הפרק אלמד בכל פעם שימוש ופונקציונליות חדשה של `wix-data`.

על מנת להשתמש במודול, יש לייבא אותו מהאזור העליון של קובץ הקוד:

```
import wixData from 'wix-data';
```

הוספה רשומה לטבלה – `wixData.insert()`

<https://www.wix.com/velo/reference/wix-data.html#insert>

מתחלים עם היכולת להכניס משימה חדשה לטבלה. זה אפשרי באמצעות הפונקציה `האיסינקורניט()` לפונקציה זו מعتبرים שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשימה החדשה – במקרה הזה – `Tasks`.
2. המשימה שאויה רוצים להוסיף. המשימה תהיה מבנה של אובייקט, שבו המשתנים יהיו זהים למאפיינים של המשימה בטבלה והערכים שלהם.

از על מנת להוסיף משימה חדשה לטבלה, מייצרים את אובייקט המשימה שיכיל:

- **title** – טקסט המתאר את המשימה, שנלקח מהתיבת הטקסט

false – משתנה בוליאני המציג את סטוס המשימה ויאווחל ב- **completed** -

קוראים ל-() `wixData.insert` עם האובייקט שיצרתם. בנוסף על כן, משלימים את החוויה בכך שמוסיפים את המשימה החדשה רק במידה שהיא לא ריקה:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }
    await wixData.insert('Tasks', newTask);
  }
}
```

* שימוש לב שמיון שהפונקציה () היא אסינכרונית, מוסיפים `async` ל- `event`.

עכשו קופצים שוב ל- `preview`, מוסיפים משימה חדשה, לוחצים על `Enter`, חוזרים ל- `editor` של `Wix` וועברים לעורך התוכן של טבלת `Tasks` כדי לראות אם המשימה החדשה התווספה. התווספה? **מעולה!**

	Title	Completed
1	new task	
2	Real task 1	
3	Real task 2	
4	Real task 3	

אבל אני מניח ששפתם לב שencersים כמה דברים כדי שהחוויה תהיה שלמה:

1. שורת המשימה לא התרוקנה אחרי הוספת המשימה.
2. המשימה לא התווספה לרשימת המשימות שלכם.

צינתי קודם שבעזרת `W` אפשר לקבל גם לשנות מידע על רכיב באתר. אז הפעם, במקומם קיבל את הערך של תיבת הטקסט שלכם, יש לפחות אותה כך:

```
$w('#taskInput').value = '';
```

ועל מנת שה- `repeater` יציג את הערכים החדשניים, יש לרענן את המידע שה- `dataset` מעביר אליו. זה לא קורה באופן אוטומטי.

כמו שכל רכיב רגיל באתר ניתן לשינוי, כך גם ה- `dataset`. לכן מבקשים מה- `dataset` לקרוא מחדש את המידע מהטבלה בעזרת קריאה לפונקציה האסינכרונית () `refresh`:

```
await $w('#dataset1').refresh();
```

למידה נוספת על פונקציית `refresh` של `dataset`:

<https://www.wix.com/velo/reference/wix-dataset.Dataset.html#refresh>

זה הקוד המלא:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }

    await wixData.insert('Tasks', newTask);
    $w('#taskInput').value = '';
    await $w('#tasksDataset').refresh();
  }
}
```

ושוב, עוברים ל-`preview` ומכניסים משימה חדשה. איזה כיף! זה עובד!

תרגילים:

בצעו שמירה של משימה חדשה כאשר המשתמש כותב משימה חדשה בתיבת הטקסט ואז מקליד על `Enter`.
רמז: הזכרתי קודם את ה-`event` על תיבת הטקסט – `textInput.onKeyPress` ואת המשמעות של הארגומנט `event` שמקבלים בפונקציה.

פתרונות:

לאחר פתיחה של פאנל ה-`properties` וaz יצרה של ה-`event` `onKeyPress` event יתווסף נסוף לסייעת הפיתוח. אבל הפעם יהיה צורך לוודא שהכפטור שעליו המשתמש לחץ הוא כפטור `Enter`, ורק אז לשמר את המשימה ולבצע את כל הפעולות הנדרשות.

```
export async function taskInput_KeyPress(event) {
  if (event.key === 'Enter') {
    // add new task here..
  }
}
```

מכיוון שלא מומלץ לשכפל קוד, מוצאים את פעולה ההוספה של משימה חדשה לפונקציה נפרדת, ולבסוף הקוד יראה כך:

```

async function addNewTask() {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }

    await wixData.insert('Tasks', newTask);
    $w('#taskInput').value = '';
    await $w('#tasksDataset').refresh();
  }
}

export async function addTaskButton_click(event) {
  await addNewTask();
}

export async function taskInput_KeyPress(event) {
  if (event.key === 'Enter') {
    await addNewTask();
  }
}

```

* גם פה ה-`taskInput_KeyPress` הפען לאсинכרוני ונוסף לו `.async`.

עוברים שוב ל-`view` ומוכנים ממשימות חדשות באמצעות לחיצה על `Enter` וגם בעזרה לחיצה על כפתור ההוספה.

שינוי סטטוס המשימה

از אפשר לראות את המשימות הנוכחיות ולהוסיף משימות חדשות. אבל המשתמש עדין לא יכול לעדכן שהמשימה הושלמה.

בכל פעם שהמשתמש משנה את הסימון בכפתור הבחירה (checkbox), שנמצא בתוך ה-`repeater`, נרצה לעדכן את הבחירה החדשה שלו בטבלה עבור המשימה הרלוונטית. מתחילה בהוספת `event` של ה-`checkbox` דרך `onChange` פannel ה-`checkbox`, כמו שמלמדתם קודם:

```

export function completedCheckbox_change(event) {
  //Add your code for this event here:
}

```

למידע נוסף על ה-`onChange` event של כפתור הבחירה:
[https://www.wix.com/velo/reference/\\$w.Checkbox.html#onChange](https://www.wix.com/velo/reference/$w.Checkbox.html#onChange)

* מכיווןשהcheckbox נמצא בתוך repeater, ה-event שמנמש על checkbox אחד יופעל אוטומטית על כל אחד ממהcheckbox שיש בכל איבר ב-repeater. כלומר אין צורך לכתוב event לכל checkbox אם יש לנו repeater.

כדי לשנות את הערך הבוליאני של מאפיין completed של המשימה לערך החדש שהמשתמש סימן ב-checkbox, קודם כל ציריך להציג את הערך החדש, ועושים זאת על ידי שימוש במשתנה ה-target, שמנמצה על ארגומנט event שמועבר לפונקציה:

```
export function completedCheckbox_change(event) {
  const completed = event.target.checked;
}
```

למידע נוסף על משתנה ה-target של event:

[https://www.wix.com/velo/reference/\\$w.Event.html#target](https://www.wix.com/velo/reference/$w.Event.html#target)

לאחר מכן נרצה לעדכן את הערך החדש בטבלה.

wixData.get() – שילוף רשומה מהטבלה

<https://www.wix.com/velo/reference/wix-data.html#get>

על מנת לייצר את המשימה שאותה רוצים לעורך, יש לשולף את הרשומה הזו מהטבלה כדי שכל הנתונים הנוכחיים שלה יהיו זמינים.

למודול wixData יש פונקציה אסינכרונית הנקראת **get()**, שמאפשרת שילוף של משימה מהטבלה. לפונקציה הזו מעבירים שני ארגומנטים:

- שם הטבלה שמננה רוצים לשולף את המשימה – **Tasks**.
- הזהה הייחודי של המשימה – **ID**.

לכל משימה שמכניסים לטבלה מתווסף באופן אוטומטי מזהה שנקרה **id**. המזהה הזה מאפשר לזהות באופן ייחודי את המשימה.

از איר בעצם מישגים את המזהה הייחודי הזה מתוך המשימה ב-repeater שעליינו לחץ המשמש?

