

תודה רבה על הרכישה של הספר!

עבדתי מאד קשה על הספר זהה: שעותת רבota של כתיבה, הגהה, תיקונים ומעבר על תוצריו העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

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

העותק הזה נמכר ל:

danielast4@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - ככלומר בתוך דפי הספר נჩאים פרטי הרוכש באופן שקוף למשתמש. כדי מאד למנוע מהעתיקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעיבר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!

לلمוד תכבות ג'אווהסקרייפט

בקלה

רון בר-זיק

מהדורה: 1.0.1



כל הזכויות שמורות (c) רן בר-זיק, 2019.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלודי, לא-יהודי, אישי, בלתי ניתן להעברה (למעט על פי דין), ובבלתי 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, ככלומר פסקה או שתים, 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 בתוך תוכנות שתפתח. אם אתה רוצה להכנסי אותן לפרויקט שלך, שלח מייל ונדבר על זה .

עריכה לשונית: יעל ניר

הגהה: חנן קפלן

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 ביגלאיין	15
צחי נמci	15
על ג'אווהסקרייפט	16
אין לומדים	18
ארגנו לעצמכם סביבת עבודה מסודרת	18
קראו את הפרקים לפי הסדר	18
קראו כל פרק פעמיים וכותבו לעצמכם את כל הדוגמאות	18
פתרו את התרגילים לדוגמה בסוף כל פרק	19
התיעצו עם אחרים	19
הגיעו לسدנאות ולמייטאפים	19
תרגלו, תרגלו, תרגלו	19
התקנת סביבת העבודה ודרך הלימוד	20
משתנים	26
טקסט	29
מספרים	34
מציאת השארית	36

37	אופרטורים מקוצרים
41	סוגי מידע פרימיטיביים נוספים
41	בוליוני
41	משתנה לא מוגדר
42	ריך
42	Symbol
43	מציאת הסוג של המשתנה
46	הערות
48	בקורת זרימה – משפטים תנאי
50	אופרטורים השוואתיים נוספים
53	אופרטורים לוגיים
55	אופרטור שלילי
56	אופרטור תנאי
57	אופרטורים המשווים ערכים
66	switch case
73	קבועים
75	בקורת זרימה – פונקציות
79	פונקציה עם ארגומנטים
81	ארוגמנטים עם ערכים דיפולטיביים
82	הפונקציה באובייקט
83	Hoisting
83	closure
85	פונקציה אונוכית ופונקציית חץ
86	פונקציה אונוכית במשתנה
88	פונקציה אונוכית שمبזדת מהסקופ הגלובלי
94	אובייקטים

98.....	מחיקת מפתח
98.....	הכנסת פונקציה בערך.....
99.....	אובייקט בקבוע.....
104.....	מערכות
107.....	מערכות ומחרוזות טקסט
110.....	<i>this-i new</i>
116.....	מבנה טקסט
119.....	לולאות.....
119.....	לולאת for
128.....	לולאה אינסופית
128.....	לולאת while
129.....	לולאת while do
130.....	לולאת forEach
133.....	לולאת for of
136.....	לולאת map
139.....	לולאת filter
142.....	לולאת sort
142.....	לולאות על אובייקטים
142.....	לולאות in
145.....	Object.keys
152.....	ג'אויסקריפט בסביבת דף
152.....	הסבר כללי על HTML
155.....	מצהים של תגיות HTML
155.....	גישה אל תגיות באמצעות ג'אויסקריפט
156.....	אלמנטים של HTML מתורגמים לאובייקטים של ג'אויסקריפט
157.....	AIROBJECTS עם HTML וג'אויסקריפט

159.....	הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווסקריפט
160.....	שינוי עיצוב
162.....	סלקטורים של DOM
165.....	אירועים נוספים
167.....	פעוף של אירועים
169.....	הצמדת אירוע ג'אווסקריפט לאלמנטים ב-HTML
175.....	דיבאג'ינג
181.....	אובייקטים גלובליים ואובייקטים מובנים
183.....	<code>parseInt</code>
184.....	<code>eval</code>
185.....	<code>Math</code>
185.....	<code>Date</code>
188.....	<code>JSON</code>
189.....	<code>setTimeout</code>
194.....	ביטויים ורגולריים
200.....	טיפול בשגיאות
203.....	<code>finally</code>
205.....	מבנה נתונים מסוג <i>Map</i> ו- <i>Set</i>
205.....	<code>Map</code>
206.....	<code>Set</code>
209.....	תכנות אסינכורוני – קולבקים
218.....	<i>Promises</i>
221.....	שרשרת הבטחות
224.....	קיוב הבטחות
227.....	פונקציית <code>async</code>
232.....	AJAX

234.....	מתודות של HTTP וארגוניים נוספים.....
236.....	ES6 Classes
241	ומה עבשו?
243	נספח: Best Practices
243.....	מה זה Best Practices ולמה כדאי לישם אותם?
244.....	שכל מפתח מקצועי צריך להכיר Best Practices
244.....	בחירה שמות
246.....	KISS
246.....	DRY
246.....	לא להמציא את הגלגל...
247.....	Make it work, make it right, make it fast
247.....	תיעוד
248.....	ניהול גרסאות
248.....	לבקש עזרה
249.....	ביקורת עמיתים – Code Review
250.....	Tech Design
250.....	Best Practices ואוטומציה
251.....	מה זה ?linting
252.....	ESLint
253.....	רישמה של Best Practices והחוק הרלוונטי של ESLint
253.....	בלי מספרי קסם
253.....	השווואה קפדיות: ===
253.....	קוד לא נגיש
254.....	חלוקת לחלקים קטנים
255.....	אורך מינימלי ומקסימלי לשמות משתנים
255.....	בלי eval
255.....	בלי משתנים שלא הוצהרו
257	נספח: בדיקות, יציבות ואיכות קוד
257.....	קצת רקע

257.....	מבנה בדיקות
258.....	בדיקות יחידה
259.....	בדיקות קצה ל Każה (End-to-End)
260.....	בדיקות ממשקי משתמש (UI Tests)
261.....	ספריות ופרימיוםרים מומלצים
261.....	סיכום
262.....	נספח: Corvid by Wix
262.....	הקדמה
262.....	מה בונים?
263.....	איך מתחילהים?
265.....	הציג נתונים מסך הנתונים באתר
266.....	Dataset
267.....	חיבור וርכיבים למידע מהטבלה
269.....	הוספת משימה חדשה
271.....	Events
274.....	wixData.insert()
277.....	שינוי סטוס המשימה
279.....	wixData.update() – עדכון רשומה בטבלה
279.....	מספר המשימות שלא הושלמו
280.....	wixDataQuery
280.....	wixData.query .1 – אתחול השאלה
281.....	2. בניית השאלה
281.....	3. הפעלת השאלה
283.....	\$w.onReady()
283.....	סינון המשימות לפי סטוס המשימה
285.....	wixData.filter() .1
285.....	2. בניית הסינון
285.....	3. הפעלת הסינון – dataset.setFilter()
287.....	ניקוי המשימות שהושלמו
288.....	wix-window
288.....	wixWindow.openLightBox()
290.....	wixWindow.lightbox.close()
291.....	wixData.remove() – מחיקת רשומה מהטבלה

292.....	נספח – יצירת מסד נתוניםים.....
295.....	Sandbox Live
296.....	Permissions
297.....	סיבום.....

על הספר

הספר "לلمוד תכנות ג'אוوهסקריפט בקלות" מלמד את השפה ג'אוوهסקריפט (JavaScript), שפה סקריפט קלה ופешוטה ללימוד שהפכה לפופולרית מאוד עם השנים. הספר מיועד לא-מתכנתים שרצוים להתנסות בשפת תכנות ולמתכנתים בג'אוوهסקריפט שרצוים להעמק את הדעת החיאורטי שלהם.

הספר מתחילה בלמידה בסיסי של משתנים ומגיע עד ללמידה של תכנות אסינכרוני ו-AJAX. בכל פרק בספר יש הסברים תיאורתיים לצד דוגמאות רבות הממחישות את העקרונות השונים שהוסברו בו, ובסיומו תמצאו שאלות דוגמה לתרגול עם הסברים מקיפים על הפתרונות.

נוסף על הספר קיימים אתר המכיל מאות תרגילים ופתרונות אשר יסייעו לכל הלומדים להשיג שליטה בעקרונות השפה ובתחריביו שללה. הספר מיועד להביא אדם שלא מכיר את ג'אוوهסקריפט לנקודה שבה הוא מכיר היטב את השפה ואת התחריביה שללה וידעו איך להשתמש בה כדי לפתור בעיות בסיסיות.

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

על המחבר

רן בר-זיק הוא מפתח תוכנה משנת 1996 במגוון שפות ופלטפורמות ועובד כמפתח בכיר במרכזי פיתוח של חברות רב-לאומיות, מ-Verizon ועד HPE, שם הוא מפתח בטכניקות מתקדמות הן מצד הלקוחה, הן מצד השירות, ושם דגש על בניית חשתית פיתוחה וכוננה, על שימוש ב-CD|CI וכמו כן על אבטחת מידע.

נוסף על עבודתו כמפתח במשרה מלאה, רן הוא עיתונאי ב"הארץ" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטכנולוגיה ולאבטחת מידע וכותב על אינטרנט ורשתות.

משנת 2008 מפעיל רן את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרי והסבירים על תכונות בעברית ומתעדכן לפחות פעם בשבוע.

רן נשוי ליעל ואב לארבעה ילדים: עומר, כפיר, דניאל ומיכל. רץ למרחקים ארוכים וחובב טולקין מושבע.

על העורכים הטכניים

דניאל שטרנלייכט

דניאל שטרנלייכט הוא מפתח Frontend מאז גיל 14, המיסד של המיזמים Common Ninja ו-*for that*, צרך כבד של קוד (ולתרום בהזורה) ונכון לזמן כתיבת הספר מוביל את גילדת ה-Frontend בחברת אוטבריין.

נסף על כן, דניאל הקים ומוביל את קבוצת הווטסאפ FEDs Community שמאגדת מפתחי Frontend מחברות מובילות בארץ ובעולם, ובה מעלים דיוונים, מתיעצים ומשתפים לינקים, חדשות ועדכונים מעולם ה-Frontend. בעבר כתב בבלוג "יעצוב גרפי וטכנולוגיה", וכיום הוא כותב בלוג טכני על טכנולוגיות Frontend וען NodeJS בשאר הזמן הוא נשוי לרונה ואבא להדר ולאליל המתוקים, מתופף, גולש סקי, משחק כדורים, צופה בסדרות, קורא ספרי פנטזיה ומהה מדבר ומדקלם את כל סרטיו דייני בעל פה.

gil fink

gil fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert . sparXys Microsoft Developer Technologies MVP. כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע לפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות ליחידים ולה חברות המעוניינים להתחמות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official Course - MOC), מחבר משותף של הספר "Pro Single Page Application Development" (Apress) ושותף בארגון הכנס הבינלאומי AngularUP <http://www.gilfink.net>. לפרטים נוספים על gil: <http://www.gilfink.net>

יגאל סטקלוב

יגאל סטקלוב הוא מפתח Frontend ו-*Full Stack*, מוביל טכנולוגי ומנהל פיתוח מנוסה בעל ותק של יותר מעשור וחצי בתעשייה. כיום משמש מנכ"ל חברת Webiya. בעבר היה מmobiliary בתחום Frontend-Wix ו-Netcraft והוביל פרויקטי פיתוח רבים. יגאל פועל מאד למען קהילת ה-Frontend בארץ והוא אחד מהיזמים והארגוני של כנס ה-Frontend הבינלאומי הראשון בישראל, כנס (YGLF) You Gotta Love Frontend.

יום ביגלאייזן

מפתח Frontend ומעצב, נשוי, אב לשתיים וגר בתל אביב. תום התחיל את דרכו בתחום ה-Frontend בתחילת שנות האלפיים מכיוון פחות צפוי – בפיתוח אפליקציות בלואוהסקריפט למשרדים של יס בסטארט-אפ קטן. אחרי שזה נגמר, עבד בעצמאות ובכמה סטארט-אפים, הפק לאחד המומחיהם הראשונים בארץ ל-HTML CSS ולא היה בקיא ממנו באגים של CSS עם RTL באקספלורר 6. תרם לפרויקטים שונים בקוד פתוח, היה שותף בגיור פלטפורמת ניהול התוכן Droptel ובנה את אחד הטמפלטים העבריים הפופולריים באותה תקופה. תום גם עיצב את האיקונים שעדיין נמצאים בשימוש עד היום בנגן הוועידי או הפופולרי VLC בקוד פתוח.

לפני קצת יותר משבע שנים נחת ב-Wix, ובמשך השנים האחרונות עומד בראש צוות קטן ומוכשר שמוביל את תחום ה-Frontend במדיה – וידיאו, תמונה, וקטורים ואנימציות בפלטפורמת בניית האתרים של Wix.

צחי נמני

צחי נמני הוא מומחה בפיתוח, סרבר, אוטומציה ודב אופס מתמחה בתכנולוגיות הבאות: Java, Kotlin, Node Js, C#, Docker, Kubernets, Git, Selenium, Appium, Terraform ו Cypress CI/CD, Automation Development, Docker, Kubernetes ועוד. בנוספ, צחי מייעץ ומרצה לחברות ויחידים בנושאים של Docker and Kubernets כרגע הוא עובד על סדרה של הרצאות בנושא צחי מאמין גדול בקוד פתוח ותרם לפרויקטים שהוא משתמש בהם.

על ג'אוהסקריפט

ג'אוהסקריפט החלה את צעדיה הראשונים ב-1993 כשפה שפעלה בסביבת דפדפן ונוועדה להעшир דפי HTML. בג'אוהסקריפט יכולנו ליצור אнимציות ופידבק למשתמשים – כל דבר שהוא קשור לאתר אינטרנט דינמי, למשל דברים שנראים היום מאד טרייוויאליים ופשוטים כמו לחיצה על כפתור שעשויה פעולה כלשהי בדף. אף על פי שההתחלת שלה הייתה צנועה, ברנדון אייך, ממציא השפה, יצר אותה מלכתחילה כשפה גמישה מאוד. האמישות הזו, וגם חוסר ההבנה של רבים מהמתכנים שהשתמשו בה בוגר לעקרונות הבסיסיים שלה, גרמו ללא מעט מתכנים בהם שפות אחרות לזרזל בה. גם השם שלה לא סייע לתדמית. השם ג'אוהסקריפט נקבע מסיבות שונות בלבד – JAVA היא שפת תכנות פופולרית, ואנשי נטסקייפ חשבו שכק יוסיפו לתדמיתה. בפועל השם הזה לא ממש עוזר, ואין כמובן שום קשר בין ג'אואה לג'אוהסקריפט.