בעזרת ארגומנט event.כפי שתכתבו קודם, ארגומנט event-המשנה בהתאם להתקף event-שנקרא. אבל יש חשיבות למיקום הרכיב שבו מתרחש ה-event ובמיוחד אם הוא בתוך ה-repeater. אם ה-event הופעל והרכיב שהוא הופעל ממנו נמצא בתוך repeater-ה-ID אז נקבל באובייקט event-המשנה נוסף, שAIN ב-event רגיל, שנקרא context ומכיל את ה-ID של הרשומה שצריכים מהטבלה:

```
event.context.itemId
```

למידע נוסף על משתנה ה-context של event:

[https://www.wix.com/velo/reference/\\$w.Event.html#context](https://www.wix.com/velo/reference/$w.Event.html#context)

וכעת, כיש בידיכם המזהה הייחודי של הרשומה, אפשר לקרוא ל-(`wixData.get`) ולקבל את הרשומה הנוכחית:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;
  const item = await wixData.get('Tasks', itemId);
}
```

* שוב, יש להפוך את ה-`event` שלכם לאסינכרוני כיוון ש-(`wixData.get`) היא אסינכרונית.

עדכון רשותה בטבלה – `wixData.update()`

<https://www.wix.com/velo/reference/wix-data.html#update>

יש לכם רשותה, כפי שהיא שמורה בטבלה. בעצם, יש לשנות את ערך ה-`completed` שלה לערך החדש ולעדכן אותו בטבלה.

למודול `wixData` יש פונקציה `update()` שמאפשרת עדכון של רשותה מסוימת. הפונקציה הזאת מקבלת שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשותה החדשה – `Tasks`.
2. המשימה המעודכנת שרוצים לשמר.

מיצרים משימה מעודכנת ואז קוראים ל-(`wixData.update`) עם העדכון:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;

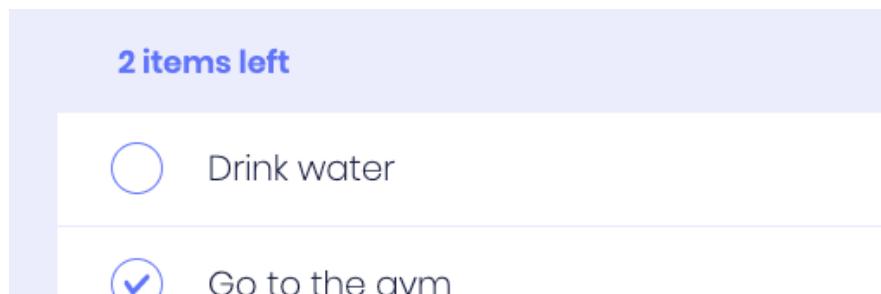
  const item = await wixData.get('Tasks', itemId);
  const updatedItem = Object.assign({}, item, { completed });

  await wixData.update('Tasks', updatedItem);
}
```

הבה נבדוק אם זה עובד ב-`Wix Preview`. מסמנים כמה משימות ו חוזרים לעורך התוכן לבדוק אם זה באמת עדכן.

נמדד! זה עודכן! הפונקציונליות הבסיסית של האפליקציה עובדת.

מספר המשימות שלא הושלמו



היכולת הבאה שורצים להויסיף לאפליקציה היא תצוגה של מספר המשימות שלא הושלמו (שהערך הבוליאני שלhn במאפיין ה-completed הוא false) ברכיב הטקסט שמוינו מעל רישימת המשימות.

מתחילה במבנה פונקציה אסינכרונית חדשה בחתית קוד (מחוץ לפונקציה זה-
(onReady), שתפקידה יהיה לעדכן את רכיב הטקסט במידע הרצוי. שמה:
updateActiveTaskCount

```
async function updateActiveTaskCount() {  
}
```

הצעד הבא הוא לספור את מספר המשימות שמאפיין ה-completed שלhn הוא false. את זה עושים בעזרת פונקציונליות נוספת ש-wixDataQuery מספק, ונקראת `wixDataQuery`.

wixDataQuery

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html>

wixDataQuery היא שאלתה המאפשרת לבצע בקשה למידע מטבלת המשימות. הבקשת יכולת להכיל שילוב של תנאים שיגדרו את התוצאות שלhn מוצפים.

יצירת שאלתה נעשית בשלושה שלבים:

1. אתחול השאלתה בהינתן שם הטבלה שמננה רוצים לקבל את המידע.
2. בניית פקודות שמגדירות את השאלתה.
3. הפעלה השאלתה.

1. אתחול השאלתה – `wixData.query`

<https://www.wix.com/velo/reference/wix-data.html#query>

את השאלתה מאתחלים בעזרת הפונקציה `wixData.query()`, שמקבלת ארגומנט אחד: שם הטבלה שאליה רוצים להויסיף את הרשומה החדשה. מתחילה לבנות את השאלתה בעזרת:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
}
```

2. בניית השאלה

לאחר מכן בונים את התנאים של השאלה באמצעות מגוון פונקציות. במקרה זה יש לקבל רק את המשימות שלא הושלמו. כלומר, כל המשימות שערך ה-`completed` שלහן הוא `false`.

- לכן, משתמשים בפונקציה `eq` (מהמילה שווה – `equal`), שמקבלת שני ארגומנטים:
- שם המאפיין שבאמצעותו רצים לסנן את התוצאות – במקרה זה `completed`.
 - ערך שרצים שייהי לתוצאות – במקרה זה `false`.

למידע נוסף על פונקציית `eq`:

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#eq>

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
}
```

* יש פונקציות רבות ומגוונות שמאפשרות בניית שאלתה ואפשר לקרוא עליהן בlienק:
<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html>

3. הפעלת השאלה

לביצוע השאלה אפשר להשתמש בשתי פונקציות אסינכרניות שונות:

- `find()` – החרזרת כל האיברים שעוניים על השאלה.

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#find> .2 – החרזרת מספר האיברים שעוניים על השאלה.

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#count>

מכיוון שציריך לדעת רק את מספר המשימות שלא הושלמו, נשתמש בפונקציה `count()`:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
    .count();
}
```

כעת, כששагנו מספר המשימות שלא הושלמו, כתובים אותן ברכיב הטקסט.
 מתחילהים ביצור הטקסט שרצים לראות:

```
let activeTaskText;
switch (activeTaskCount) {
  case 0 :
    activeTaskText = 'Completed all tasks';
    break;
  case 1 :
    activeTaskText = '1 item left';
    break;
  default:
    activeTaskText = `${activeTaskCount} items left`;
    break;
}
```

עכשיו, מציבים את הטקסט החדש בערך ה-`text` של רכיב הטקסט שה-ID שלו הוא `.activeTaskCount`.

זכרים איך עושים את זה?

```
$w('#activeTasksCount').text = activeTaskText;
```

למיידנו נוסף על משתנה ה-`text` השיב לרכיב הטקסט:

[https://www.wix.com/velo/reference/\\$w.Text.html#text](https://www.wix.com/velo/reference/$w.Text.html#text)

הינה כל הפונקציה:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
    .count();

  let activeTaskText;
  switch (activeTaskCount) {
    case 0 :
      activeTaskText = 'Completed all tasks';
      break;
    case 1 :
      activeTaskText = '1 item left';
      break;
    default:
      activeTaskText = `${activeTaskCount} items left`;
      break;
  }

  $w('#activeTasksCount').text = activeTaskText;
}
```

יצרתם פונקציה שמעדכנת את מספר השירותים שלא הושלמו, אבל אף אחד לא קורא לפונקציה. מתי בעצם צריכים לקרוא לפונקציה?

1. לאחר הוספה של משימה חדשה – בסוף פונקציית `addNewTask`

```
async function addNewTask() {
  const taskTitle = $w('#taskInput').value;

  if (taskTitle.length !== 0) {
    // task insert code

    await updateActiveTaskCount();
  }
}
```

2. לאחר שינוי סטטוס של משימה – בסוף ה-`event`

```
export async function completedCheckbox_change(event) {
  // completed update code

  await updateActiveTaskCount();
}
```

3. כאשר העמוד סיים להיטען בעזרת `$w.onReady`

\$w.onReady()

[https://www.wix.com/velo/reference/\\$w.html#onReady](https://www.wix.com/velo/reference/$w.html#onReady)

זהוי פונקציה שרצה כאשר כל הרכיבים של העמוד סיימו להיטען. בפונקציה זו אפשר לכתוב קוד שירוץ לפני שהמשתמש יתחל לבצע שינויים באפליקציה.

לכן, קוראים ל-`updateActiveTaskCount` בתוך הפונקציה `onReady` כך:

```
$w.onReady(function () {
  updateActiveTaskCount();
});
```

ובדרך כלל-`preview`.

- מודדים ששמות המשימות הראשונית נכונה
- מוסיפים משימה חדשה ומודדים שמספר המשימות שלא בוצעו עולга ב-1
- מסמנים משימה שלא בוצעה ומודדים שמספר המשימות שלא בוצעו יורד ב-1
- מודדים סימון למשימה שבוצעה ומודדים שמספר המשימות שלא בוצעו עולגה ב-1

עובד? שיגען! הלא!

סינון המשימות לפי סטטוס המשימה

עכשו נסנן את המשימות המוצגות לפי הסינון שהמשתמש יבחר בcptori הרדיו (radio group) שנמצאים מעל רשימת המשימות:



למידע נוסף על רכיב cptori הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html)

להלן התחנות הרצינית עבור כל אחד מהcptoriים:

All Tasks - כל המשימות יופיעו ללא סינון

Completed - רק המשימות שהושלמו (שהמשתנה completed שלהם שווה ל-true) יופיעו

Active - המשימות שעדיין לא הושלמו (שהמשתנה completed שלהם שווה ל-false) יופיעו

מכיוון שהסינון יבוצע לאחר שהמשתמש ילחץ על אחד הcptoriים, ה-event המתאים הוא radioButtonGroup.onChange – מօסיפים אותו כמו שלמדתם קודם:

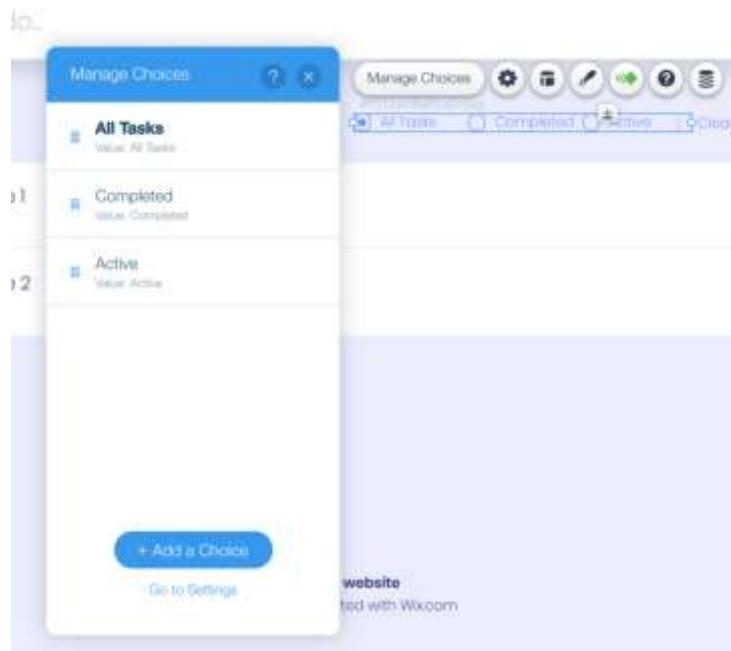
```
export function filterRadioGroup_change(event) {
  //Add your code for this event here:
}
```

למידע נוסף על onChange event של cptori הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html#onChange](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html#onChange)

בשלב הבא מבינים מהו סוג הסינון שבחר המשתמש וממשים אותו.

עבור כל אחת מהאפשרויות ב-radio group מזינים ערך זהה לטקסט שמייצג את הבחירה. אפשר לראות ולשנות את הערבים בלחיצה על **Manage Choices**.



על מנת לקבל את הערך שנבחר על ידי המשתמש, קוראים לערך ה-value שהוא חלק מכל כפתור רדיו, וכשמו כן הוא – הנתון (או הערך) שאותו אנו רוצים לקבל מהמשתמש.

```
export function filterRadioGroup_change(event) {
  const filterValue = $w('#filterRadioGroup').value;
}
```

למידע נוסף על משתנה ה-value של כפתורי הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html#value](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html#value)

עכשו מבצעים סינון של המידע שה-dataset מביא אל העמוד בהתאם לבחירה של המשתמש.

wixDataFilter

<https://www.wix.com/velo/reference/wix-data.WixDataFilter.html>

wixDataFilter מאפשר לנו לסנן את המידע שmagiu אליו מה-dataset לפי תנאים מוגדרים.

סינון מידע מה-dataset נעשה בשלושה שלבים:

1. אתחול המסנן `wixData.filter()`.
2. בניית הפקודות שגדירות את הסינון.
3. הפעלת הסינון על ידי העברת של אובייקט הסינון שיצרתם ל-dataset.

wixData.filter() .1

<https://www.wix.com/velo/reference/wix-data.html#filter>

מייצר אובייקט סינון, שmorpher ל-dataset, ובכך מאפשר לסנן את המידע שה-dataset מביא לעמוד:

```
let filter = wixData.filter();
```

2. בניית הסינון

לאחר מכן בונים את הסינון על ידי שימוש באותם התנאים שמצדירים שאלתיה. במקורה ה자가 צריכים ליצור שלושה סוגי שונים של סינון: כאשר המשתמש לוחץ על `Completed`, רוצים לסנן את כל המשימות שעורך ה-`completed` רק את המשימות שהושלמו. בעצם רוצים להשאיר את כל המשימות שעורך ה-`completed` שלhn שווה ל-`true`. לצורך זה משתמשים ב-`eq` (בדומה לשאלתיה שיצרתם קודם):

```
let filter = wixData.filter().eq('completed', true);
```

כאשר המשתמש לוחץ על `Active`, רוצים לסנן את המשימות שהושלמו ולהציג את המשימות שלא הושלמו. פה, רוצים להשאיר את כל המשימות שעורך ה-`completed` שלhn שווה ל-`false`. שוב משתמשים ב-`eq`:

```
let filter = wixData.filter().eq('completed', false);
```

ועבור All Tasks בונים אובייקט סינון ריק:

```
let filter = wixData.filter();
```

3. dataset.setFilter() - הפעלת הסינון על ה-`dataset`

<https://www.wix.com/velo/reference/wix-dataset.Dataset.html#setFilter>

(`dataset.setFilter`) היא פונקציה אסינכורונית שmaps על רכיב ה-`dataset`. היא מקבלת ארגומנט אחד – אובייקט הфиילטר שיצרתם עד עכשו.

לאחר שקוראים לפונקציה הזו, ה-`dataset` יבצע קריאה מחודשת למשימות מהטבלה, אבל ללא משימות שאין עונות על קритריון הסינון.

```
await $w('#tasksDataset').setFilter(filter);
```

עכשו מחברים הכל יחד ל-`event` של ה-`onchange` שיצרתם קודם:

```

export async function filterRadioGroup_change(event) {
  const filterValue = $w('#filterRadioGroup').value;

  let filter;
  switch (filterValue) {
    case 'Completed':
      filter = wixData.filter()
        .eq('completed', true);
      break;
    case 'Active':
      filter = wixData.filter()
        .eq('completed', false);
      break;
    case 'All Tasks':
      filter = wixData.filter();
      break;
  }

  await $w('#tasksDataset').setFilter(filter);
}

```

לא מספיק לקרוא לפונקציה הסינון רק כאשר המשתמש לוחץ על אחד מכפתורי הסינון. יש לסקن את המשימות בעוד כמה מצבים בקדוד. לכן נתחיל בהעברת קוד הסינון לפונקציה ייודית `filterTasks`:

```

async function filterTasks() {
  const filterValue = $w('#filterRadioGroup').value;

  // create filter code

  await $w('#tasksDataset').setFilter(filter);
}

```

עכשו, קוראים לפונקציה מכל המקומות השונים שדרושים אף הם סינון:

1. כאשר המשתמש לוחץ על אחד הכפתורים מה-buttons radio group (כמו שעשיתם):

```

export async function filterRadioGroup_change(event) {
  await filterTasks();
}

```

2. כאשר המשתמש משנה את סטטוס ה-`completed` של אחת המשימות:

```

export async function completedCheckbox_change(event) {
  // change completed status
  // update uncompleted counter
  await filterTasks();
}

```

3. כאשר המשתמש מוסיף משימה חדשה:

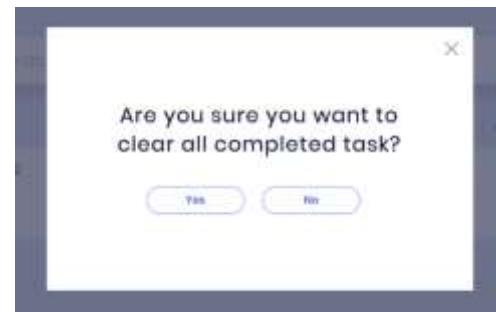
```
async function addNewTask() {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    // add new task
    await updateActiveTaskCount();
    await filterTasks();
  }
}
```

עבורים ל-view, "משחקים" עם הסינון שזה עתה נכתב, משנים סטטוס משימות כאשר נמצאים על סינון מסוים ומואדיים שהן נשארות או נעלמות ולבסוף, מוסיף משימה חדשה כאשר נמצאים במצב סינון של משימות שלא הושלמו ומואדיים שהמשימה לא מופיעה ברשימה.

שמחה וSSHON! יש סינון! 😊

ניקוי המשימות שהושלמו

עכשו ממשים את כפתור ה-clear, שבלחיצה יפתח את חלון האישור:



- אם המשתמש ילחץ Yes => סוגרים את חלון האישור ומוחקים את כל המשימות שהושלמו
- אם המשתמש ילחץ No => סוגרים את חלון האישור ולא עושים דבר

את חלון האישור כבר הכתינו בעבריכם וקרואתי לו Clear Confirmation. תמצאו אותו ב-Site Structure מתחת ל-lightboxes.

מוסיפים event של לחיצה על כפתור ה-clear פאנל ה-Properties:

```
export function clearCompletedButton_click(event) {
  // Add your code for this event here:
}
```

ועכשיו אלמד איך פותחים את חלון האישור.

wix-window

<https://www.wix.com/velo/reference/wix-window.html>

wix-window הוא מודול שמאפשר עבודה מול חלון הדף שעליו עובדים. על מנת להשתמש במודול, יש ליבא אותו מהאזור העליון של קובץ הקוד (בדומה ליבוא של :(wix-data

```
import wixWindow from 'wix-window';
```

wixWindow.openLightBox()

<https://www.wix.com/velo/reference/wix-window.html#openLightbox>

המודול `wixWindow` תומך בفتיחה של חלון חדש על ידי קריאה לפונקציה האסינכרונית `openLightBox`, שמקבלת שני ארגומנטים:

1. שם החלון שאוטו רוצים לפתח. במקרה זה – '`'Clear Confirmation'`
2. האובייקט שרוצים להעביר לחלון. במקרה זה אין צורך בכר.