על אף ההתחלה הקשה, ג'אוהסקריפט הפכה לפופולרית מאוד. בשנת 1996, חברת נטסקייפ העבירה את השליטה בסטנדרטים של השפה אל ארגון ECMA, ארגון אירופי (היום בינלאומי) המתמחה בתיקינה. המהלך הוביל לשחרור הספסיפיקציה של השפה, שידוע בשם ECMAScript, וג'אוהסקריפט "התישראל" לפי התקינה של ECMAScript. משנת 1997, שנת שחרור ECMAScript, כפי שהיא מושמת בדפדפניים שונים, ECMAScript עוקבת אחר התקינה של ECMAScript, שהיא בעצם "תוכנית המתאר", וג'אוהסקריפט עצמה היא היישום. לכל גרסה יש מספר משלחה בצדך למילימ'ס (ראשי תיבות של ES – ECMAScript).

מיקרוסופט התנגדה בתוקף ליישום השפה וייסמה שפה משלה בשם Jscript בדף אינטרנט אקספלורר, שהייתה בנוייה בדומה לג'אוהסקריפט. למרות היריבות הגדולה בין אנשי מיקרוסופט לאנשי ECMA, שנוצרה כתוצאה מפיתוח שתי שפות שנשענות על שני תקנים מתחדים, העקרונות של ג'אוהסקריפט שלובו גם בגרסה של מיקרוסופט. הפופולריות של השפה עלה כאשר מקромדייה (יוצרת פלאש) שיתפה פעולה עם ארגון ECMA וישיבה את עקרונות השפה בשפת ActionScript, ששימשה את תוכנת פלאש שהייתה פופולרית מאוד אז.

בשנת 2008 נפגשו אנשי מיקרוסופט ו-ECMA באוסלו והחלו בשיחות שלום. בנגד לשיחות שלום אחרות שהתקיימו באוסלו, שיחות השלום האלו הסתיימו בהצלחה. תקן ES5, הגרסת הרביעית של ג'אוהסקריפט, שוחרר ויושם בכל הדפדפניים שהוא קיימים אז.

מאז, התפתחות השפה והתפוצה שלה הויצו דרמטית. דפדפן כרום, שמרין ג'אוהסקריפט באופן יוצא דופן, נכנס אל השוק בסערה ואפשר למפתחי ג'אוהסקריפט לכתוב סקריפטים שפועלים על מנוע V8 העוצמתי של כרום ולהריז ג'אוהסקריפט במהלך מסחררת. השימוש ב-XAJAX תקשורת אסינכרונית עם השרת – נכנס לפועל, החליף שיטות מישנות כגון Long pulling ואפשר לאתרים לספק חוותות שימושית מדהימות למשתמשים. בשנים האחרונות, פרימורקים וספריות ג'אוהסקריפט אפשרו פונקציונליות מורכבת מאוד וספריות אחרות אפשרו כתיבה של ג'אוהסקריפט גם לטלפונים ניידים ואףלו בклות. הראשונות שבספריות האלו נקראו MooTools ו-jQuery והן אפשרו לכל מתכנת לכתוב אפליקציות בקלות. הספריות האחרונות נקראות ריאקט, אングולר ו-Vue והן מאפשרות לבנות תוכנות מורכבות מאוד על גבי הדפדפן (צד הלקוח). ג'אוהסקריפט לא נתרכה מוגבלת רק לצד הלקוח, ככלומר לדפדפניים ולמכשירי קצה אחרים; השימוש של ג'אוהסקריפט לצד השרת, הידוע בכינויו node.js, הפך לפופולרי

גם בשרתים. ג'אווהסקריפט מРИיצה כיום אפליקציות מורכבות גם לצד השרת, במירוח אפליקציות שצרכו לבעזע קריאות ולשרת מילוני משתמשים. כיום אפשר למצוא ג'אווהסקריפט בכל מקום: באתירי אינטראנט, באפליקציות של טלפונים ניידים, באפליקציות המיעודות למחשבים רגילים וכמוון בשרתים. הביקוש למתכנת ג'אווהסקריפט נמצא בשיאו ואין זה פלא – אפשר לעשות בשפה זו המון דברים יישומיים כמעט מ一封. יש כל כך הרבה ספריות וכל עוזר, עד שכמעט בכל שבוע יוצאה ספרייה שימושית חדשה. בעוזרת ידע מועט אפשר לעשות הרבה מאוד. מה שהוא מושב הוא ידע בסיסי בשפה.

בשנים האחרונות, תקן ES מתעדכן בכל שנה ומתווספים אליו תכונות ושימושים חדשים. ספר זה מעודכן לגרסה האחרונה של ECMAScript. חשוב לציין שם התקן מתעדכן, אין פירוש הדבר שהעדכון החדש מופיע מיד בדפדפניים שMRIיצים ג'אווהסקריפט או בשרתים שMRIיצים ג'אווהסקריפט, אלא לווח זמן עד שהעדכונם החדש ביותר עושם את דרכם אל הדפדפניים/שרתים שוכלו מושתמשים בהם. אם שמעתם מפתחי אינטראנט "מקטרים" על דפדפניים ישנים – זו בדיקת הסיבה.

זה המנייע לכתיבת הספר. הבן שלי, כיום מתכנת בזכות עצמו, ניסה ללמד ג'אווהסקריפט מ一封 ולא הצליח למצוא ספרים בעברית. החומר שיש כיום בעברית בנוגע לג'אווהסקריפט הוא דל ומיושן. חלק מחוברות העוזר הנמצאות בבתי הספר מתייחסות לתקנים שהפכו להיות בשימוש בשנת 2007! התיחסות אמיתי לתקנים החדשניים ביותר של השפה, שיצאו בשנת 2017, אין בנמצא בעברית. התחלתי לכתוב הסברים עברori בני, ומפה לשם הבנתי שאני חייב לקחת את זה הלאה.

ג'אווהסקריפט היא שפה שקל ללמידה. בנגדו לשפות אחרות, לא נדרש בה סביבת שרת מורכבת או כלי פיתוח שעולים כסף. לא נדרש ידע מקיף במדעי המחשב. כל צורך הוא לפתח Notepad במחשב, לפתוח דףדף ולהתחליל ללמידה ולפתחה. צריך גם הדרך נcona ומשמעות עצמית. אני מקווה שהספר הזה תמצאו לפחות הדרכה נcona. המשמעות העצמית – עליהם. אני מאמין שככל שתתקדמו בספר תראו את פירות הלימודים, הניצוץ בענינים יתחזק ולא תצטרכו עוד משמעת עצמית – אתם פשוט תאהבו בג'אווהסקריפט בכלל ובפיתוח לווב בפרט. אל תלגו על הפרק שבו אני מסביר איך ללמידה; זה הפרק החשוב ביותר בספר.

אני מאמין לכם הצלחה רבה, בין שאתם מתכנתים בתחילת דרככם בין שאתם מתכנתים ותיקים שימושיים בספר כדי לשפר את הידע שלכם.

רון בר-זיך

איך לומדים

לא למדתי מדעי המחשב באוניברסיטה או במללה. לעומת האמת, עד גיל מאוחר מאוד לא למדתי תכנות בעזרת מדריך. רוב מה שאני יודע למדתי ללא הרכה, ומכמה סיבות: הראשונה היא שכאשר עשית את צעדי הראשונים בחכונה, בשנות 1996, החומר שהוא 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 הרכה היא למידה מאוד לא יעילה. הרכה משמעה לא רק מרצה מנוסה, אלא גם אתר אינטרנט שבו יש הסבר מקיים, פורום או קבוצה בראשת חברתית זו או אחרת (לא רק פייסבוק), שיש בהם אנשים מנוסים שיכולים לסייע או להפנות לחומר עוזר. היא גם מקום שבו אפשר לתרגל ולהתנסות. לרובו המזל, בימים אלו קיימים שלל חומרים, עוזרה וסיוע. גם הספר הזה הוא הרכה. לא תידרשו לצולול לתוך הדוקומנטציה של ECMAScript על מנת להבין את השפה, אבל גם בעזרת הספר תמצאו דרך טובה ללמידה.

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

ארגנו לעצמכם סביבת עבודה מסודרת

הפרק הראשון עוסק במבנה סביבת העבודה והוא הפרק החשוב בספר. ארגנו את המחשב שלכם וסדרו לעצמכם תיקייה מאורגנת שבה תשמרו את כל התרגולים. אם המחשב שלכם מקפץ התראות של עדכוני ג'אווה או משהו בסגנון דומה, טפלו בכך. וDAO שאתם יכולים להיכנס לאתר הלימודי شاملווה את הספר ושהסימה שלכם תקינה ופועלת.

קראו את הפרקים לפי הסדר

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

קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות

קראו את הפרק פעם אחת קרייה שוטפת. לאחר מכן קראו אותו שוב. הפעם קחו את כל דוגמאות הקוד, העתיקו אותן לסביבת העבודה שלכם ושחקו 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 מאחוריו מה שמנסים לעשות, אך ללא מימוש, הידע הזה יתנוון וייעלם, בדיקו כמו בשפת דיבור. אם לא תשימוש ותתרגם, גם

הלימוד התיאורטי המעמיק ביותר לא יהיה שווה הרבה. הקריאה הראשונה נועדה ללימוד התיאוריה, הקריאה השנייה נועדה לתרגול. לימודי שפה הוא לא מروع! קחו את הזמן, כתבו את כל דוגמאות הקוד ונסו לכתוב ככלו משלכם.

פתרו את התרגילים לדוגמה בסוף כל פרק

בסוף כל פרק יש תרגילים לדוגמה. נסו לפתור אותם. אל תהיו מהר ואל תרצו אל הפתרון אלא שברואם 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) המיועדות ללימוד ג'אווהסקריפט ולדיאן בה. מצאו את זו שהכי נוח לכם בה. אל תהססו לשאול שם שאלות. לא הבנתם משהו? דוגמה כלשהי לא הייתה מובנת? הפתרון לתרגיל לא היה ברור מספיק? שאלו שם, ובערבית.

אל תהססו לקרוא, למשל דיוון על MERCHANTABILITY, להשתתף בדיונים או לסייע למי שהידע שלו דל משלכם. אל תשכחו להבין את רוח הדברים בקבוצה ולא להציג או להעיק. רוב המשתתפים בקבוצה הם אנשים עובדים שלאו דווקא זמינים להודעות מיידיות.

הגינו לסדרנות ולМИטאפים

לא מעט חברות מארכנות בחינוך סדנאות, מיטאפים ומפגשים שבהם מתכנתים מציגים את ג'אווהסקריפט ומרצים עליה. כדי מאד להגעה למפגשים האלו, לא רק על מנת לשמע את התכנים ולהשתתף בסדנאות אלא גם להכיר אנשים אחרים בתעשייה. עם חלكم אתם תכתבו בקבוצות הפיסבוק לג'אווהסקריפט. קהילת מפתחי הג'אווהסקריפט בארץ היא קהילה מאוד שיתופית וקרובה, ורבים בה מכירים זה את זה.

תרגלו, תרגלו, תרגלו

הספר לא שווה הרבה בלי תרגול מكيف ושימוש בו. אל תעברו לפרק הבא לפני שתסייעו את כל התרגילים שקשורים לפפרק שקראתם. זה לא קרייטי, אלא סופר-קרייטי.

התקנת סביבת העבודה ודרך הלימוד

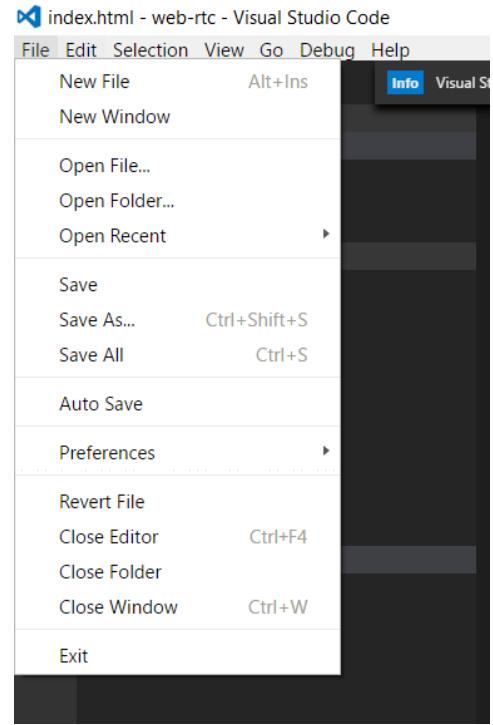
ג'אוوهסקריפט יכולה לזרז במסיבת שרת או דרך הדפדפן. איך זה בדיקע עובד? ג'אוوهסקריפט נמצאת בקובץ טקסט. כן, בדיק כמו זה שאפשר ליצור באמצעות כתבן הטקסט (Notepad) שיש בכל מערכת חלונות. בדף מותקן כל' שלוקה את קובץ הטקסט זהה ומריצ אותו ואת הפקודות שומצאות בתוכו. אם הדפדפן היה אדם, הוא היה פותח את קובץ הטקסט וקורא את מה שיש בתוכו, למשל: "לך ימינה ופתח את הדלת", ועשה בדיק מה שכתב. הפעולה זו נקרהת בלשון הפופולרית "רינדור", מלשון `render` בלען. הדפדפן לוקח את קובץ הג'אוوهסקריפט ומריצ אותו. קובץ הג'אווהסקריפט יכול לעשות כל מיני דברים ולהציג או לא להציג אותם.

איך הדפדפן טוען את קובץ הג'אווהסקריפט? יש כמה דרכם לעשות זאת, אבל כרגע ארצה למד אתכם איך לכתוב קובץ ג'אווהסקריפט ולראות אותו פועל על מנת להבין את כללי השפה ולכתוב משחו באופן ראשוןי ביותר. החלק החשוב והקשה ביותר הוא ייצרת סביבת העבודה, ככלומר סביבה ממוחשבת שבה אפשר להקליד ג'אווהסקריפט ולראות אותה עובדת. סביבה זו היא חשובה מאוד כאשר לומדים, כיון שלימוד של שפת תכנות נעשה ראשית כולם "דרך הידים" וחשוב מאוד לא רק לקרוא אלא גם לתרגל. וכייד לתרגל צריך סביבה שמאפשרת להקליד פקודות שפה, לשמר ולראות את הפלט. אני שב ומדגיש: התקנת הסביבה היא החלק הקשה ביותר בתחילת לימוד שפה חדשה וגם החשוב ביותר. לפיכך כדי להיאוז בסבלנות, לקחת נשימה ארוכה ולזכור שדווקא עכשו מתחודדים עם החלק הקשה ביותר.

הבה נתחיל בעורך טקסט טוב. אמןអ אפשר להשתמש בנוטפ, אבל הוא לא מציע צביעת קוד לצורך עזרה בקריאות ולא ייצור הזוחות בקלות. יש כמה עורכי טקסט המותאמים במיוחד לכתיבת ג'אווהסקריפט, שאציגן כמה מהם. בחרו בעורך טקסט אחד! רובם זהים למדי ומכליהם יכולת עירכה בסיסית של `HTML`, `CSS` וג'אווהסקריפט.

שיםו לב: מתכנים מנוסים לא משתמשים בעורך הטקסט הפשט אלא בעורך טקסט מסויל יותר שנקרא IDE או שגיואת בכתיבה וגם יכולה להריץ את הקוד עצמו.