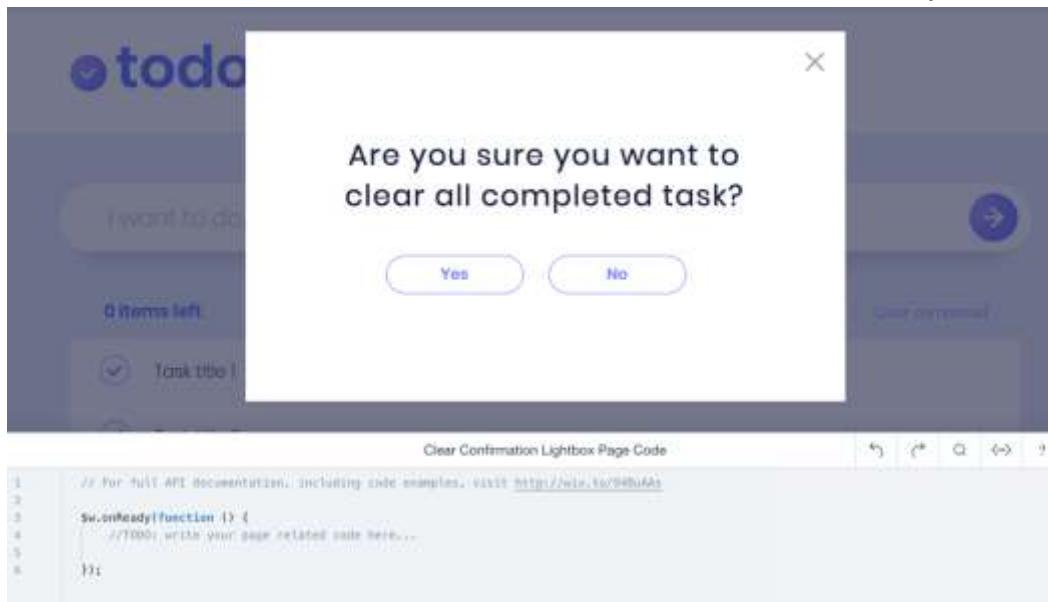
ערך ההחזרה של promise יהיה האובייקט שיוחזר על ידי החלון.
לכן, על מנת לפתח את החלון כתובים את הקוד:

```
export async function clearCompletedButton_click(event) {
  const shouldClearCompleted = await wixWindow.openLightbox('Clear Confirmation')
}
```

ועוברים ל-`preview` כדי לוודא שהחלון אכן נפתח.

כעת רוצים לדעת אם המשתמש לחץ על Yes או על No, כדי לדעת איך לפעול. כמובן, רוצים שההתשובה שתחזיר promise מה-`promise` כשהחלון יסגור תהיה ערך בוליאני, שאם הוא חיובי – "מחקנו המשימות שהושלמו ואם הוא שלילי – לא יקרה דבר.

עוברים לחלון ה-`onClear` כדי להחזיר תשובה ל-`event`.



גם פה, כמו בעמוד ה-`HOME` שלכם, אפשר לכתוב קוד בעורק הקוד שבתחתית המסך.
ועכשיו, מוסיפים `event` לחיצה על כפתור ה-`Yes`:

```
export function approveBtn_click(event) {
```

```
// Add your code for this event here:  
}
```

ומוסיפים event לחיצה על כפתור ה-**ס-ו**:

```
export function rejectBtn_click(event) {  
    // Add your code for this event here:  
}
```

כעת רוצים לסגור את החלון בלחיצה על כל אחד מהכפתורים. מתחילהם ביבוא המודול - **wixWindow** לחלק העליון של הקוד:

```
import wixWindow from 'wix-window';
```

wixWindow.lightbox.close()

<https://www.wix.com/velo/reference/wix-window/lightbox-obj/close>

(**wixWindow.lightbox.close()**) היא פונקציה שסגורת את החלון שבו נמצאים ומחזירה את המשתמש לעמוד שפתח את החלון. הפונקציה הזו מקבלת ארגומנט אחד: ערך שינון **wixWindow.openLightbox** promise שהזרם מ-**ה-א-ס-ו**.

לכן, בלחיצה על Yes מוחזרים true ובלחיצה על No מוחזרים false:

```
export function approveBtn_click(event) {  
    wixWindow.lightbox.close(true);  
}  
  
export function rejectBtn_click(event) {  
    wixWindow.lightbox.close(false);  
}
```

אש! חוזרים לעמוד ה-**HOME** לפונקציית **onClick** שהתחלנו לכתוב קודם, כדי לבצע את המ剔ה במידה שהמשתמש ילחץ על Yes. במידה שהמשתמש בחר למחוק את המשיקה בשני שלבים:

1. שאלתה שתחזיר את כל המשימות שערך ה-**completed** שלן הוא true.
2. מחיקה של כל המשימות שהתקבלו מהשאלתה.

למגדתם מוקדם יותר איך לבצע שאלתה דומה, אבל בשאלתה הקודמת רציתם לקבל את מספר המשימות ولكن השתמשתם ב-**count()**. הפעם תשימושו ב-**find()**:

```
const queryResult = await wixData.query('Tasks')  
    .eq('completed', true)  
    .find();  
  
const completedTasks = queryResult.items;
```

למידע נוסף על פונקציית **find**:

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#find>

כשקוראים ל-`find`, חוזר אובייקט שמכיל כל מיני דברים. אחד מהם הוא ה-`items` שמחזיק מערך של כל הערכים שחזרו – במקרה זה, מערך של כל המשימות שחזרו מהשאילתה.

כעת עוברים על כל אחת מהמשימות הללו ומחקם אותן באמצעות:

מחיקת רשומות מהטבלה – `wixData.bulkRemove()`

<https://www.wix.com/velo/reference/wix-data.html#bulkRemove>

מחיקת רשומות מהטבלה נעשית על ידי שימוש בפונקציה האсинכרונית `bulkRemove()` שמקבלת שני ארגומנטים:

1. שם הטבלה שמננה רוצים למחוק את הרשומות – במקרה זה `Tasks`.
2. מערך המכיל את ה-`ids` של הרשומות שאוותם רוצים למחוק.

זכרים את מזהה ה-`_id` שנכנס באופן אוטומטי לכל שימוש בטבלה? מעולה!
از עכשו תשתמשו בו על מנת ליצור מערך של ה-`ids` שאנו רוצים למחוק.

```
const queryResult = await wixData.query('Tasks')
  .eq('completed', true)
  .find();
const completedTasks = queryResult.items;
const completedTasksIds = completedTasks.map(task => task._id);
```

את המערך שיצרנו נעביר לפונקציה האсинכרונית `bulkRemove`.

```
await wixData.bulkRemove('Tasks', completedTasksIds);
```

ולקינוח, קוראים שוב לפונקציה `filterTasks`, כדי להסיר מהציג את המשימות שנמחקו.

ולקood המלא:

```
export async function clearCompletedButton_click(event) {
  const shouldClearCompleted = await wixWindow.openLightbox('Clear Confirmation');

  if (shouldClearCompleted) {
    const queryResult = await wixData.query('Tasks')
      .eq('completed', true)
      .find();
    const completedTasks = queryResult.items;
    const completedTasksIds = completedTasks.map(task => task._id);

    await wixData.bulkRemove('Tasks', completedTasksIds);
    await filterTasks();
  }
}
```

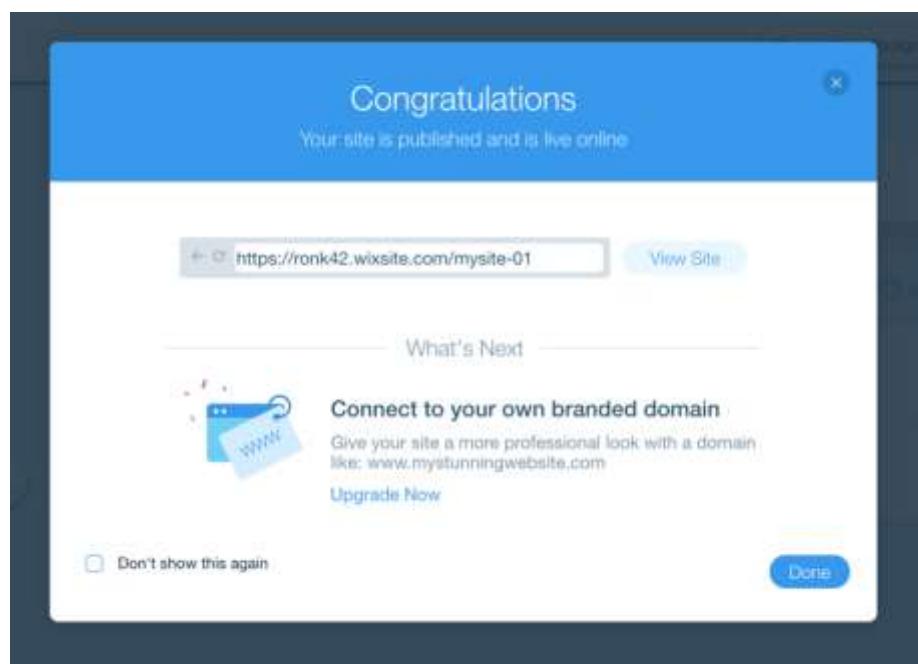
ועכשו ל-preview. צריך לוודא שיש כמה MERCHANTABILITY שהושלמו וכמה MERCHANTABILITY שלא הושלמו
ולבצע שתי בדיקות שונות:

1. לחיצה על Clear completed ולאחר מכן לחיצה על No => יש לצפות לך ששם דבר לא יקרה.
2. לחיצה על Clear completed ולאחר מכן לחיצה על Yes => יש לצפות שכל MERCHANTABILITY שהושלמו ימחקו.

ממש כמעט ס"מתמן!
 זוכרים את כפתור ה-publish מתחילת הפרק?
 עכשו הזמן ללחוץ עליו!

אחרי לחיצה על כפתור ה-publish תהיה האפליקציה שלכם זמינה באתר לכל דבר וכל העולם!

הכתובת של האפליקציה שלכם תוצג בחלון שייפתח לאחר שתלחצו על publish:



* אחרי שפרסמתם את האפליקציה, MERCHANTABILITY שהוסיףتم בזמן שעבדתם על האפליקציה לא יופיעו ב-preview, מכיוון שאתם עובדים על מסדי נתונים שונים עבור פיתוח ועבור האפליקציה האמיתית.

תוספת — ייצור מסד נתונים

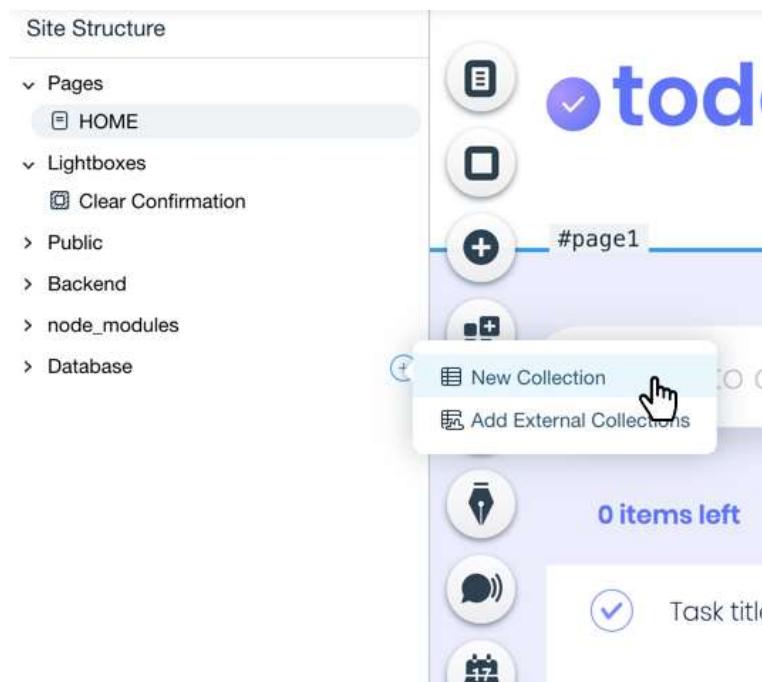
מסד נתונים הוא אמצעי המשמש לאחסון מסודר של נתונים. במקורה של Velo, הנתונים הללו מאוחסנים במודול של טבלאות (collections) ובכל טבלה העמודות מייצגות שדות (fields) והשורות מייצגות רשומות שונות.

נוהג לשיר כל טבלה לישות מסוימת באפליקציה, אז כל שורה מייצגת מקרה ספציפי של הישות.

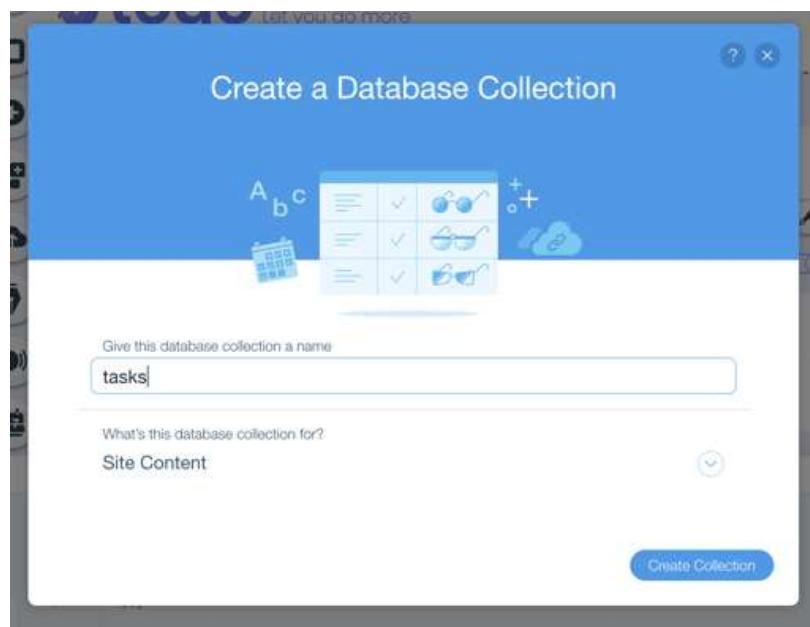
מוסך על כך, כל עמודה בטבלה היא בעצם שדה שמייצג מאפיין מסוים של הישות. לכל שדה יש סוג כגון טקסט, מספר, בוליאי, תמונה וכדומה. הבה נתכנן את מסד הנתונים של האפליקציה.

באפליקציה שלכם יש ישות שנקראת משימה (task), ולכן יש ליצור טבלה שנקראת Tasks. כל שורה בטבלה תציג משימה בודדת. על מנת ליצור טבלה חדשה עוברים עם העבר על כפתור +, שיופיע בעבר על צידו הימני של ה-site structure שנמצא בפאנל database.

לאחר מכן לוחצים על ה-`New Collection`:



לחיצה תקפיים מסר שבו נתונים לטבלה את השם tasks ולאחר לחיצה על תיווצר טבלה חדשה.

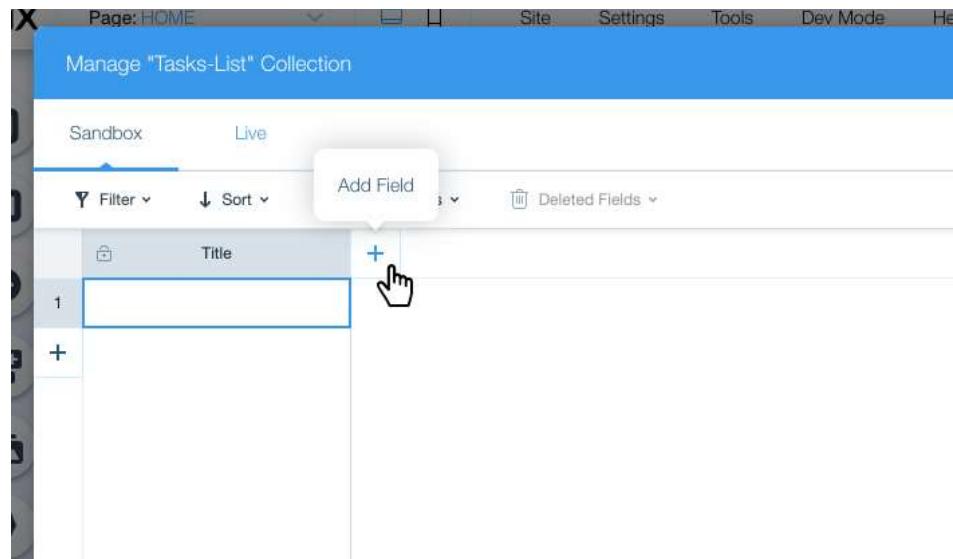


יצירת הטבלה הוסיפה טבלה חדשה מתחת ל-site structure, database – מתחת site structure, ובנוסף העבירה אתכם ישר לאזורי שנקרא "עורך התוכן" – content manager, שבו אפשר להוסיף שדות ולערוך את התוכן של הטבלה.

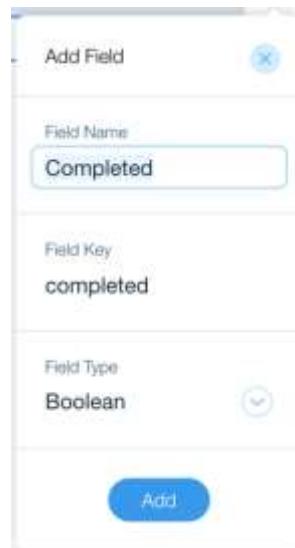
הגיע הזמן לייצר את השדות. כל משימה מכילה את תיאור המשימה בטקסט וערך בוליאני שmagdir את סטטוס המשימה – אם היא הושלמה או לא. לכן, יש צורך בשני שדות:

- שדה **Title** מסוג טקסט (text) – שמיוצר באופן אוטומטי ברגע שמייצרים טבלה חדשה
ולכן לא צריכים להויף אותו ידנית.
- שדה **Completed** מסוג בוליאן (boolean).