מעולה הוא Visual Studio Code. הוא חינמי, מבוסס קוד פתוח, כתוב בג'אווהסקריפט (כן, כן), נתמך על ידי מיקרוסופט ונitin להורדה מה: <https://code.visualstudio.com/>. אחרי ההתקנה תוכלו לפתח את התוכנה, לבחור ב-File וואז לפתח את התקינה שבחורתם ולפתח או ליצור בה קבצים.

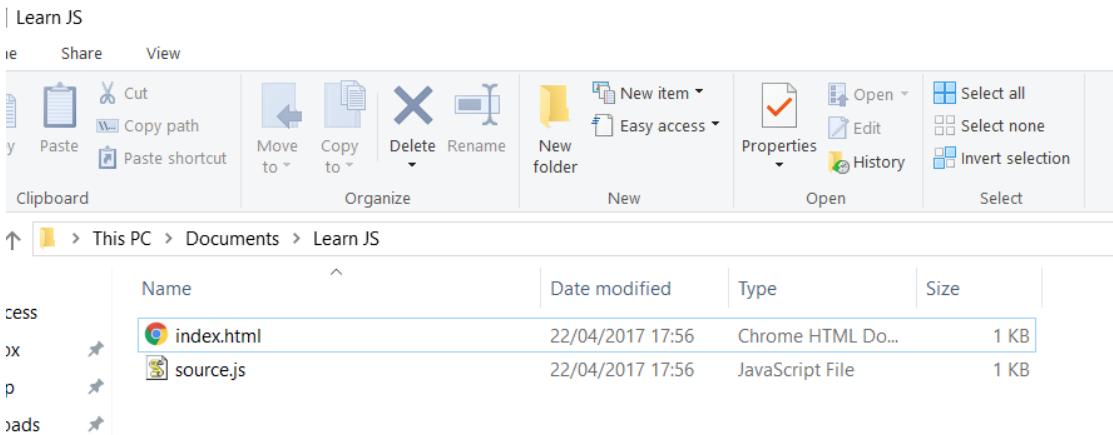


IDE מוצלח אחר הוא Atom. גם הוא חינמי וمبוסס קוד פתוח וגם הוא... כתוב ב-ג'אוהסקריפט. הוא נחנך על ידי גיטהאב וניתן להורדה מה [האתר](https://atom.io/). הוא דומה ל-Visual Studio Code ואחרי ההורדה והתקינה שלו אפשר להפעיל אותו בקלות.

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 משני קודמוני.

שימוש ב-Visual Studio Code או ב-Atom הוא מומלץ יותר כיון שיש בו "השלמה אוטומטית" של פקודות נפוצות ב-ג'אוהסקריפט, דבר המקל מאוד את הלמידה. כמו כן הוא מציג שגיאות בקוד כבר במהלך הכתיבה, עוד לפני הרצאה.

הסיבה הטובה ביותר ללימוד היא ייצרת קובץ HTML שטוען קובץ ג'אוהסקריפט. קובץ HTML הוא קובץ שהדפן יודע לפרש ולהציג, והוא יכול לקרוא לקובץ ג'אוהסקריפט באופן כזה שהדפן יрендר אותו. כאמור, רינדרו הוא הרצת הפקודות שנכתבות בקוד ג'אוהסקריפט. יש ליצור במחשב תקיה – זה יכול להיות ב"המסמכים שלי" או על שולחן העבודה – ובתוכה ליצור קובץ בשם source.js וקובץ בשם index.html.



ודאו שמערכת החולנות או המק שלכם תומכת בתצוגת שם הקובץ המלא, כולל הסיומת (.Extension). אחרת, קובץ index.html.txt יהיה בעצם index.html.

בקובץ HTML כתבו את הקוד הבא:

```
<!doctype html>
<html>

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

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

</html>
```

בתוך קובץ source.js כתבו את הטקסט הבא:

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

אחרי לשמורם את תוכן שני הקבצים, פתחו את הקובץ בדפדפן כרום או בפיירפוקס (לא באזג'). אם הכל נכון, תראו שכותב על המסך "Hello World!" – כתבתם את הגאווה סקריפט הראשון שלכם!

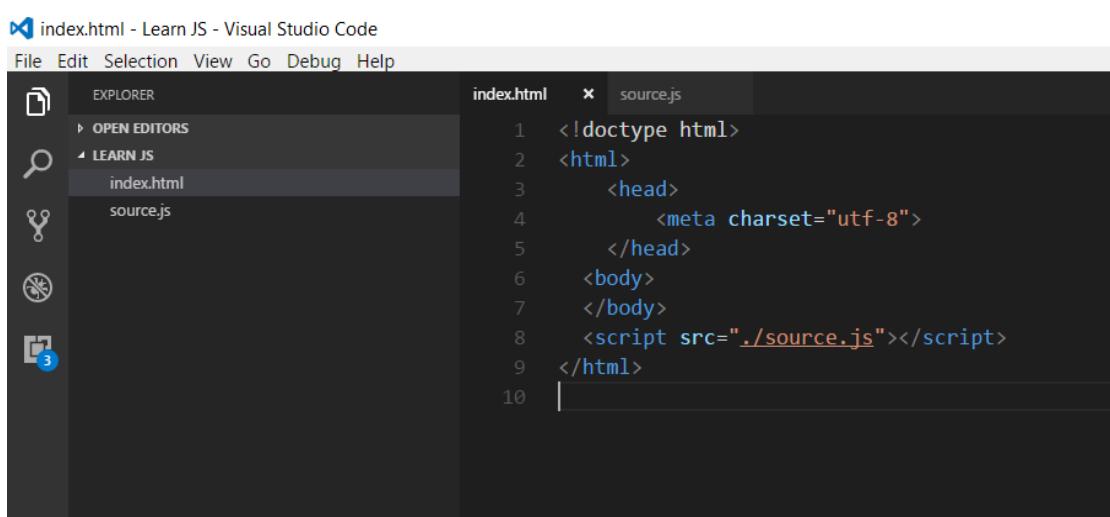
שיםו לב: זה השלב המועדף ביותר לפורענות, שעלול להיות מתסכל מאוד, אבל הוא שלב חשוב ביותר ואסור להרים ידיים ולהתיאש. אם פתחתם את הקובץ ודבר לא הופיע, נסו את הדברים הבאים:

בדקו שגם קראתם לקבצים index.html ו-source.js. הבדיקה צריכה להתבצע באמצעות הכפתור הימני של העכבר בחלון, כדי למנוע מצב שבו מערכת הפעלה הוסיפה תוספות לשמות index.html ו-.index.html.txt.

1. בדקו WHETHERם פותחים את הקבצים בדףן כרום או בפיירפוקס. בדקו WHETHER אין תוספים מיוחדים לדףדף.
2. בדקו WHETHER הגדלתם את הטעסט כשרה בקובץ source.js ללא רווחים או תווים מיוחדים.
3. נסו להשתמש ב-Atom או ב-Visual Studio Code. שימו לב WHETHER התראות על שגיאות הקלדה. CAN למשל מובאת דוגמה של שגיאת הקלדה שביצעת. אם קיבלתם התראה כזו, בדקו שוב WHETHER טיעתם בגרשיים ושלא הכנסתם רווחים מיותרם:

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

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



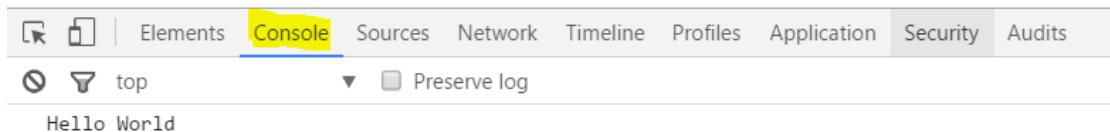
מצד שמאל תוכלו לראות את כל הקבצים בתיקייה. כדי ליצור תיקיה מיוחדת בשם learnJavaScript או בשם דומה ולא לשים את כל הקבצים בתיקייה משותפת כמו "המסמכים שלי".

מצד ימין תוכלו לצפות בתוכן הקבצים. בצלום המスクראם את תוכן הקובץ index.html. אם תקלידו דבר מה, תוכלו לראות שיש השלה אוטומטית של קוד HTML, source.js בקובץ source.js שמכיל את ה'אואהסקריפט תראו שיש השלמה אוטומטית של פקודות ג'אואהסקריפט. נוסף על כן, תקבלו גם התראה על קוד לא תקין.

כיוון WHETHERם כבר מפתחי ג'אואהסקריפט מקטועים, כדי שתלמדו להשתמש בكونסולה של הדףדף. מדובר בממשק מיוחד שמאפשר "לדבג" את ג'אואהסקריפט. הפעול "לדבג" כוננו להסתכל על צפונות הרינדור ולראות ממש את הפלט של השפה. נשמע מסובך? יש להבין איך הדףדף מנסה להרייך את הקוד שלו (מה שנקרא "רינדור", מלשון

הרצאה, באנגלית) ולקבל מידע נוסף במקרה שהוא נכשל, ככלומר "זורק" שגיאת צבואה באודם בקונסולה ואף מפנה לשורה הביניתית.

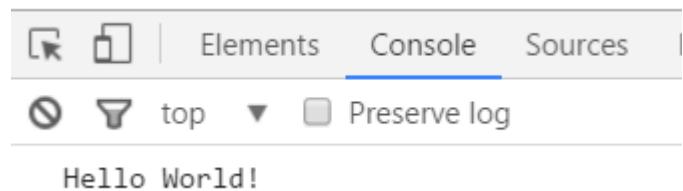
פתחו את כל המפתחים של הדפדן. בכרום ובפיירפוקס לחצו i + $Ctrl + Shift$ אם יש לכם חלונות או i + $Shift$ אם יש לכם מק. אפשר לעשות את זה גם דרך התפריט העליון בשני הדפדים. לאחרפתיחה כל המפתחים, לוחצים על Console.



הבה נבדוק מה אפשר לעשות בעזרת הקונסולה. היכנסו אל source.js והליכפו את הטקסט ל:

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

טענו מחדש את הדף באמצעות $Ctrl + F5$ בחלונות או $Cmd + R$ במק. הטעינה מחדש חשובה. הדפדן לא יודע שהוכנסו שינויים בקובץ ג'אווהסקריפט, ויש לגרום לו לטעון מחדש את קובץ ג'אווהסקריפט על מנת להריץ את הפקודות החדשות. אחרי הטעינה מחדש הסתכלו על לשונית ה-Console. אם הכל תקין, המסך יהיה ריק, אך בקונסולה תראו !Hello World !יש!



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

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

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

מכאן, דרך הלימוד תהיה פשוטה למדי. אני אסביר על תוכנות מסוימות של השפה ואתן דוגמאות. מומלץ בחום רב להעתיק את הדוגמאות אל קובץ `source.js` ולהריץ את ה'גלאוּסקריפט כדי לראות איך זה עובד באמת. בסוף כל פרק יש תרגילים לתרגול עצמאי ומומלץ מאוד לנשות אותו בסביבת העבודה. אי-אפשר ללמידה שפה על ידי קריאה תיאורטיבית בלבד ורצוי לתרגול, לתרגול. את התרגול עושים רק בסביבה עבודה יציבה. לפיכך, أنا אל תלגנו אל הפרק הבא לפני שיש לכם סביבה עבודה יציבה. מומלץ מאוד לעבוד ב-`Visual studio code` החינמית.

תרגילים:

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

פתרון:

היכנסו לקובץ `source.js`, מחקו את הטקסט שיש שם והזביקו במקומו את:

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

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

תרגילים:

גרמו לקובץ ה-HTML לטעון קובץ JS בשם targil.js וshedpis בקונסולה: I am new file.

פתרון:

צרו קובץ `targil.js` באוטה תיוקיה של הקובץ `index.html`. יש לוודא שהוא שמו האמתי של הקובץ ושמערכתו הפעלה לא מצמידה לקובץ עם סימנת `.txt`. את זה עושים על ידי בדיקה בהגדירות התוצאה של מערכת הפעלה. פתחו את קובץ ה-HTML בעזרת עורך טקסט (מומלץ להשתמש ב-`Visual Studio Code`) ושנו את שם קובץ `targil.js` לה'גלאוּסקריפט ל-`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 בדפדפן ולחצו **Ctrl + Shift + I** על מנת להציג את כל המפתחים. בחרו את הלשונית **Console** ובחנו את התוצאה.

משתנים

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

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

מה קורה פה? יש כאן הגדרת משתנה בשם `variant`. ההגדרה נעשית באמצעות המילה השמורה `let`. מילה שומרה היא מילה מיוחדת בשפה שהשימוש בה שמור למשך מסויים. במקרה זה, `let` שומרה אך ורק להגדרת משתנים. לאחר מכן מכנים ערך למשתנה. הפעולה הזו נקראת "הצבת ערך" או "השמה", והוא נקראת כך מכיוון שהערך מוצב בתוך המשתנה. במקרה הזה מדובר בטקסט הכלול את המילים `Hello World`.

שימוש לב: יש סימן ";" בסוף כל שורה. בג'אוהסקריפט מוטב להציב סימן ";" בסוף כל שורה כדי למנוע בעיות וכדי שהסקריפט יעבד. הוא מסמן סוף שורה עבור מנוע הג'אוהסקריפט שמנדר את הסקריפט, ממש כמו נקודה בסוף המשפט. על אף שלא כל מנוע מקפיד על כך, אני ממליץ לכם: הקפידו תמיד להקליד ";" בסוף כל שורה.

אפשר לזרר ולהציב את הערך מיד בהגדרת המשתנה:

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

הבה נבדוק את המשתנה ואת הערך שלו. אפשר להציב את המשתנה הזו בתוך `console.log` כפי שלמדנו בפרק הקודם:

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

אם תציצו בקונסולה, תראו שמודפס המשפט `"Hello World"`. מדוע? כיון שהוא מופיע בתוך המשתנה. מן הסתם, המשתנה 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" על ידי הערך החדש. הכנסתם (כלומר הצבתם) ערך חדש לתוכה המשנה, ועכשו מה שיש בתוכו השתנה. כשמציגים את המשנה 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 שימושים הוגדר, אי-אפשר להגדיר אותו מחדש. הקוד הבא:

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

יגרום לשגיאה הבאה: Identifier 'variant' has already been declared שמות המשתנים יכולים להיות מגוונים אך יש להם כמה כלליים מחייבים. אפשר להשתמש בכל אות שהיא ובסימנים _ או \$ בתחילת השם, ובכל האותיות והמספרים ובסימנים _ או \$ בתחילת 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 זאת זה גם כיון שהקוד שלכם יהיה פהות קרייא וגם כיון שהוא עלול להוביל להתנגדות מזורה ולצורות. אתן לכם טיפ: בעולם התכנות אל תחשוף צורות כי יש מספיק מהו גם כך.

שימוש לב: בגרסאות קודמות של גיאו-הסקרייפט השתמשו במילה השמורה var להגדרת משתנה. בתקנים החדשים כבר לא מקובל להשתמש ב-var כיון שלשיות בו יש השלכות שנדרן בהן בהמשך הספר. הוא נשאר איתנו בעיקר בשבייל תאים לאחר עברו קוד שנכתב בשנים עברו.