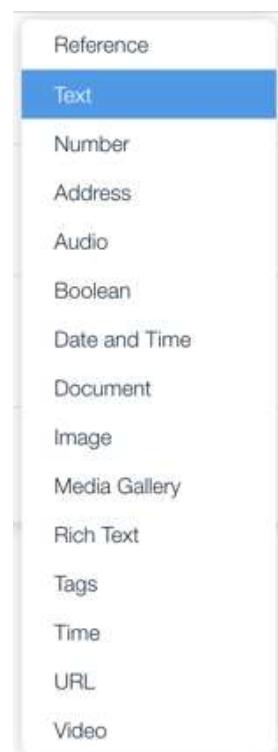
על מנת להוסיף שדה יש ללחוץ על כפתור הפלוס ואז יפתח פאנל ההגדרות של השדה.



בפאנל ההגדרות יש שלושה קלטים שצורך למלא כאשר מייצרים שדה חדש:



- **Field Name** – שם השדה שורצים. במקרה זהה – `Completed`.
- **Field Key** – זהה מפתח השדה שנוצר באופן אוטומטי על ידי השם שניתנו לשדה.
- **הערך** של השדה ישמש אתכם כאשר תרצו לפנות לשדה זהה דרך קוד. לרוב אין צורך לגעת בערך שהמערכת נותנת באופן אוטומטי.
- **סוג** השדה שורצים ליציר. אפשר לבחור ממבחן גדול של סוגים.



כעת קוראים לשדה הנוסף `Completed` ומגדירים אותו מוגן מסוג `Boolean`.

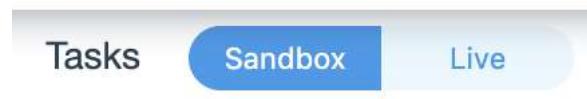
לבסוף לוחצים על **Add** והשדה יתווסף.

כעת יש טבלה בעלת שתי עמודות. אפשר להוסיף רשומות באופן ידני, לשנות אותן ואףלו למחוק אותן מתוך עורך התוכן.

למדתם את רוב המידע שהייתה רלוונטי לכם עבור מסדי נתונים ב-*SQL*. עכשיו תלמדו על עבודה עם מסדי הנתונים לאחר שמספרם את האתר.

Sandbox | Live

אם תשים לב, בחלק העליון של עורך התוכן יש שני כפתורים:



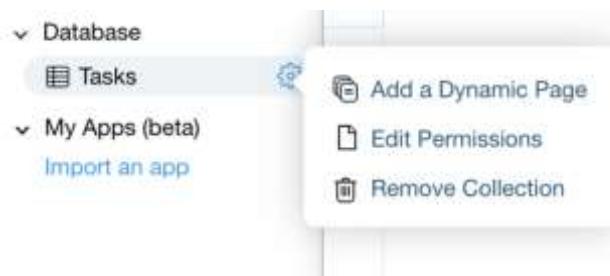
- בלחיצה על כפתור *Sandbox* יוצגו הרשומות שיפויו בזמן עריכה האפליקציה. כמובן, הרשומות שוראים כאשר לוחצים על *view*.
- בלחיצה על כפתור *Live* יוצגו הרשומות שוראים באפליקציה האמיתית אחרי שלוחצים על *publish*.

Permissions

כל טבלה יש הרשאות שמאגדירות מי יכול לראות את הנתונים שבטבלה, מי יכול ליצור אותם, לעורך אותם ולמחוק אותם. הרשות הראשונית שמקבלים כשיצרים טבלה חדשה הן מאוד שמרניות. כמובן, הן לא ייתנו יותר מדי יכולות למשתמשים חיצוניים. לכן, הפעולות שמבצעים על הטבלה באפליקציה האמיתית (לא ב-*preview*) יחוירו שגיאה:

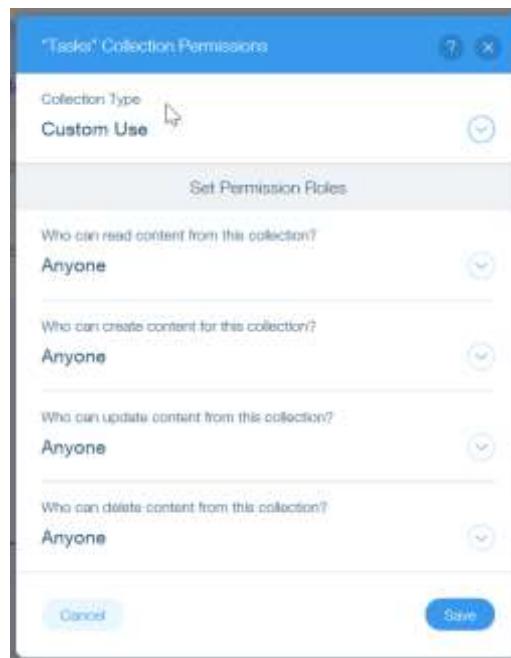
<i>wixData.get</i>	-
<i>wixData.insert</i>	-
<i>wixData.update</i>	-
<i>wixData.delete</i>	-

כדי למנוע את זה, מוטב לשנות את הרשות. עושים זאת באמצעות לחיצה על *Edit* *Permissions*:



ואז, יופיע חלון שבו תוכלן לעורוך את ההגדירות של הרשות. עבור אפליקציית המשימות שלכם, תרצו שההרשאות יראו כך:

* בתבנית שיצרתי עבורכם כבר הגדרתי את הרשות של הטבלה בצורה הזו.



סיכום

סימתמו לכתוב את האפליקציה שלכם. מה למדתם?

- חיבור מידע מטבלאות הנתונים לאפליקציה
- יצירה אינטראקטיבית עם רכיבים של העורך הויזואלי דרך הקוד
- צפיה, הוספה, עריכה ומחיקה של רשומות בטבלאות בעזרת `wix-data`
- יצירה של אילות וסינונים על הרשומות מהתבלה
- פיתוח חלונית וסגירתה באמצעות קוד
- מי שקרה את הנספה – איך ליצור טבלאות חדשות

עכשו הגיע הזמן שלכם ליצור אפליקציות חדשות, ללימוד על עוד יכולות ש-`Velox` מספקת לכם ולהתנסות בהן.

נספח: ג'אווה סקריפט מונחה עצמים

מחברים: יהודית גלעד, חברת Chegg, רן בר-זיק

מה זה תכונות מונחה עצמים?

תכונות מונחה עצמים הוא שיטה לכתוב קוד מורכב. עד כה בספר התנסיתם בעיקר בלמידה השפה וכתיבת קוד פשוט יחסית. פונקציה אחת, פקודה אחת או שתיים ומערכות מידע לא מורכבים. אבל בעולם האמיתי אנחנו חווים חלק את הקוד לחלקים או ל"חטיות" בדיקן כמו פונקציות או אובייקטים שלמדנו כאן. חלוקת הקוד היא הכרחית כאשר אנו מפתחים מערכות גדולות כיוון שאחרת התוכנה תהיה מאוד מסורבלת. הרבה יותר קל למצוא את הדרך בכמה קבצים, שכל אחד מהם אחראי לחלק אחד בתוכנה מאשר בקובץ אחד ענק. החלוקת לכמה קבצים או כמו "יחידות" היא חיונית.

ישן כמו וכמה שיטות לחלק קוד לחלקים שניתן לנו להנשל אותם ביחד. אחת השיטות הבסיסיות ביותר היא תכונות מונחה עצמים המאפשרת למתכנתים לחלק את הפונקציונליות של התוכנה שלהם לחלקים שונים. כך חלקים שונים מתחווים עם אפיון פונקציונלי ייחודי להם – אפשר אף לומר אופי. בתכונות מונחה עצמים יכול עצם ניתן לנכס שני סוגים תכונות: מאפיין או מתודה. כך שבתום הגדרת העצם יש לנו ייחידה לוגית בעלת מאפיינים ויכולות פונקציונליות הקשורות אליה באופן בלעדי. לדוגמה תהשבו על כלב – לכלב יש מאפיינים ייחודיים ויכולות פונקציונליות, נבייה למשל. במסגרת השיטה זו אנו משתמשים על המרכיבת מכלול. אם אפשר לחלק את המכלול לעצמים בעלי מכנה משותף נוכל ליצור "תבניות" יותר מופשטות שמקילות כמה סוגים עצמים. למשל יונקים, או בעלי חיים בדוגמה הכלב.