תרגילים:

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

פתרונות:

```
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 ה-`console.log` בפרק הקודם משמשת להדפסת הטקסט - ובמקרה זהה המשתנה שהוא מקבלת.

תרגילים:

צרו משתנה בשם myVar והכניסו לתוכו את טקסט "me not know JavaScript". דרשו את הטקסט זהה בטקסט `console.log("I know JavaScript")`

פתרונות:

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

הסבר:

יויצרים את המשתנה `myVar` ומכניסים לתוכו את הטקסט "me not know JavaScript" משברגע הייצור. לאחר מכן דורסים את הערך זהה באמצעות השמה נוספת. ההדפסה של מה שיש המשתנה נעשית באמצעות `.console.log`

תרגילים:

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 שנייהם באמצעות `:console.log` (בונוס: הדפיסו את שני המשתנים בקונסולה באמצעות פקודת `console.log` אחת)

פתרונות:

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

הסבר:

אין מניעה להגדיר כמה משתנים באותו סקריפט. הגדרתי את המשתנה באמצעות המילה השמורה `let`, שם המשתנה (הכולל מספר בסוף) והדפסה שלו ל-`console.log`.

הסבר לתרגיל הבונוס: נסו לחפש באינטרנט איך להדפיס ל-`console` שני משתנים. הרבה פעמים יהיו דברים שלא תבינו בתרגול. במקום לקרוא שוב ושוב את ההסבר, נסו לחפש תשובה בראשת. כך עושים מתכנתים מקצועיים: הם מהפשים תשובה בראשת.

טקסט

טקסט (באנגלית `text string`) הוא סוג מידע חשוב מאוד ובסיסי מאודמערכות מידע. משתמשים בטקסט בכל מקום ובכל מערכת, ומדובר בחלק חשוב מאוד בלמידת קוד – יותר מתמטיקה ומפעולות חשבוניות. בעבר היה קשה לעבוד עם טקסט שאינו אנגלי. אם אתם מבוגרים מספיק אתכם ודאי זוכרים כל מיני אותיות מיוחדות שהופיעו באתרים או בתוכנות. היום קל לעבוד במגוון שפות בתוכנות, אתרים וכמוון בגלואוסקריפט, בזכות תקן חדש שנקרא "יוניקוד". יוניקוד מאפשר לכתוב בעברית, ביפנית, ברוסית ובכל מערכת כתוב אחרת בקלות ובלוי חשש להופעת גיבריש והוא מוטמע במחשב שלכם באופן אוטומטי.

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

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

הטקסט מוקף בגרשיים בלבד, אבל אפשר להשתמש גם בגרשיים כפולים:

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

הבחירה ביניהם היא בדרך כלל עניין של טעם. רוב המהנדסים נהגים לנ强壮 להיום להשתמש בגרש בלבד להגדרת טקסט או בגרש מסווג (`backtick), שעליו נלמד מאוחר יותר. גלאוסקריפט מאפשר לחבר בין מחרוזות טקסט דיבוקים בעזרת הסימן + (כמו בפעולת חיבור):

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

מה שיזופס על הקונסולה הוא `foobar`. אף על פי שסימן ה"`+`" מוכר מהיבור מספרים, בג'אוּהסְקָרִיפֶט אפשר כאמור להשתמש בו לחיבור מחרוזות.

שימוש לב: הרוחה בין שמות המשתנים לבין הסימן `+` לא הכרחי להרצה תקינה אבל נוח מאוד לкриאה.

חשיבות ציין שהשם התקני של טקסט הוא "מחרוזת טקסט".

שימוש לב: השימוש ב-`foo` וב-`bar` הוא מאד נפוץ בדוגמאות בשפות תכנות ונדר למצוות מדריך לשפת תכנות שלא משתמש במילים האלה ובמילה `baz` למשתנים (או לחלקים אחרים בקוד שעוד תלמדו עליהם). דוגמאות האלו יש שם מפחד: משתנים מטה-סינטקטיים – אבל לא צריך לפחות ממשות מסוובכים.

נשאלת השאלה מה קורה אם רוצים להכניס מחרוזת טקסט עם גרש, שהוא בסגנון זהה: **לדוגמה:**

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

יהזיר שגיאה בקונסולה. כיוון שככל מחרוזת טקסט חיבת להיות מוקפת בגרשיים, המנווע של ג'אוּהסְקָרִיפֶט חושב שכל מה שמניע אחרי הגersh הוא משתנה ולא יודע מה לעשות אליו. בדוגמה הבאה, החלק המודגם הוא מחרוזת הטקסט, והחלק שאינו מודגם הוא מה שהנווע של ג'אוּהסְקָרִיפֶט לא יודע מה לעשות אליו:

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

יש כמה דרכים לפטור את הבעיה הזו. הדרך הטובה ביותר היא **לציין** לפני המנווע של ג'אוּהסְקָרִיפֶט שהגersh שיש במילה `don't` הוא גersh שהוא חלק מחרוזת הטקסט. איך עושים את זה? באמצעות הסימן `\`:

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

ההדפסה תציג את הטקסט כמו שצריך, עם הגersh. הפעולה הזו נקראת **escaping** ובה לוקחים טקסט שהוא שובר את שפת התכנות והופכים אותו לבתו. לחולפני, אפשר ליצור אליו תווים שלא ניתן כתוב במקלדת, למשל ירידת שורה. איך כתבים ירידת שורה? בעזרת `\n`. העתקו את הדוגמה הזו:

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

ותראו מה קורה. בקונסולה יופיעו שלוש שורות.

יתכן שתיתקלו ב-**escaping** במקומות נוספים. אם רוצים לכתוב רק `\`, צריך לעשות לו **escaping** ולכתוב `\\\`. במקרה מרגשים בנווה בכל מה שקשרור לטקסט או, נכון יותר, למחרוזת טקסט בג'אוּהסְקָרִיפֶט, הבה נסבר מעט את העניינים. בג'אוּהסְקָרִיפֶט מחרוזת טקסט נחשבת לסוג מידע. באנגלית זה נקרא **Data Type**. יש כמה סוגי מידע, כמו מספרים למשל, אבל סוג המידע של מחרוזת טקסט נקרא **string** והוא עומד בראשות עצמו. כאשר יוצרים משתנה ומכניסים לתוכו טקסט, בעצם אומרם **שהמשתנה הוא מסוג מחרוזת טקסט**. ברגע שימוש מסויים מקבל

לעצמו מידע והופך למשתנה מסוים, הוא מקבל גם סט של יכולות מיוחדות, ממש כמו קלארק קנט שהופך לסופרמן. מהשניה שנכנס למשתנה טקסט, אפשר לעשות בו כמה פעולות שאפשריות אך ורק לסוג המידע של הטקסט.

כך למשל אפשר לבצע על המשתנה המכיל את הטקסט פעללה שתగורם לכל האותיות שלו להיות ראיות (כלומר קפיטלס). לדוגמה:

```
let myVar = 'Hello World';
let myVarUpper;
myVarUpper = myVar.toUpperCase();
console.log(myVarUpper);
```

ויצרים משתנה עם טקסט. על המשתנה זהה עושים פעולה שאפשרית אך ורק עם משתנה מסוים טקסט. שם הפעולה היא ()`.toUpperCase()`. התוצאות של הפעולה זו מושמות למשתנה נוסף בשם `myVarUpper` ומודפסות. אם תעתיקו את הטקסט הזה ותריצו בקונסולה, תראו שהתוצאה היא 'HELLO WORLD'.

השימוש בהפעלה פועלות על משתנה עם Data Type מסוים אינו בלבד רק למחרוזות, ובהמשך הספר תראו שאפשר לבצע פועלות גם על Data Types אחרים. בין היתר צריך להבין שיש סט רחב מאוד של פועלות שאפשר להפעיל על מחרוזות, ו-`toUpperCase` היא רק אופציה אחת. על מנת להפעיל פועלות שונות משתמשים ב-. (נקודה) על המשתנה. אם משתמשים ב-IDE (עורך טקסט) כמו VS Code, כאשר תרשמו ". " אחרי המשתנה, עורך הקוד יראה לכם את כל הפעולות האפשרות על המשתנה זה.

שים לב שהפעולה `toUpperCase` לא משנה את המשתנה עצמו! שינוי המשתנה עצמו נקרא "מוטציה", וזה לא מה שעושים פה. כדי לקבל את הערך שהשתנה צריך להגיד המשתנה נוסף ולהכניס את הערך למשתנה.

מבלבל? הנה נדגים זאת שוב בפעולה נוספת. הפעם צרו משתנה והכניסו לתוכו טקסט. המשתנה הראשון, מהרגע שיש בתוכו טקסט, יכול לבצע פעולה של טקסט ולהחזיר את התוצאה אל המשתנה אחר שאותו תדפיסו:

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

יש פועלות נוספות שאפשר לעשות על מחרוזות טקסט, אבל כרגע התמקד בשתי הפעולות הללו גם בתרגול.

תרגילים:

צרו שני משתנים, אחד מהם מכיל את המילה Hello, והآخر מכיל את המילה World. חברו ביניהם, הכניסו את התוצאה אל המשתנה השלישי והדפיסו אותה.

פתרון:

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

הסבר:

יצרים ומצייבים ערכיהם במשתנים בשם myVar1 ו-myVar2 (יש להקפיד שמהרווזות הטקסט יהיו מוקפות בגרשיים בודדים). מגדירים משתנה שלישי בשם answer ובתוכו את הסכום של myVar1 ו-myVar2. הדפסה נעשית כרגע. שימו לב שרואה גם נחשב לאות. אם לא מכנים רוחה באחת מהרווזות הטקסט בתוך המשתנים, לא יהיה רווח.

תרגיל:

צרו משתנה שיכיל את מהרווזת 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:

שימו לב לשימוש ב-escaping! מה זה escaping? פשטן שימוש בסימן \ על מנת להודיע למנוע של ג'אווהסקריפט שהגרש הוא חלק מהרווזת הטקסט. ההשמה והדפסה של המשתנה נעשות כרגע.

תרגיל:

הכינוו 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:

יווצרם שני משתנים ריקים בשם myVar ו-answer. הראשוון מכיל את מהרזהת הטקסט JavaScript. אחרי שמכניסים לתוכו את הערך זהה, הוא מסוג טקסט ואפשר להשתמש בפעולתtoUpperCase. את תוצאות הפעולה זו מכניסים ל-answer ומדפיסים. אחרי הדפסה משתמשים בפעולתtoLowerCase על מה יש ב-myVar, מכנים את התוצאה אל answer ומדפיסים.

מספרים

בדוק כמו שאפשר להכניס טקסט לתוך 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, מחרוזת טקסט, שעליה למדנו קודם, מוקפת בגרש או בגרשיים. מספרים לא מוקפים בגרש או בגרשיים.

אם תדפסו את המספר כמו בדוגמה, תראו שירופס 12. כדי העין שביניכם ישימו לב שההדפסה בكونסולה נראהית מעט שונה כשמדבר במספר ולא במחרוזת טקסט. ההבדל נוצר בגלל השוני בין סוג המידע של מחרוזת טקסט לסוג המידע של מספר.

אם מחברים בין מספרים איזי החיבור ייעשה בדיק 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 שתודפס תהיה 11. 2 ועוד 4 ועוד 5.
בדומה לחיבור, אפשר לעשות פעולה חיסור בעזרת הסימן "-". למשל:

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

שימוש לב שהחטאה 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 akan 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 akan תהיה 0.5. לאוوهסקרייפט לא מפחד משברים עשרוניים, כמו כן, אפשר להגיד לו גם שברים עשרוניים.

אפשר לעבוד גם עם חזקות. חזקות מסמנים בג'אווהסקרייפט באמצעות "**":

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

התוצאה akan תהיה 100 כמו כן. 10 בחזקת 2.

בדומה לפעולות המוחדרות של מהרוות טקסט שראיתם שאפשר להפעיל על משתנים מסווג מהרוות טקסט, גם למשתנים מסווג ייש פועלות מיוחדות. למשל, פעולה המגבילה את המספר לאחר הנקודה העשרונית. אם תחלקו 10 ב-3, תקבלו 3.3333333 עד שלא ידע. אפשר להעיף את הנקודות העשרוניות. איך? באמצעות פעולה מיוחדת Math.round()

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

הבה ננתח 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 מכיל? foo, bar, שזה בעצם 10 חלקי 3. מה התוצאה של התרגיל זהה? 3.3333333, שנמצאת כרגע במשתנה answer.

בשורה הרביעית מפעילים את הפעולה Math.round על answer ומכוונים את התוצאה למשתנה finalAnswer. הפונקציה round מעגלת את המספר. למה? ל-3. אם תדפיסו את finalAnswer תקבלו 3.

הינה טבלה קצרה שמסכמת את כל הפעולות הבסיסיות שאפשר לעשות:

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

מציאת השארית

אם אתם זוכרים מתקופת בית הספר הייסודי, יש דבר כזה שנקרא שארית. כמה זה שיש חלקי ארבע? אחת ושארית שתים. כמובן, החלק מהמספר השלם שהוא קטן מהחלק. תשע חלק שמונה זה אחת עם שארית אחת. תאמינו או לא, בתרנגולות יש עדנה לדברים שלומדים בכיתה ד'. אפשר לחלק ולמצא את השארית בעזרת סימן מיוחד – %. הסימן הזה נקרא על ידי המתכנתים "מודולוס" (באנגלית modulus).

אפשר לקרוא לו שארית:

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

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

הפעולות המתמטיות שנלמדו עד כה – חיסור, חילוק והעלאה בחזקה – נקראות אופרטורים (באנגלית operators).

אפשר להציג את תוכנות האופרטורים ישירות בתחום המשתנה. למשל, אם רוצים להציג בתחום המשתנה את התוצאה של 4×2 , אין צורך ליצור משתנה שיכיל 4, משתנה שיכיל 2 ומשתנה שיכיל את המכפלה של שניהם, כמובן משווה כזה:

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

במקרהו, אפשר לכתוב משהו כזה:

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

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

אופרטורים מקוצרים

נניח שיש משתנה שרוצים להוסיף לו רק ספרה אחת. אפשר להשתמש באופרטור מקוצר להוספה או להחסרה. האופרטור להוספה הוא `++` והוא נראה כך:

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

מה יש פה? מגדירים משתנה בשם `answer` והערך שלו הוא 4. עם האופרטור `++` מוסיפים לו 1. ערך המשתנה החדש הוא 5. באותו אופן כמו האופרטור `++` קיים 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 שהזרום על התוצאה שלוש פעמים, המשתנה שווה ל-7. הבעה היא שזה מכוער... יש גם אופרטור קיצור שיכל להוסיף או להוסיף איזה מספר שרוצים. 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 המקוצר מורידים ממנו 3. התשובה היא 7.

הינה טבלה חלקית של האופרטורים המזוקצים הנפוצים ביותר:

משמעות	אופרטור מזוקץ
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'אויסקריפט היא שפה סלחנית מאוד. אם מנסים לחבר בין משתנה מסווג טקסט למשתנה מסווג מספר היא לא תעיף שגיאה אלא תמיר באופן אוטומטי את המספר לטקסט ותבצע החיבור. כך למשל:

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

תיתן את התשובה המדהימה מהרוזת טקסט של 11. למה? כי bar הוא מסווג טקסט. אם אתם כבר מכירים שפת תכנות, זה השלב שבו אתם מתמלאים בזעם או בתדעה. אבל כאמור ג'אויסקריפט היא שפה סלחנית ותנסה להגיע לפשרה אם מדובר בחיבור בין טקסט למספר. בחיבור בין מספר לבין חלק מסווג המידע האחרים, תתקבל שגיאה של NaN, שהיא ראשי תיבות של Not a Number. זה נראה שונה לעומת מילון, אבל ג'אויסקריפט היא שפה שסוגי המשתנים בה הם implicit – כלומר המנווע של השפה מנשה לנחש אותם ולא מהר לזרוק שגיאה כמו בשפות אחרות. כאמור, משתמש ללא מילון עלול להתבלבל, ולמצבנויות בשפות אחרות זה נראה כאוטי, אבל ברגע שמתרגלים – מדובר בתוכנה 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:

יווצרם שלושה משתנים ושמיהם בתוכם מספרים באופן מוקוצר כפי שלמדנו. שלושת המספרים מחוברים וההתוצאה נכנסת למשנה answer, שבתורו מודפס.

תרגיל:

צרו משתנה, הציבו בתוכו את המספר 12 והדפיסו אותו. באותו משתנה הציבו מהרוזת טקסט של 12 והדפיסו אותה.

פתרון:

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

הסבר:

כאשר מציבים מספר במשנה, לא משתמשים בגרשיים. כאשר מציבים מהרוזת טקסט משתמשים בגרשיים. ההדפסה נעשית באותו אופן, אך כאמור, ברוב הקונסולות תראו הבדל בין שתי ההוראות שנoved לשם שהדפסה אחת היא מספר והאחרת היא טקסט.

חשוב: צריך לשים לב שסוג המידע של המשתנה השתנה וכדי להימנע מלזהב 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.

פתרון:

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

הסבר:

מציבים 100 ו-0.5 בתחום משתנים. שורש, למי שלא זכר מתמטיקה, הוא חזקת חצי. התשובה היא 100 בחזקת 0.5
ואני מתביחס בעצמי על זה שהכנסתי תחכום במתמטיקה לתרגיל כאן.

תרגיל:

צרו משתנה שיכיל את השארית של 10 חלקי 3.

פתרון:

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

הסבר:

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

תרגיל:

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

פתרון:

```
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 ומצביעים בו את המספר 10. השתמשתם באופרטור המקוצר `+=` להוספה 1 בדיק למשתנה זהה. אפשר להשתמש באופרטור המקוצר `+=` כדי להוסיף 1.

סוגי מידע פרימיטיביים נוספים

בפרקים הקודמים למדנו על מידע מסווג מחרוזת טקסט ומספר. סוג מידע נוסף נקראים "סוגי מידע פרימיטיביים". כאשר מודים שישו מידע שהוא פרימיטיבי לא מתכוונים לכך שהוא לא אוכל בסכין ובמזלג אלא שהוא מהויה ייחודה במידע בסיסית ולא מחוcharה שבהה עם השפה. חז' מספר ומחרוזת טקסט יש עוד כמה סוגים מידע כאלה.

بولיאני

סוג מידע בוליאני הוא סוג מידע שמכיל אחד משני ערכים, `true` או `false`, כלומראמת או שקר. כך מצייבים אותו:

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

שימוש לב ש-`true`, בדיק כmo מספר, אינו מוקף בגרשיים. אם הוא היה מוקף בגרשיים הוא היה מידע מסווג מחרוזת טקסט. פעללה מיוחדת שיש לסוג מידע בוליאני היא המרת למחרוזת 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 גרשים כיון שהוא מילה שמורה. אם תדפיסו אותו, תראו שהקונסולה מראה אותו כ-`true`. מהרגע שהמשתנה הזה 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. כדי העין יבחן יש הבדל בקונסולה בין ההדפסה הראשונה לשניה. הסיבה היא שבהדפסה הראשונה המשתנה הוא בוליאני ובהדפסה השנייה הוא מחרוזת טקסט. בהמשך הספר אשוב למשתנים בוליאניים ואdon במצבים שבהם כדאי להשתמש בהם.

משתנה לא מוגדר

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

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

אפשר לאפס משתנה בעזרת המילה השמורה undefined:

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

אין פעולות שאפשר להצמיד ל-undefined.

ריך

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

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

כאן הדרישה 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 כרגעמשמעותי אך הוא ממשוני מאד. undefined משמעתו משתנה שהוגדר אך לא אוחת בערך כלשהו. null משמעתו משתנה שהוגדר ואוחת בערך null. כדאי לזכור שמדובר בשני סוגים ייחודיים להלוטין. המידע הזה יהיה חשוב בהמשך הדרכך. כדאי לציין שיש כאן שחוובים שמדובר במקרה בעיצוב השפה ושה-null לא אמור להתקיים.

Symbol

מדובר בסוג חדש של מידע פרימיטיבי שנכנס לשפת JavaScript בשנת 2015 (ES2015). מטרתו היא ליצור משתנה בעל ערך ייחודי להלוטין. הערך זה לא מחרזות טקסט, לא מספר ולא משהו אחר:

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

במקרה הזה, foo ו-bar הם משתנים שקיבלו לכואורה אותו ערך, אבל הם שונים להלוטין. () נתן למשתנה ערך ייחודי שאפשר להשתמש בו בהמשך. במקרה הזה כל הסיפור נשמע מאוד ערטילאי, אבל אני מבטיח שהוא יתבהר בהמשך. מה שכן – מדובר בסוג מידע פרימיטיבי שאיןו שונה מהותו מכל סוג מידע שדיברנו עליו עד כה.

שימוש לבי: Symbol הוא סוג מידע שנכנס רק בגרסאות האחרונות של ג'אווה סקריפט ושימושו כאשר רוצhim ליצור מזהה ייחודי. חשוב להכיר אותו.

מציאות הסוג של המשתנה

בכל רגע ניתן לאפשר למצוא את סוג המשתנה באמצעות האופרטור typeof. בדומה לאופרטורים המתאימים לסוג המידע מספר, גם הוא אופרטור, אבל אופרטור שעומד על כל סוגי המידע ומהזיר 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 והוצאה בו מחרוזת הטקסט "Hello World". מהרגע הזה,(foo) 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 ועדיין להשתמש בו אך ורק לסוגי מידע פרימיטיביים.

תרגיל:

צרו משתנה בוליאני והכינוו 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:

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

תרגילים:

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

פתרונות:

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

הסבר:

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 ננסה להדפיסו תקבלו `.undefined`.

תרגילים:

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

פתרונות:

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

הסבר:

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

תרגילים:

צרו משתנה `foo` והכניסו לו מהרזה טקסט. הדפיסו את ה-`typeof` של מהרזה הטקסט בקורסולה. לאחר מכן הכניסו למשתנה מספר, וגם כאן הדפיסו את ה-`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 ובודקים אותו באמצעות typeof שערךו נכנס ל-bar. אפשר לראות שבהתחלת הפעלה הוא undefined. כמשמעותם לתוכו מספר, סוג המידע בו הוא number וכמשמעותם לתוכו מחרוזת טקסט, שהוא שונה מהמספר בכך שיש סביבה גרשימית, והוא מסוג string. כמשמעותם ערך בוליאני, שהוא המילה השמורה -true, הטעתו הוא Boolean, וכמשמעותם Symbol הוא הופך לסוג Symbol. לבסוף מקבלים את הערך null. הערך null הוא מציין היסטוריות שלא אונס אליו פה.

הערות

ברוב המקרים רוצים לכתוב הערות בקוד, טקסט שיסביר כל מיני דברים על הקוד בלי שירונדר. למה צריך את זה? לפעמים כדי לכתוב הסברים לזמן מאוחר יותר, לעיתים עבור מתכנתים 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 החזאה תהיה undefined, כי ההגדלה של a נמצאת בהערה. מקובל בהערות לשמר על רווח בין שני תוווי הערה // להערה עצמה, אך אין חובה לעשות זאת.

לעתים רוצים לכתוב הערה בכמה שורות. לפיכך משתמשים בתווים /* */ ו-/*/. בתחילת גופ הערה כתבים */ ובסופה */:

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

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

במשך אשותם בהערות על מנת להציג את הערכים השונים. כך למשל אם ארצת לציין שבשלב מסוים משתנה שווה לערך כלשהו, אכתוב את זה בהערה במקום לכתוב console.log. למשל:

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

כאן נוצר משתנה foo והוצב בו הערך המספרי 11. באמצעות האופרטור המקוצר ++, שכבר למדנו עליו, נוסף 1 למשנה זהה. על מנת לציין את הערך החדש נוספה הערה.

תרגילים:

בדומה לתרגיל הקודם, צרו משתנה ובאמצעות typeof שנו את סוג המידע שלו שש פעמים לשישה סוגים מידע 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 מאוד להבין מה `bar` מכיל.

בקורת זרימה – משפטים תנאי

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

למשל, הבה נניח משתנה foo מסוים. אם הוא שווה ל-5, יודפס בקונסולה "This is five". איך עושים את זה?

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

משפט התנאי if מאפשר לשאול שאלה, במקרה הזה אם foo שווה ל-5. סימן השווון הוא === והוא בודק אם מדובר במספר המבוקש, במקרה הזה 5. אם התנאי נכון כל מה שקרה בסוגרים המסלולים מתרחש. אם לא, לא קורה כלום.

שימוש לב: === נקרא אופרטור. כן, בדיק כmo האופרטורים של החיבור, החיסור, הכפל ודומיהם, שלמדנו בפרק המספרים. האופרטור הזה הוא אופרטור של השוואה, ובניגוד לאופרטור נומרי (של מספר) הוא מוחזר ערך בולייאני. כשלמדנו עליו הוא היה ערטילאי מאד. עצשוו הוא קצר יותר הגיוני, אבל רק קצר. בקרוב הוא ייחקר לעומק.

בדוגמה שלහלן, אם מדובר במספר מסוים אז בקונסולה יודפס "זה 5", ואם לא, בקונסולה יודפס "זה לא 5". גם פה זה עושים את זה בקלות, 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 כתוב בסוגרים המסלולים מיד אחרי ה-if. אם לא, מתרחש מה כתוב בסוגרים המסלולים אחרי ה-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 בתוך הסוגרים המסלולים הראשוניים יופעל והקונסולה תקבל "This is five". אם לא – רק הקוד שבסוגרים המסלולים לאחר המילה השמורה 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 היא תדפיס שהוא לא זה ולא זה. איך עושים את זה? באמצעות else-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,قطع הקוד שבתוך הסוגרים המולסים הראשונים יופעל אבל כאן יש גם עם עוד תנאי, שבודק אם foo שווה ל-6. בסופה של דבר יש תנאי שתופס את הכל. אפשר לשלב כמה else-if שרצוים, 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"ך. שימוש לב שיש תנאי if else מרובים בקוד הזה, 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 שם חוסיפו סהם else באמצע, לעולם לא הגיעו ל-if else שמופייע אחריו.

עוד משהו שחשוב לציין הוא שצורך להיזהר מהשימוש בתוך משפט תנאי. ג'אויסקריפט היא סלחנית מדי, ואם כתבו `y==x==y` הקוד שלכם יעבד ולא יזרוק שגיאה – זו הסיבה שבאופרטורים השוואתיים צריך לשים לב היוט מה כתובים ולעולם לא לכתוב = אחד בלבד.

אופרטורים השוואתיים נוספים

כמו `==` שבודק השווותה טהורה, יש אופרטורים השוואתיים נוספים, למשל `!=` (סימן הקריאה משמאלי לשני סימני השווה) שבודק אי-שווון 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 הזה הוא שווה ל-`4`, וקטע הקוד בתוך הסוגרים המסלולים יפעל. אוישר גדול! 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 שבתוך הסוגרים המסלולים יופעל.

יש גם אופרטורים שבודקים גדול/קטן מ-.

```
let foo = 10;
if (foo > 1) {
    console.log('Bigger than 1');
} else {
    console.log('Smaller 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 שבתוך הסוגרים המסלולים יפעל.

מה לפि דעתכם יודפס בקונסולה אם תרצו את הקוד הבא?

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

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

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

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

אפשר גם להשתמש באופרטור נוסף שמשמעותו גדול או שווה:

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

כאן אפשר להשתמש באופרטור `=>` שמשמעותו גדול מ- או שווה ל-. במקרה זהה גדול מ-5 או שווה לו, וכיון ש-`foo` שווה ל-5 התנאי מתממש.

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

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

מישפט התנאי בודק אם `foo` קטן מ-5 או שווה לו. במקרה הזה, כיוון ש-`foo` שווה ל-10-, התנאי מתממש, ועל הקונסולה תראו `"Smaller or equal to 5"`

הבה נבחן את כל האופרטורים ההשוואתיים שהכרתם:

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

הסביר מאחורי הקלעים בקשר לאופרטורים ההשוואתיים האופרטורים ההשוואתיים בעצם מתרגםים את הביטויים שמעבירים להם לערך בוליאני, כולם אמת או שקר. למשל:

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

זאת אומרת שכמו שהאופרטור הנוומי מוסיף או משנה מספר, האופרטור ההשוואתי מחזיר תמיד `true` או `false`. זאת 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 ההשוואתי 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 מזמן, אני יודעת, אבל זה חשוב מאוד להבין של משפט התנאי. בגודל, if ו-else קובעים את זרימת הקוד באמצעות משתנים בוליאניים.

אופרטורים לוגיים

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

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

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

הבה נדגים זאת שוב במחוזות טקסט. נניח שתרצה לבדוק אם המשתנה מסוים שווה למחוזת הטקסט "sunday" או "Sunday". איך עושים את זה? ממש פשוטות. צריכים שאחד משני התנאים האלו יתמשח:

```
foo === 'sunday';
```

או:

```
foo === 'Sunday';
```

ممמשים את ה"או" באמצעות אופרטור לוגי של "או", שכאמור 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. למשל:

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

כאן למשל אפשר לראות שהתנאי יתמשח אם foo יהיה שווה ל-sunday או ל-1 או ל-true.

מה קורה מאחוריו 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 לו.

כדי לשים לב שההשוואה היא משמאל לימין. אם התנאי הראשון מתמשח, אין בדיקה של התנאי השני.

אופרטור לוגי נוסף הוא האופרטור "גם", שבו אפשר לקבוע שני תנאים יתملאו יחד. אם אחד מהם לא מתמלא, האופרטור הלוגי יחזיר false ומשפט ה-if לא יתקיים. האופרטור הלוגי "גם" כתוב כך: &&. איך כתובים 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 משפטי תנאי עם אופרטורים השוואתיים כמו שאנחנו מכירים וביניהם, כמו דבוק, לחבר האופרטור הלוגי && שימושו "גם", כלומר גם זה וגם זה. שני התנאים חייבים להתמלא על מנת שמשפט התנאי יחזיר true ויתמשח. אם רק אחד מהם לא יהיה נכון, האופרטור הלוגי יחזיר false והתנאי ייכשל.

אפשר לשלב בין התנאים באמצעות סוגרים, ממש כמו במקרים מסוימות שלומדים בבית הספר. כך למשל כתובים משפט שבו בודקים אם foo או bar שוויים ל-1 או 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 הראשונים יש אופרטור לוגי של "או", שיחזר true אם אחד מהם נכון. בשלב הראשון של הרינדור, המנו יבצע את הבדיקה של:

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

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

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

אם גם `baz` יהיה שווה ל-1, התנאי יתממש.

אפשר להתפרק כמה שרוצים 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 מרכיבים כאלו, סימן שימושו לא בסדר בקוד. קוד אמרור להיות קרייא, ואף על פי審 משפט תנאי שיש בו מרכיבות כזו הוא אפשרי, לא בטוח שמדובר בנהג נכון.

אופרטור שלילי

אופרטור לוגי נוסף ו שימושי מאוד הוא אופרטור לוגי שלילי. האופרטור הזה קצת קשה להבין, אך יש לו חשיבות רבה ושווה להשקיע זמן בהבנתו. בגדול, האופרטור הלוגי הזה לוקח כל ערך בוליאני והופך אותו. למשל:

```
!true
```

יהיה בעצם `false`. אפשר להשתמש בו במשפט תנאי כדי לומר הפק. 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 המתכוונים משתמשים באופרטור הזה על מנת להפוך תנאים מרכיביים. 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 !**. הנה דוגמה:

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

אמנם יש שפות שמאלצות שימוש בתנאי שלילי, אבל הפיכת תנאים נחשבת מנהג פסול מאוד ולא הייתה ממליצה להשתמש בה. כדאי לדעת את זה על מנת להבין את פעילות האופרטור ! שבסופו של דבר ממיר ל-true/false, אבל מומלץ מאוד לכתוב את התנאים כמו שצורך. למה כן משתמשים באופרטור הזה? הוא שימושי מאוד לבדיקה אם משתנה מסוים הוא "ריק" ולמנוע else מיותר. כשאתם ניסו 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 מחרוזת ריקה, 0, null או אפילו משתנה שלא הוצב בו ערך. כל הערכים האלה ניתנים לבחינה במתוך 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 הערכים 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
	או
&&	וגם
!	אופרטור שלילי

אופרטור תנאי

אופרטור תנאי, "if" מקוצר או ternary operator הוא האופרטור המשמעותי האחרון שתלמידו. 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, אז 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, שבהם מבצעים הצבת ערך, לבצע את הפעולה שלעיל בעזרת משפט תנאי מקוצר:

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

ועכשו הסבירו: משפט תנאי מקוצר מתחילה בשם של משתנה, סימן שווין ואז תנאי עם סימן שאלה. מיד אחרי סימן השאלה מופיע מה שנכנס לתוכה המשתנה אם התנאי מוחזר true, ואז נקודתיים ומה שנכנס לתוכה המשתנה אם התנאי מוחזר false.

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

```

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

```

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

אופרטורים המשווים ערכיים

יש סוג נוסף של אופרטורים שימושיים בהם פחות ואני באופן אישי לא משתמש בהם בכלל. עברור עליהם בקצרה כיוון שהם מופיעים לא מעט פעמים, אבל השימוש בהם נחשב למנגנון פסול. נניח שיש את שני המשתנים הבאים:

```

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

```

האם הם שווים? אם עברתם על סוג מידע, 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:

```
foo++;
```

יעבור, כי אפשר להשתמש באופרטור נומי (במקרה הזה ++ מוסף אחד) על מספר, אבל:

```
bar++;
```

יכשל ותויפי שגיאת NaN, שימושו הוא "לא מספר". כי "1" זה לא 1.
לא מובן? דוגמה נוספת:

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

המשנה foo שווה למספר 1, המשנה bar שווה למחרוזת הטקסט 1. אם מוחרים שני מספרים שהם 1, התשובה היא 2. אם מוחרים שתי מחרוזות טקסט של 1, ג'אוהסקרייפט מחבר את מחרוזות הטקסט ואז 1 ועוד 1 שווה ל-11. אבל כיוון שאתם מתכונתי ג'אוהסקרייפט מנוסים, אתם כבר יודעים שהוא נכון רק אם ה-1 הוא מחרוזת טקסט.

כאשרם הםושים בתזכורת הזה על מבני נתונים, עולה 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 יתממש, אף על פי שבשני המשנים, foo ו-bar, יש את הערך 1. משנה אחד הוא מסווג מספר והאחר הוא מסווג 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 שלילן כן יתמשח, כיון שהאופרטור ההשוואתי `==` (להבדיל מ-`====`) מבצע את ההרמה של סוגים מיידע ואז מבצע את ההשוואה – שווי בין ערכים ולא בין סוגים מיידע. מדובר בפרקтика שනחשבת היום מאד לא מקובלת. נהוג להשווות גם בין סוגים מיידע ולא רק בין ערכים. זה נועד למנוע בלבול והתאמות כושלות של תנאים. השתמשו תמיד בהשוואה של סוג. אבל לעיתים אפשר למצוא קודמים שימושיים בהשוואה של ערכים בלבד. הטבלה הבאה מציגה את האופרטורים המשווים ערכים בלבד לעומתם אליהם גם את סוגים מיידע:

אופרטור השוואתי של ערכים בלבד	אופרטור השוואתי
<code>==</code>	<code>====</code>
<code>!=</code>	<code>!==</code>

איך בדיקות נעשה התרגומן הזה? בשביב זה צריך להכיר את המונחים **Truthy** ו-**Falsy** – מאמת ושקר. כאשר ממיררים משתנה לסוג מידע בוליאני, בודקים אם הוא **Truthy** או **Falsy**. כל סוג מידע וערך בג'אווהסקריפט נחשב ל-**Truthy**? מעת הערכים הבאים: `0, false, "", undefined, null, NaN`. כל מה שהוא **Falsy** מתרגם ל-`false`, כל מה שהוא **Truthy** מתרגם ל-`true`, בהרבה למשתנה בוליאני. וכך, כל מה שהוא **Truthy** מתרגם ל-`true`, כל מה שהוא **Falsy** מתרגם ל-`false`, בהרבה למשתנה בוליאני. וכך, כשמשתמשים בהרמות בוליאניות בהשוואה, כמו בדוגמה שלילן, נתקלים בבעיה.

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

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

למה זה ככה? כי מהירות ריקה מתרגם ל-`false` ו-0 גם הוא מתרגם ל-`false`. זכרם ששניהם **Falsy**? אם כך, שניהם **false** ואז נעשה ההשוואה, וזה עלול להיות בעיתוי. השבו על מערכת שמקבלת קלט מספרי ועושה אותו דברים. אם נעשה טעות ונשלח קלט ריק, ואז משתמשים בהשוואה רגילה, תהיה תקלה במערכת. היא תקבל קלט ריק ותחשב שהוא אפס, וההנתגנות בהתאם.

זו הסיבה שבגללה כדאי לעשות הכל ולהשתמש בהשוואה מדוקית של סוגים מיידע ולא של ערכים בלבד. תראו הרבה דוגמאות של אופרטור השוואה בין ערכים בראשת (לא בספר הזה); השתדרלו מאוד לכתוב נכון – מהניסיון שלו זה רק וועליכם.

תרגיל:

צרו משפט תנאי שמתראח אם המספר foo שווה ל-10.

פתרון::

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

הסבר:

המילה השמורה if בודקת את המשפט שיש בתוך הסוגרים העגולים. אם הוא נכון, החלק בתוך הסוגרים הממוסלים יירוץ. אם לא אז לא. בתוך הסוגרים יש אופרטור לוגיمسוג שווין שבודק אם המשתנה הוא 10. קטע הקוד זהה, 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 שבבודק את המשתנה foo. אם הוא שווה למחروف הטקסט "I am correct", הקונסולה תדפיס "He is correct". ואם לא, הקונסולה תדפיס "He is not correct" "correct"

פתרון::

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

הסבר:

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

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

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

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

יפעל.

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 השווה. אם המשתנה `day` הוא 1, כל קטע הקוד שבין הסוגרים המסלולים שבאים אחר כך יעבוז, והקונסולה תדפיס את השם `Sunday`. מיד אחר כך יש

if else if שבודק אם המספר הוא 2, וזו הקונסולה תדפיס את השם Monday, וכך הלאה. שימושו לב ל-if וולסוגרים שבתוכם יש את התנאי ולסוגרים המסמלים שפועלים אך ורק אם התנאי נכון. בסוף הקוד יש שמתקנים אם כל ה-if שהיו לפניו 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');
};
```

הסבר:

מגדירים משתנה BMI ובזדקים 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).

תרגילים:

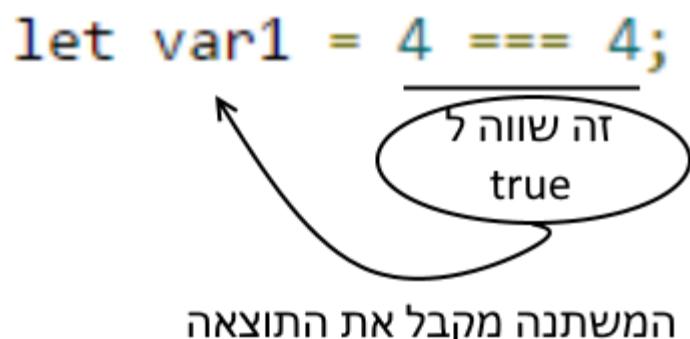
השתמשו בחמישה אופרטורים השוואתיים כדי להכניס true או false ל חמישה משתנים שונים. הדפיסו אותם בקונסולה.

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 שlidנו, האופרטורים ההשוואתיים מוחזרים ערך בוליאני – true או false. פשטוט יוצרים משתנה ומכניסים לתוכו את תוצאות פעולות האופרטורים הללו, 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, ל-Thursday או ל-5. אם כן יודפס בקונסולה . "Yay! Thursday!"

פתרונות:

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

הסבר:

כותבים את התנאים כרגע ומחברים ביניהם באמצעות האופרטור הלוגי "או", שכתוב `C||`. יש בעצם שרשרת של תנאים שבין כל אחד מהם נמצא האופרטור הלוגי `||`. כך בעצם כתובים "או" ג'אווה סקריפט.

תרגום:

כתבו משפט תנאי שבודק אם המשתנה `foo` שווה ל-5 וגם אם המשתנה `bar` שווה ל-`Thursday`. אם שני התנאים הללו מתקיימים ביחד, יודפס בקונסולה המשפט .`"Yay! 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 לחבר האופרטור הלוגי `&&` שמשמעותו "גם". יחד, משמעות הביטוי היא שהמשתנה `bar` יהיה שווה ל-`Thursday` וגם שהמשתנה `foo` יהיה שווה ל-5.

תרגום:

כתבו משפט תנאי שבודק אם `foo` הוא 0 או `null` ומדפיס 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 מתכנתיה היג'אווה-סקרייפט היו בוחרים לעשות. באמצעות האופרטור הלוגי ! מקבלים על true אם foo null, 0, מחרוזת טקסט ריקה או false. הפתרון השני הוא אינטואיטיבי יותר למי שעושה את צעדיו הראשונים בשפה ומכליל שני אופרטורים השוואתיים ואופרטור לוגי || שימושתו היא "או": foo === 0 או foo === null. שימושו לב ש-null היא מילה שמורה (לא מוקפת במירכאות). null הוא סוג מיוחד מוחדר שלמהנו עליו בפרקיהם הקודמים.

תרגילים:

כתבו משפט תנאי שמשווה בין:

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

ומצלה.

פתרונות:

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

הסבר:

נעשה שימוש באופרטור ההשוatoi == שמשווה בין הנתונים ולא בין סוגם המידע. אף על פי שהסוג המידע של 2 הוא מספר ושל "2" הוא מחרוזת טקסט, האופרטור ההשוatoi זה יחזיר true.

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 שגיאת. 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 כזה. switch case מאפשר לבצע השוואת המשתנה מסוימת לערך כלשהו (מספר, טקסט, ערך בוליאני וכו') ואז להריץ קוד בהתאם לערך ולספק גם פעולה ברירה מחדל. נשמע מוסף? 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 בדיקת כמו הקוד הקודם. פותחים בהצחה switch שהיא מילה שומרה. ב-switch שמים משתנה שיכול להכיל כל דבר ואחריו סוגרים מסוללים. בתוכם יש הצהרות case 1.case 1 למשל אומר שבמקרה שהערך הוא 1 – שימושו לב שמדובר מהמספר ולא במחרוזת טקסט – אם הערך עונה על התנאי זהה, כל הקוד שיש בין.defaultים ל-break מתקיים. אם שום תנאי לא מתקיים, כל מה שקרה אחרי ה-defaultים מתקיים עד ה-break. לא הייבים את ה-break האחרון, אבל הצבתי אותו בשביל הסדר הטוב. break היא מילה שומרה שמשתמשים בה כדי לשבור את ה-switch וליצאת ממנו.

חשוב לציין שה-default הוא חיוני תמיד, אפילו אם אתם חושבים שלא תגינו לשם. הקפדה על תמיד יכולה למנוע בעיות וצרות אחרות.

הבה נדגים זאת בדוגמה אחרת – קוד שמחזיר שם של חודש לפי מספר. עם if else היה קשה מאוד לעשות את זה, אבל עם 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 בקיז 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 מסוים ושם לו אותו בתוך ה-`switch`.`switch` ה-`switch` היא כאמור מילה שומרה שמודיעה למנוע של ג'אווה סקרייפט שישפה תנאי `switch` case. מיד אחרי ה-`switch` case המשתנה שנמצא בסוגרים העגולים יש סוגרים מסוולים, המכילים תנאים שנקרו `case`. כל `case` זה 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 כזה (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 כזה:

```
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 משתמשים במילה השמורה `break`, הקוד לא יעזור ואפשר להכנס כמה תנאים שרוצים. נשאלת השאלה, למה להשתמש בזו? אחרי הכול, בעזרה `if else` אפשר לכתוב את הקוד שלויל בלי ללמידה עמוק. זו טענה ולדית, אבל כאן נכנס הנימוק של קוד קרייא ואלגנטiy. המטרה שלכם כמתכנתים היא לכתוב קוד קרייא שקל לכל אחד (גם לכם) להבין. שימוש מרובה ב-`else if` הוא מבלבל וקשה לкриיא וلتיקון. ללא מעט מקרים, `switch case` יכול להקל את הקרייא ומשתמשים בו הרבה. לפיכך כדאי וציריך להכיר אותו ולהשתמש בו בכל פעם שיש יותר משלושה `if else`. זה לא כלל ברזול, אלא רק במקרה של תנאי לא מסובך מדי – אתם לא רוצים להתחיל להכנס משפטים תנאי והצבות בתוך ה-`switch case` שלכם.

תרגילים:

באמצעות `switch case`, כתבו קוד שלוקח 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;
}
```

הסבר:

כותבים את המילה השמורה `switch`, שמקבלת משתנה בסוגרים העגולים. המשתנה זה יבחן נגד ערכים שונים.
פורמט הבדיקה הוא כזה:

case value:

המילה השמורה `case` ומיד אחריה מגע הערך. אם הערך זה תואם את הערך שיש במשתנה שנמצא בסוגרים העגולים ליד ה-`switch`, הקוד שמופיע מיד אחרי הנקודותים ועד ה-`break` יירוץ. במקרה זה מדובר בהדפסה בקורסולה.