הבה ונdagים בדוגמה מהחיים. בפרק על AJAX הראינו איך יוצרים קשר עם API של בדיחות צ'אק נוריס. הבה ושוב נניח שיש שני סוגים קריאות. הראשונה לבדיחות צ'אק נוריס על אוכל:

<https://api.chucknorris.io/jokes/random?category=food>

והשנייה לבדיחות צ'אק נוריס על כסף:

<https://api.chucknorris.io/jokes/random?category=money>

נניח ומבקשים ממוני להציג את שתי הקריאות לקומפוננטה. אם אין לנו ארכיטקטורה מסודרת – המכוננת אותה לסדר לוגי של הקוד שלו, אני פשוט אוצר שתי פונקציות עם קריאות fetch. אם יבקשו ממוני קריאה שלישית אני אוצר קריאה נוספת וכך הלאה. אבל זה לא תכון נבון של המרכיבת, כי אם יהיה עוד ועוד קריאות כאלה, אני אלץ ליצור עוד ועוד פונקציות ואז יהיה קשה לתחזק אותן. (ותחזקה היא 90% ממחזור חיים של כל תוכנה). יתרה מכך, ברור לנו שיש כאן מכנה משותף לכל קריאות ה-API שלנו, ואם הכתובות של ה-API תשתנה, למשל, אצטרך לשנות את הכתובות בכמה פונקציות. זהה מתכוון לטעויות אנוש. במיללים אחרות באגים, ההופכים את התוכנה ללא יציבה.

מה הדרך? הדרך היא לדמיין עצם בסיסי בעל המאפיינים והפונקציונליות הכלילית שאנו מחפשים, ולהגדיר את התבנית שלו בצורה שתאפשר למחזר את התבנית לצרכים השונים

שללה. לתבניות אלו קוראים בתכנות מונחה עצמים "קלאסים" עליהם למדנו. השם הרשמי שלהם במדעי המחשב נקרא "מחלקה", אך לא תמצאו מונחת אחד בחיים האמיתיים שיקרא לקלאס מחלקה. בפרק זהה אנו משתמשים במונחים המקובלים בתעשייה ולפיכך המחלקה תיקרא בפרק קלאס.

העצמים שנוצרים לפי המחלקות השונות הם למעשה מימוש של תבנית מוגדרת מראש. קוראים להם גם אובייקטים – עצמים.

לפנינו נראה קוד חשוב לצין עוד שני דברים. אפשרות שנותן לנו תכנות מונחה עצמים היא להגדיר תכניות מוכנות מראש, כך שנדע את היקף הפונקציונליות של עצמים שאנו עובדים איתם אף אם לא הכיר איך זה נעשה. וגם כדי שנוכל להרחיב את הפונקציונליות שלהם לבחירתנו. היכולת להרחיב פונקציונליות בתכנות מונחה עצמים נקראת ירושה. הכלב "ירוש" את תכונות ההולדה ואת מאפייני המין השונים של כל היונקים. כאשר יונקים היא מחלקה שמרחבת למקרים פרטיים של כלבים. כי כלבים יכולים גם לחשוף בזנב. ירושה מאפשר להרחיב את הגדרת העצם בהתאם על תכניות עצמים אחרות. لكن בג'אווהסקריפט תפגש את המילה `extends` – מרחיב.

עוד דבר חשוב שנצין כאן – הוא שישנו עוד דרכים בג'אווהסקריפט לקטול פונקציונליות, "ירשת" אותה. מסיבות היסטוריות תכנות מונחה עצמים קלאסי אינן הצד החזק של השפה. לדברי המיסדים של ג'אווהסקריפט נעשו טיעות שקשה לתunken. (Javascript: The Good Parts by Douglas Crockford). ג'אווהסקריפט מצטיינת בגישת התכונות הפונקציונלי שללה, עליה יש פרק אחר בספר. כאן נראה את גישת תכנות מונחה העצמים הקלאסים.

נזכיר אובייקט או עצם של חיבור שהוא ייה אב הטיפוס, ואז נירש ממנו את תכונותיו שיאפשרו לנו להתחבר ל-`API`. ככלומר אב הטיפוס תהיה הפונקציונליות המשותפת, האובייקטים היורשים ירחיבו רק את הדברים שהן צריכים.

כך למשל באובייקט האב יהיה את החיבור עצמו ל-`API`, העיבוד של `JSON` והחזרה של הבדיקה. כאשר מי שיורש מאובייקט האב יקבל את כל זה מובנה ויצטרך רק להגדיר על כתובות ה-`API` שמשמעותו.

```

class APIConnector {
  constructor(category) {
    this.category = category;
    this.url = `https://api.chucknorris.io/jokes/random?category=${this.category}`;
  }

  getJoke() {
    return fetch(this.url, {})
      .then((response) => {
        return response.json();
      })
      .then((jsonObject) => {
        return jsonObject.value;
      });
  }

  getCategory() {
    return this.category;
  }
}

class MoneyJoke extends APIConnector {
  constructor() {
    super('money');
  }

  calculateMoney() {
    console.log('Money not a problem when U R Chuck');
  }
}

let moneyJoke = new MoneyJoke();

moneyJoke.getJoke().then((joke) => {console.log(joke)})

```

בדוגמה זו יש לי **קלאס** – או תבנית עצם שנקרא **קלאס הבסיס** שלו. ממנה כל הקלאסים יכולים לרשף. כך למשל **קלאס MoneyJoke** ירש ממנה וכאשר נוצר עצם מסוג **ה-Object MoneyJoke** יעמכו לרשותו כל המאפיינים ומתקודמות שהוגדרו ב**קלאס** של **MoneyJoke** והורחבו על ידי **קלאס MoneyJoke**. בדוגמה זו, **קלאס MoneyJoke** גם מפעיל **APIConnector**

את הקונסטרוקטור של הקלאס המקורי עם התוכנה שהוא רוצה. אם יש לקלאס היורש מתודות נוספות, הן מתווספות כרגע. למעשה זו תבנית מובנית בתוכנות מונחה עצמים שברגע שעצם נוצר בדומה לקלאס מסוים מוצצת מתודת `constructor` בראשונה כדי לאותל את היישות החדשה שנוצרה. האובייקט שלנו. קונסטרוקטור גם עובר בירושה וכל הקונסטרוקטורים של הקלאסים האבות יופעלו לפי הסדר, אלא אם איזושהי קלאס בדרך תמנע את זה על ידי קונסטרוקטור משלו.

ככה אני יכול ליצור כמה קלאסים שאין רוצה שיורשים מה-`APIConnector`. מה היתרון? אם כתובות ה-`API` משתנה, או הדרך שבה עובד החיבור ל-`API` משתנה, אני צריך לשנות את זה בקלאס אחד.

באו נחקרו עוד היבט של מערכת היחסים בין מחלוקת שירשת תוכנות מקלאס אב אחר. בדוגמה הבאה יש לנו שני סוגי משתמשים במערכת: משתמש פשוט ומשתמש מנהל. לכל אחד מרהסוגים הללו יש תוכנות ומתודות מיוחדות לסוג המשתמש שלו. בדוגמה שלנו נגדיר קלאס `Shiafin` משתמש כלשהו – קלאס שיתאים לנו לכל מיני סוגים של משתמשים במערכת – גם למנהלים, גם למנהלים פשוטים ואולי עוד סוגים משתמשים שנרצה להויסף בעתיד. כי, זוכרים? 90% ממחזורי ח'י התכונה הוא במצב התחזקה, لكن סביר מאוד להניח שנרצה להויסף משתמשים מסוגים שונים בעתיד.

```
class BaseUser {  
    constructor(name) {  
        this.name = name;  
    }  
  
    setName(name) {  
        this.name = name;  
    }  
  
    getName() {  
        return this.name;  
    }  
  
    setAddress(address) {  
        this.address = address;  
    }  
  
    getAddress() {  
        return this.address;  
    }  
}  
  
class AdminUser extends BaseUser {  
    constructor(name) {  
        super(name);  
        this.admin = true;  
    }  
  
    performAdminAction() {  
        // performs action allowed only to admins  
    }  
}  
  
class StandardUser extends BaseUser {  
    contactAdmin() {  
        // Contact the user admin  
    }  
}
```


Prototype based

כבר הזכרנו שג'אווהסקריפט לא תוכנה לתוכנות מונחה עצמים קלאסי. היא פשוט אחרת. מחלקות מופשטות ומימושם הוא אשליה — סוכר סינטטי — נועד בזמןנו להפוך את השפה לאטרקטיבית בעיני מפתח ג'אווה ועוד כל מיסי סיבות פוליטיות. למעשה ג'אווהסקריפט חזקה יותר ממה שהיא נראה, ומאפשרת עוד לפחות שני סוגי של הרחבות פונקציונליות או כמו "ירשה" — מילה השאלה ממילון תכנות מונחה עצמים קלאסי.