שימוש לב שימושי ערך דיפולטיבי שמודפס אם במשתנה יש ערך שלא צimenti במפורש במבנים. זה נעשה באמצעות התחן `.default`.

קבועים

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

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

כאן למשל מוגדר משתנה בשם myVar ומוצבת בתוכו מחרוזת טקסט ריקה. בהמשך מוצב בתוכו המספר 10. מעלים את המספר ב-1 באמצעות האופרטור `++` ולבסוף הופכים את המשתנה ל-`undefined`, סוג מידע של מדנו עליו קודם לנו.

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

```
const MY_CONSTANT = 10;
```

קונבנצייה אחת שבה משתמשים כדי להגדיר אותן היא אותיות גדולות עם קו תחתון שמחיד בין מילים, אבל ראוי לציין שימוש לא חובה לעשות את זה. אפשר להגדיר קבוע בדיקן כמו כל משתנה אחר. אם מנשים לשנות את הקבוע, מקבלים שגיאה בזמן הרצת הקוד.

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

נשאלת השאלה, למה להגביל את עצמנו ולהשתמש בקבועים? קבועים יעילים יותר מבחינה זיכרון מסיבות שלא אכנס אליהם כאן. ואם זה נשמע לכם אוטרי, אפשר לבוא ולומר ששימוש מושכל ב-`const` מעלה את ערככם בעניין כל מי שיקרא את הקוד שלכם. לפיכך כדאי וצורך להכיר אותו.

תרגיל:

הגדירו קבוע בשם `foo` והכניסו 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`, הכניסו לתוכו מספר, שנו את המספר ובחנו את התקלה שהקונסולה מזיגה.

פתרון:

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

הסבר:

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

בקרת זרימה – פונקציות

פונקציה היא סוג של רכיב בלאוּהסְקְרִיפְט שמחזיר תוצאה. אפשר להשתמש בפונקציה כאשר בנוין לבניית דברים מסוימים והוא החלק החשוב ביותר בשפה.

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

אבל? הנה נסתכל על הדוגמה הבאה:

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

ראשית הוגדרה פונקציה בשם `myFunc`. ההגדרה נעשית באמצעות המילה השמורה `function` ושם הפונקציה. במקרה הזה נבחר השם `myFunc`. אחרי השם יש סוגרים ריקים ואוז סוגרים מסולסים. בתוך הסוגרים המסולסים מופיעה הפונקציה עצמה, ייחודה אוטונומית של קוד שלא תרצו אם לא נקרה לפונקציה. במקרה הזה אין מיפוי-מה קוד, רק המילה השמורה `return` ו-1. ככלומר מי שיקרא לפונקציה קיבל 1 בתגובה.

איך מרכיבים פונקציה? ככה:

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

כאן מגדירים את הפונקציה וקוראים לה. את תוצאות הפונקציה מכניםים למשתנה. מה לפי דעתכם יקבל המשתנה? את מה שהפונקציה מחזירה. במקרה הזה 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 ההגדרה מתבצעת? באמצעות המילה השמורה `function` ואוז שם הפונקציה וסוגרים עגולים. הסוגרים המסולסים כוללים את הפונקציה. כרגע אין בה הרבה – היא מחזירה

באמצעות המילה השמורה `return` לכל מי שקורא לה את מחרוזת הטקסט "Hello World". אם קוראים לה ."Hello World" foo, הוא יקבל את מה שהפונקציה מחזירה, במקרה זה "Hello World" ומכניסים את התוצאות שלה למשתנה

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

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

מוכן שהפונקציה יכולה לעשות עוד דברים ולא רק להחזיר מחרוזת טקסט או מספר. הנה 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 המסלולים מתרחש האקסנון: מגדירים משתנים, עושים חיבורים ומקבלים תוצאה שאותה מחזירים עם ...`return`. כל הדבר הנדר הזה שהוא הפונקציה לא יעבד אם לא תקרו לו. קוראים לו באמצעות קריאה בשם הפונקציה והסוגרים העגולים והכנסת מה שהוא מחזירה לתוכה המשתנה `foo`. במקרה הזה הוא יקבל את מה שהפונקציה מחזירה, שהוא 30. ההגאה שמרתחשת בתוך הפונקציה – ככלומר הגדרת קבועים ופעולות מתמטיות – מתרחשת אך ורק בתוכה. היא לא זולגת החוצה. זה מה שיפה בפונקציה – אפשר לעשות מה שרצים בתוכה וזה לא יזלוג החוצה. בדוק כמו לאס וגאס – מה שקרה בפונקציה נשאר בפונקציה, והדבר היחיד שזולג ממנו 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 מגדירים שני קבועים: foo ו-bar, וכן את result שלתוכו מכנים את הסכימה של foo ו-bar. מוחזירים את result. כאמור, מה שקרה בפונקציה נשאר בפונקציה, והפונקציה לא תופעל אם לא תקרו לה ולא תכניסו את מה שהוא מוחזירה (כלומר מה שיש מיד אחרי ה-return) למשתנה כלשהו. אך חזיז ורעם! שם המשתנה הוא !foo! איך זה יכול להיות? הרי 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

```

יקפי שגיאה ולא ירוץ. מנוע הג'אושקריפט לא יכול להרים אותו כי אחד מכללי השפה הבסיסיים הוא שא-אפשר להגדיר משתנה וקבוע בעלי אותו שם. אז איך זה קרה פה? התשובה היא כאמור שם שנמצא בתוך הסוגרים המסולסים, ככלומר בתחום הפונקציה, מתקיים במוד אחר ולא זולג החוצה. המוד לאחר זה נקרא "לקסיקל סקופ" (Lexical Scope), ולפונקציה יש סקופ משלה, ככלומר מוד שבו אפשר להגדיר משתנים שהיו מבודדים לחלוטין מהסקופ של מי שקורא לפונקציה. נושא הסקופ י חוזר בהמשך כנושא בunosאים מתקדמים יותר.

```

function calculateMe() {
    const foo = 20;
    const bar = 30;           scope 1
    const result = foo + bar;
    return result;
}

let foo = calculateMe(); | scope 2
console.log(foo); // 50

```

אפשר כמובן לקרוא לפונקציה כמה פעמים שרוצים. גם הקוד הזה, למשל, יעבדו:

```
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 השתמשה בדיקן באמצעות קראתם לקבועים בתחום הפונקציה, והכל עובד. למה? כי מה שקרה בתחום הסkopf של הפונקציה לא רלוונטי ולא זולג לסקופים אחרים. הסkopf שבו עובדים קרוא נקרא "הסקופ הגלובלי" ואם קוראים לפונקציה מתוכו, דבר לא יזלוג החוצה אליו. בפונקציה אפשר לעשות כל מה שרוצים ולקרוא למשתנים כרצונכם. אמשיך לכתוב על סkopfs בהמשך. שימוש לב: בדוגמה קראתי לפונקציה מספר רב של פעמים, ובכל פעם הפונקציה רצה כאילו זו הפעם הראשונה. הפונקציה היא ייחודה אוטונומית ועצמאית ויכולת להיקרא מספר רב של פעמים. בכל פעם היא תחזיר למי קורא לה את מה שיש מיד אחרי ה-`return`.

ברגע שיש מיליד אחרי ה-`return`, הפונקציה מחזירה את הערך שנכתב. אבל מה קורה אם לא נכתב דבר? משחו בסגנון זהה:

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

מה שיקרה הוא שהפונקציה תחזיר `undefined`. זכרם אותו? למדנו עליו בפרק על סוגי מידע פרימיטיביים נוספים. זהו מבנה נתונים בסיסי והוא חוזר כאשר כתבים סתם `return` או כאשר אין `return` כלל. למשל פה:

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

אם יש כמה return, ה-first return הוא שקובע והקוד שמתבצע לאחריו לא יירוץ 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 יירוץ לעולם.
シימו לב: אפשר להגיד כמה פונקציות שרוצים ולקראות להן איך שרוצים. בכל הדוגמאות הוגדרה פונקציה אחת, אבל אין שום בעיה להגיד כמה פונקציות שרוצים.

פונקציה עם ארגומנטים

אפשר להעביר לפונקציה ערכים מבוחן 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. המשנה זהה מוגדר בתוך הפונקציה כאילו הגדרתי אותו עם let. אפשר להכפיל אותו או לחתה אותו ולהציג אותו איפה שרוצים, לבדוק כמו משתנה רגיל. השם המקובל למשתנה כזה הוא ארגומנט.
אין 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 שמכריזים עליו בסוגרים אפשר להשתמש בו בפונקציה עצמה. במקרה הזה לוקחים אותו, מכפילים אותו ומכניסים את התוצאה למשנה answer.

אין מכניסים אותו בקריאה? פשוט מאד, מעבירים את הארגומנט בסוגרים. מה שמעבירים נחשב ל-arg1.

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

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

הfonקציה היא אותה פונקציה, אבל התוצאה שלה משתנה בהתאם לארגומנט שמעבירים. כש庫ראים ל-fonקציה כך:

```
multiply(10);
```

או arg1 בתחום הפונקציה מקבל את הערך 10 וההתשובה המוחזרת היא 20, arg1 כפול 2 שנכנס ל-answer ומוחזר עם return.

כש庫ראים ל-fонקציה כך:

```
multiply(20);
```

או arg1 בתחום הפונקציה מקבל את הערך 20 וההתשובה המוחזרת היא 40.

כלומר, הפונקציה נותרת כפי שהיא, רק הקריאה שלה משתנה.

שימוש לב: בדרך כלל משתמשים בפונקציות כאשר כתבים ג'אוועסקרייפט כי זו הדרך הטובה ביותר לפרק את הקוד לחלקים קטנים ולהימנע מוחזרות על קוד. חזרות על קוד הן דבר שモובן להימנע ממנה בקוד כיון שאם רוצים לשנות אותו, נדרש עבודה רבה.

אפשר להשתמש בכמה ארגומנטים שרוצים. כך למשל נראה 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
```

מכניסים את כל הארגומנטים שרוצים בשמות 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 בתחום הסוגרים המסלולים של הפונקציה. מה שהשוו הוא שהfonקציה תעשה return. כשתקראו לפונקציה, תכניסו שלושה ארגומנטים בקריאה:

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

כל ארגומנט יופרד בפסיק.

ארגומנטים 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 פונקציה multiply שמקבלת ארגומנט (הוגדר בפונקציה כ-`arg1`), אבל בקריאה של הפונקציה:

```
let foo = mutliply();
```

לא הועבר שום ארגומנט. איזה ערך יקבל `arg1`? התשובה היא `undefined`, וזה הקוד שליל יקיים, כי אין אפשרות להכפיל `undefined` ב-2. התוצאה תהיה `NaN` (כאמור, ראשית התיבות של `NaN` – Not a Number – לא מספר). אפשר להכנס ערך בירית מחדל (דיפולטיבי) לארגומנטים שייכנס אליהם אם מי שקרה לפונקציה לא העביר לה ארגומנטים. עושים זאת באופן פשוט למדי – כך:

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

אם בקריאה מעבירים ערך, כל ערך, `arg1` יקבל אותו. אבל אם לא מעבירים ערך, `arg1` יקבל במקרה הזה את הערך 1 ואז התשובה תהיה 2. יש לזכור שהשם של ערך נעשית על ידי הסמן `=`. אם לפונקציה יש כמה ארגומנטים, בלאו הסקריפט אפשר להכניס לכלם ערך בירית מחדל. הסתכלו על הדוגמה הזו:

```
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 בירית מחדל של המספר 0. אם לא מעבירים ארגומנט כלשהו, הוא מקבל 0 (ואז המכפלה של שלושת הארגומנטים תהיה 0).

שים לב לדוגמה الأخيرة: הארגומנט הראשון שמעבירים הוא `undefined`, שהוא מילה שמורה שהוזכרה בפרק על סוג מידע. אני בעצם מצהיר שהารגומנט הראשון `arg1` הוא `undefined` ומפעיל את בירית המחדל. שימו לב שמדובר ב-`undefined` אמיתי ולא בערך ריק כמו מחוץ ריקה או אפילו `null`.

שימו לב: זה נחשב לנוהג רע להכניס יותר מהמשה ארגומנטים לפונקציה. אם בפונקציה שלכם יש יותר מהמשה ארגומנטים סימן שצורך 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 בדוק אותו דבר כמו להגיד פונקציה בדרך שלמדנו. שימו לב שהגדירת הפונקציה היא כמו הגדרת סוג מידע פרימיטיבי מסווג מחרוזת טקסט, מספר או `Symbol`. אם תעשו `typeof` למשתנה 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'אווחסקריפט היא מבנה נתונים חדש. בנויגוד לערכים הפרימיטיביים שהכרתם עד כה, פונקציה היא ערך לא פרימיטיבי ומורכב יותר. צריך לזכור שפונקציה בסופה של דבר היא מבנה נתונים שהיבאים להכניס למשתנה בבדיקה כמו מספר, טקסט, ערך בוליאני וחבריהם. מסיבות היסטוריות, אפשר להגיד פונקציה בשתי הדרכים, 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 שנקרו – Hoisting – העמלה.

Hoisting

כאשר מגדירים פונקציות, המנווע של ג'אווהסקריפט מזיז את כוון למעלה. לשם הדוגמה, מה לפי דעתכם תהיה תוצאה הקוד זהה?

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

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

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

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

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

יקבל שגיאה, כי כאמור קוראים לפונקציה לפני שמוגדרים אותה. יש להמשמעויות כאשר עובדים בקוד מתקדם יותר ולא בהתחלה, כמובן.

closure

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

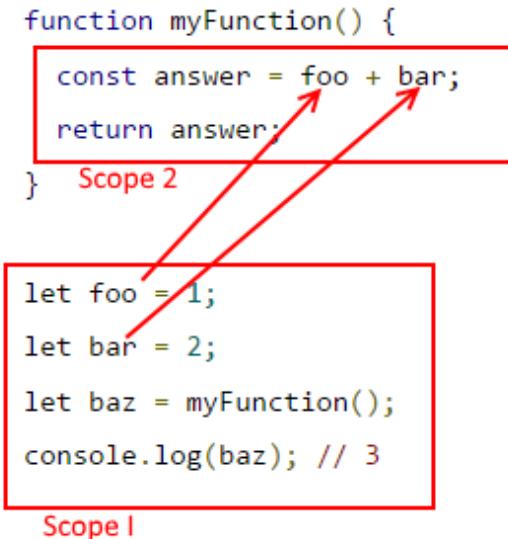
```
function myFunction() {
  console.log(foo);
}
let foo = 'Hello';
myFunction(); //Hello
```

מה קורה פה? הפונקציה myFunction היא פשוטה למדי ומדפסה משתנה בשם foo. אבל שימוש לב-`sh-foo` מוגדר מחוץ לפונקציה! ועדיין, כשמכניסים 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 מחזיקה לה. כלומר, foo ו-*bar* אינם מוגדרים בסcopו של הפונקציה אלא בסcop הגלובלי ועודאי שלא מועברים לפונקציה כארגומנטים, וזאת הפונקציה מכירה אותם. הדיאגרמה זו אמורה להבהיר את העניין:



התוכנה הזו נראית בגלאוּסְקָרִיפֶט closure. המשמעות שלה היא שימושים שהוגדרו בסcop האב (אם קיימ) של הפונקציה יהיו מוכרים בסcop של הפונקציה (סקופ הבן). זו תוכנה חשובה מאוד בגלאוּסְקָרִיפֶט והוא מאפשרת גמישות, אך היא גם מסוכנת מאוד.
מובן שגם דורסים את המשתנים בסcop של הפונקציה, המשתנים בסcop 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 הסcop של הפונקציה דורסים את foo. foo שמוגדר מחוץ לפונקציה כਮובן לא נפגע, אבל מה שמתתרחש בתחום הפונקציה הוא ש-foo הופך לעצמאי לחלוטין.
MOVEDIN ש-closure הוא דו-כווני ואפשר לשפע על משתנים בסcop האב דרך סcop הבן. למשל:

```

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

```

שימוש לב שבתוק הפונקציה לא הוגדר משתנה `foo` באמצעות `let`. מהרגע שעושים את זה מושנים כਮובן את המשתנה שנמצא בסcopה האב.