במוקם לנחל את תכניות העצים במחלקות מופשטות, בג'אווהסקריפט "הגנו" של כל אובייקט נשמר אף הוא אובייקט ובכך פותח אפשרויות נוספות כדי למדוד לעומק. כאן נביא רק שתי יתרונות חשובים של עבודה עם ה-prototype — הוא שם האובייקט נשא ה-DNA של כל אובייקט בג'אווהסקריפט:

1. עצם יכול לרשף פונקציונליות ישירות עצמו אחר, ולא מקלט מופשט של אותו עצם. כך אם אובייקט `a` מרחיב את פונקציונליות של אובייקט `b`, ומוסיף לו מתודה בשם: `(what)_say` למשל. עתה אובייקט חדש `c` יכול לרשף מאובייקט `b` וישר לקבל את כל תכונותיו, אף לשנות להרחב אתם בעצמו. אין צורך בהגדרת מחלקות המאפיינות טווחים אלה, אלה ישר להרחב אובייקטים.
2. דבר נוסף — סוג של קסם בג'אווהסקריפט. אם ננסה את ה-prototype של אובייקט כל האובייקטים היורשים ממנו (בעל' אותו ה-prototype) ישתנו מיד! לדוגמה אפשר להרחב את הפונקציונליות של אובייקט `String` המבנה של השפה שתכילה מתודה `... fire` שתתבצע את המחרוזת עם אמוג'י של אש. מיד כל העצים מסוג `String` יוכל להשתמש בה.

זה נשמע אבסטרקט, אך בואו ונדגים. הנה קלאס של משתמש.

```
class User {
  constructor() {
    this.type = 'user';
    this.theme = 'light';
  }

  setName(name) {
    this.name = name;
  }

  setAddress(address) {
    this.address = address;
  }
}
```

מה קורה מאחרי הקליים כשהאני מבצע את הפקודה זו?

```
let newUser = new User();
```

מה שקורה הוא שנוצר אובייקט ריק בשם newUser עם פרנס – כלומר מצביע אל האובייקט המקורי User. בנויגוד למה שהיינו מצפים בשפה מונחית עצמים רגילה – שבה נוצר אובייקט שלם עם כל המתודות של הקלאס User. בג'אווהסקריפט אין הבדל בין קלאס לאובייקט.

כשאני קורא ל:

```
newUser.setName('Moshe');
```

ג'אווהסקריפט ראיית ניגשת לאובייקט newUser ובודקת אם יש לו את המתודה setName. אם אין לה את המתודה זו, היא עוברת לאבא – האובייקט (או הקלאס) User, אם היא מוצאת אותו שם.מצוין, היא תפעיל אותו. במידה ולא, היא תמשיך לאבא של User. מי האבא? האובייקט Object שנמצא בסיס ג'אווהסקריפט.

אתם מוזמנים להריץ את הקוד של הקלאס והאובייקט שנוצר מהקלאס:

```
let newUser = new User();
console.log(newUser);
```

תוכלו לראות בכל הפתחים שתחת prototype יש לאובייקט newUser את התכונות שנוצרו בפונקציה הבנאית אבל המתודות עצמן מקורן באובייקט האב – ה"קלאוס" User. וגם לו יש מתודות אחרות שmagiuot מהאובייקט של ג'אווהסקריפט שכל האובייקטים יורשים ממנו.



למה חשוב לדעת את זה? כי מה לפי דעתכם יקרה אם נכתוב את הקוד הזה?

```
let newUser = new User();
delete User.prototype.setName;
newUser.setName('moshe'); // TypeError: newUser.setName is not a function
```

נקבל הודעה שגיאה! מדוע? כי שוב, העובדה שאנחנו יוצרים newUser מקלט User לא אומרת שמעכשו הקשר בין השניים ניתק. להיפך, newUser הוא אובייקט ריק שמכיל רק מה שקבענו לו בקונסטרקטור והמתודות שלו בעצם מגיעות מקלט האב User. שינו את User שינו את כל מי שיורש ממנו. כי אין כאן בעצם ירושה, יש כאן הפניה דרך הפירוטוטיפ. שינו את הפירוטוטיפ? שינו את כל מי שיורש ממנו.

זה יכול להיות שימושי כי אנו צריכים לזכור שם האובייקט הבסיסי בג'אווהסקריפט, ה-Object. שוכם יורשים ממנו גם ניתן לשינוי. כר למשל:

```

class User {

  setName(name) {
    this.name = name;
  }

  setAddress(address) {
    this.address = address;
  }

}

class DataObject {

  setName(name) {
    this.name = name;
  }

  setProp(key, value) {
    this[key] = value;
  }

}

let newUser = new User();
let dataObject = new DataObject();

Object.prototype.isValid = true;

console.log(newUser.isValid); // true
console.log(dataObject.isValid); // true

```

מה קורה כאן מאחורי הקלעים?

צרתי שני קלאסים וממהם יוצרתי אובייקטים עם פונקציה בנאית. אז הניסתי לפרקוטווטיפ של Object, האובייקט שהוא חלק מהשפה של ג'אווהסקריפט וכל האובייקטים יורשים ממנו את התכונה `isValid`.

מאחורי הקלעים נוצר בעצם עצם עץ היררכיה זהה:

Object (object) -> User (object) -> newUser (object)
Object (object) -> DataObject (object) -> dataObject (object)

כשאני מבקש `isValid` מ `newUser`, מנوع הג'אווהסקריפט בודק ראשית אם זה קיים באובייקט `newUser`. זה לא? נמשיך לפרטוטיפ של האבא, `User`. זה לא? נמשיך לפרטוטיפ של הסבא, במקרה זהה `Object`, האובייקט האב של כל האובייקטים של ג'אווהסקריפט. האם הוא נמצא שם? כן! כי אנחנו הכננו `isValid` לפרטוטיפ!

זה כבר מתחילה להיות מעניין, חישבו למשל שני רוצח פונקציה שתדוע אם לאובייקט שלי יש `name`. במידה וכן, הפונקציה תחזיר `true`. במידה ולא, הפונקציה תחזיר `false`. איך אפשר לעשות זהה דבר? בקלהות! על ידי תוספת לפרטוטיפ של `Object`.

```

class User {

  setName(name) {
    this.name = name;
  }

  setAddress(address) {
    this.address = address;
  }

}

function isValid() {
  if(this.name && this.name !== "") {
    return true;
  } else {
    return false;
  }
}

Object.prototype.isValid = isValid;

let newUser = new User();
let another(newUser = new User();

newUser.setName('Moshe');

console.log(newUser.isValid()); // true
console.log(another(newUser.isValid()); // false

```

זה לא דבר טריוויאלי להבנה, אבל ברגע שambilים אותו ומבינים את ה-prototype, אפשר באמצעות לנצל את הכוח והגמישות של ג'אווהסקריפט במקום להתייחס אליה כעוד שפה מונחית עצמים. הגמישות הזאת מאפשרת לג'אווהסקריפט להפתח ולהיות נפוצה ככל כך הרבה פלטפורמות.

ישנו עוד סוג של "ירשה" בג'אווהסקריפט — ירשה פונקציונאלית. זאת צורת הרחבות פונקציונליות הטבעית, החזקה והגמישה ביותר ביג'אווהסקריפט. היא מאפשרת חשיבה אחרת ו פשוטה יותר על היררכיה פונקציונלית של הקוד, ואף מאפשרת ליצור תכונות ומתקנות מוסתרות בرمמות שונות של ירשה. השימוש בפונקציות — שהם בעצם גם אובייקטים מן המניין

בג'אווהסקריפט – חופש קלפינו את כוח `the closures` העוצמתי של השפה, המאפשר ללבוד מצבים נתוניים, וליצור קוד מושתת מהיורשים ועוד. כת עולם פיתוח `frontend` יושמי פונקציוני בשתי ידיים. אנו רואים זאת בכיוון של ריאקט למשל, והדברים הופכים לפשטיטים מאוד, כאשר ג'אווהסקריפט לא מנסה לחקות שפות אחרות, אלא משתמש בכשרונות הניתנים לה מילידה.

נספח: שינויים מהמהדורה הקודמת (מהדורה 2.0.3)

נוספו הרחבות על פקודות `Promise.any` ו-`Promise.allSettled`, `Promise.race` ו-`Promise.raceWith`. בהתאם לתקן ES2021.

הווספה תת הפרק קינון ובדיקה מפתח בפרק על אובייקטים ותרגילים נוספים שבו מוסבר עלOptional Chaining. בהתאם לתקן ES2020.

עדכון הנוסף על `x` ו-`W` בקורס החדש של `Velo`. העדכון נעשה על ידי מור גלעד.

הווספה נוספת לקובד בפרק על פונקציה כתוכנות אובייקט.

תיקונה הפקודה `open` לפתיחת כל מפתחים במקום אחד.

תודה רבה על הרכישה של הספר!

עבדתי מאד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוכרי העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואייפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהkindle, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתווך תקווה שהרוכש והותםך לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

beninson@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעה אצבע דיגיטלית - כלומר בתוך דפי הספר נჩביםaim פרטי הרוכש באופן שקויף למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העברו לו את הפרטיהם שלכם באתר ומיהקנו את העתק שנמצא ברשותכם.

תודה וקריאה נעימה!