כלומר, כל משתנה שモוגדר מבחוון חשופ לפונקציה הפנימית. זה כדי חזק שעלול ליצור כאום משמעותי. בגרסאות קודומות של ג'אווהסקריפט (ES5 ומטה) הכאום היה גדול אף יותר כיון שהגדרת המשתנה, שנעשתה באמצעות `var`, אפשרה ל-`var` להיכנס תמיד לסקופ הגלובלי. ה-`let` מוגבל יותר.

שימוש לב: נראה זה נראה תיאורתי למדי ואפיילו מטופש, אבל יש חשיבות עליונה להבנת ה-closure. בלולאות ובפונקציות רקורסיביות, ובטה ובטה באפליקציות מורכבות יותר, יש היישנות הרבה על התוכונה זו. לפיכך כדאי להש퀴ע זמן בלימוד שלה ועוד אהזור 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);
})

```

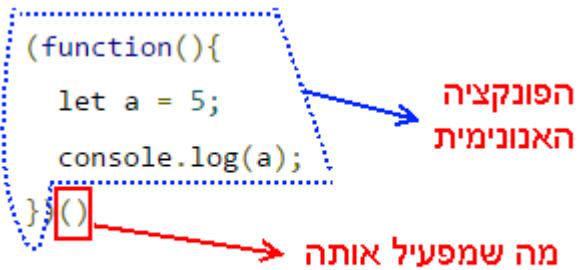
שימוש לב שיש כאן הגדרת פונקציה ללא שם, באופן די דומה למה שהכרתם: המילה השמורה `function`, סוגרים פותחים וסגורים והסוגרים המסורלים שבתוכם הפונקציה עצמה. בדוגמה זו הפונקציה מגדרה משתנה `a` ומדפסה אותו בקונסולה. אבל הדבר הזה לא ירוץ כי אף אחד לא קורא לפונקציה. איך קוראים לה? מוסיפים סוגרים פותחים וסגורים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

```

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

```

אם לא תתעצלו ותריצו את הקוד ש לעיל, תראו שהוא אכן רץ. למה? כי הפעלתם את הפונקציה האונומית מיד לאחר היצירה שלה.



פונקציות אנונימיות, ככלומר كانوا ללא שם, משמשות בהמוני מקרים ומקומות בג'אוועסקריפט, בין שMRIיצים אותן מיד לאחר הפעלה ובין שלא.

בג'אוועסקריפט מודרנית, אפשר לכתוב פונקציה אנונימית ללא שימוש במילה השמורה `function` אלא באמצעות פונקציית חץ או, באנגלית, Lambda:

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

זה בדוק אותו דבר, אבל מבחינת הסקוֹפּ יש לפונקציות חץ כמה יתרונות גדולים וכדי להשתמש בהן. נשאלת השאלה, למה להשתמש בכלל בפונקציה אנונימית? ויש לה כמה תשובה.

פונקציה אנונימית כמשתנה

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

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

מה שתרחשפה הוא שיש שתי פונקציות. פונקציית `returnSomething` מחזירה 100. פונקציית `runMe` מקבלת משתנה. מה היא עשויה בו? לocket אתו ו"מרים" אותו באמצעות סוגרים פותחים וסוגרים. ככלומר היא מניחה שהארוגמנט הוא פונקציה. במקרה הזה הוגדרה פונקציה בשם `returnSomething` והועברה כמשתנה לפונקציית `runMe`. אבל אפשר להעביר לפונקציית `runMe` גם פונקציית חץ אנונימית:

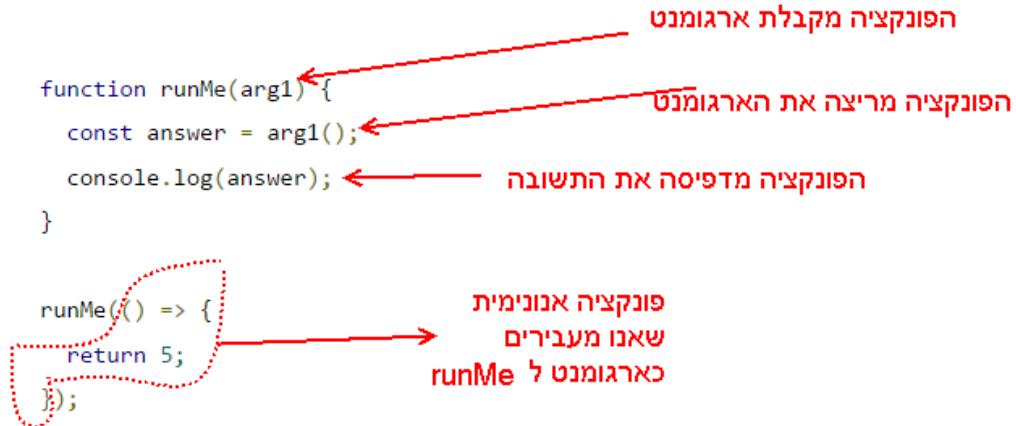
```
function runMe(arg1) {
  const answer = arg1();
```

```

        console.log(answer);
    }
runMe(() => {
    return 5;
});

```

ואם תרצוו את זה תראו הדפסה של מה שהפונקציה האנונימית מזוררת.



זה נראה מעט אבסטרקטי ואף מיותר, אבל משתמשים בזה בהמון מקומות בג'אוהסקריפט. המקום הראשון שבו רואים את השימוש בפונקציות אונונימיות הוא בלולאות, אבל לא רק שם. יש לא מעט מקומות שבהם מעבירים פונקציה כארוגמנט. הפונקציה שעוברת כארוגמנט נקראת "콜백" ובאנגלית callback. השם הזה ניתן לה כי היא "נקראת בחזרה" על ידי הפונקציה ברגע המתאים. זכרו את השם "콜בק", עוד אשוב אליו.

אפשר להעביר גם ארגומנטים ל콜בק. שימו לב לדוגמה זו:

```

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

```

היא זהה כמעט לחלוטין לדוגמה הקודמת, אבל במקרה זה מצפים שבפונקציה האונונימית שמעבירים יהיה ארגומנט אחד. במקרה זהה מעבירים 2 לארוגמנט. אם העניין אבסטרקטי מדי זה בסדר, רק זכרו שקולבקים יכולים לקבל ארגומנטים שהפונקציה הקוראת להם מעבירה. במקרה הזה פונקציית runMe, שאליה מעבירים ארגומנט שהוא פונקציה (או קולבק), מעבירה 2 כארוגמנט ל科尔בק.

פונקציה אונומית שմבודדת מהסקופ הגלובלי

פונקציה אונומית שמריצה את עצמה היא נחרת בכל מה שקשרו לביוד מסקופ הגלובלי ולמן אפשרות למוכנה להגדיר שכבה שרק דרך אפשר להגיד הקוד שהוא כתוב. זה מעט מתקדם מדי בכך לעכשו, אבל אנסה להסביר בכל מקרה. שימו לב לפונקציה 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
```

הפונקציה האונומית שמריצה את עצמה מקבלת ארגומנט בשם jQuery. כל מה שמתרחש בתוך הפונקציה זו נשאר חסוי מסוקפים אחרים. אם מגדירים foo (לא משנה אם בעזרת let או var) אז הוא יישאר חסוי. אפשר לבחור לחושף את bar אם מזמנים אותו לאובייקט jQuery. מן הסתם, אם רוצים להשתמש באובייקט jQuery על מנת לתקשר עם הסביבה החיצונית, הסביבה החיצונית תצטרך ליצור אותו.

כאמור, אם זה נראה תלוש מעט זה בסדר גמור. בשלב הזה של הלימוד טוב לזכור שפונקציה אונומית שמריצה את עצמה לגיטימית לשימוש בספריות ג'אויסקריפט שונות (כמו ספריית jQuery) ובכל מודול שהוא, שבו רוצים למש скופ נפרד ופרטי. לא נדריר לראות ברשות דוגמאות של פונקציות אונומיות שמריצות את עצמן.

תרגיל:

צרו פונקציה בשם myFunc שמחזירה null למשתנה. הדפיסו את המשתנה בקונסולה.

פתרון:

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

הסבר:

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

תרגילים:

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

פתרונות:

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

הסבר:

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

הפונקציה לא תתקיים ולא תרצו אם לא תקראו לה. את זה עושים באמצעות:

```
let foo = myFunc();
```

כאן יש קראה לפונקציה והכנסה של מה שהיא מחזירה ל-foo. הדפסה של foo תראה את זה.

תרגילים:

צרו פונקציה בשם yay שמחזירה את מחרוזת הטקסט "yay". הכניסו את מה שהיא מחזירה למשתנה foo והדפיסו את המשתנה בקונסולה.

פתרונות:

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

הסבר:

יצרים פונקציה בשם yay עם המילה השמורה yay. מיד אחרי ההגדלה יש סוגרים מסוללים, שבהם מתרחש כל מה שקורה בפונקציה. במקרה זה, דבר לא מתרחש. הפונקציה מחזירה את מחרוזת הטקסט "yay" באמצעות המילה השמורה return.

מה שקורא לפונקציה – כי ללא הקראה היא לא תופעל – הוא הקוד הבא:

```
let foo = myFunc();
```

הקוד הזה עושה שני דברים - קורא לפונקציה ומעביר את מה שהיא מחזירה למשנה foo. המשתנה foo יודפס וכייל את מה שיש בו, כלומר מה שהפונקציה החזירה.

תרגיל:

כתבו פונקציה שמקבלת ארגומנט, מדפיסה אותו בקונסולה ואז מוסיפה לו 1 ומהזירה אותו. קראו לפונקציה עם ארגומנט 1 והכניסו את התוצאה שלה למשנה. הדפיסו אותו בקונסולה.

פתרון:

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

הסבר:

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

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

כלומר, מדובר פונקציה ובתוכה הסוגרים העגולים מכנים ארגומנט. Unless ציריך להוסיף את ההדפסה בקונסולה:

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

הารוגמנט הוא משתנה לכל דבר והוא חי בתחום הפונקציה ברגע שקוראים לה. אפשר להדפיס אותו בקונסולה, כמובן, כמו כל משתנה אחר. Unless ציריך להוסיף לו 1 ולהחזיר את התוצאה. את זה כבר הראתם:

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

אחרי שכותבים את הפונקציה, צריך לקרוא לה. את הקראיה מבצעים כך:

```
let foo = addMe(1);
```

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

עבירים כารוגומנט את המספר 1 לפונקציה ומכניסים את מה שהיא מוחזירה לתוך המשתנה foo, שהוא מדפיס.

תרגילים:

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 קריאות לפונקציה - עם מספר חיובי, עם מספר שלילי ועם 0 - כדי לראות שהכל עובד.

פתרונות:

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

הסבר:

כותבים את הפונקציה כמו כל פונקציה אחרת עם ארגומנט שהוא מקבלת. בחרתי את השם number עבור הארגומנט. הארגומנט הזה הוא משתנה לכל דבר בתוך הסkopf של הפונקציה, ואפשר לכתוב עליו משפטים תנאי כי שלמדנו בפרק הקודמים. שימו לב שלפונקציה זו אין return כיון שהיא לא מוחזירה ערך אלא מדפסה בקונסולה בלבד. לפיכך גם לא צריך להכניס את מה שהיא מוחזירה לתוך משתנה אלא לבצע קריאה בלבד שפעולתה אותה.

תרגילים:

כתבו פונקציה שמקבלת מספר ובודקת אם הוא מספר או סוג מידע אחר. אם הוא סוג מידע אחר היא תדפיס בקונסולה a/n ותסייעים את פעולתה. 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 %, אופרטטור הבודק חילוק לפי שארית. למדנו על האופרטורים הללו בפרק 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 חי וקיים בפונקציה כמו כל משתנה.ומו הוא `number`. ראשית בודקים אם סוג המידע שלו שונה ממספר באמצעות האופרטור `typeof` שהוזיר את סוג המידע הקיים במשתנה. אם הוא לא מספר מדפיסים בקונסולה `a/n` ומבצעים `return`. ה-`return`, כפי שלמדנו, יסימן את פועלות הפונקציה סופית, אבל הוא יופעל, כמובן, רק אם התנאי יתממש. בהנחה שההתנאי לא התממש, ניגשים לבדוק הוויגי/אי-זוגי ואת זה עושים באמצעות האופרטור שארית, שסימנו `%`. האופרטור הזה, להזכירכם, מחזיר את השארית של החלוקה. אם מספר מחלק ב-2 ללא שארית סימן שהוא זוגי. אם יש שארית סימן שהוא אי-זוגי. כותבים את התנאי ומהזירים ערך בוליאני באמצעות המיללים `true` או `false`. כל מה שנותר לעשות הוא לבדוק את הפונקציה עם כמה ארגומנטים שונים.

תרגילים:

צרו פונקציה שמקבלת ארגומנט. הארגומנט אמרור להיות פונקציה. אם הוא לא פונקציה מופעלת שגיאה. אם הוא אכן פונקציה מריצים אותו ומדפיסים את התוצאה. בדקו פעמיים – עם פונקציה אונונית שמחזירה מחרוזת טקסט ועם פונקציה אונונית 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'; }); // prints hello;  
callMe(() => { return 5; }); // prints 5
```

הסבר:

פונקציית callMe מקבלת את arg1. בודקים אותו באמצעות האופרטור `typeof`. אם הוא לא פונקציה, מדפיסים הודעה שגיאה ומחזירים את הפונקציה כריקה. אם arg1 הוא פונקציה, מתייחסים אליו כפונקציה וקוראים לה. את מה שהיא מ>Returns מדפיסים.

עד כאן זה פשוט; הבדיקה מעט יותר מסובכת. על מנת לבדוק צריך ליצר פונקציה שמחזירה משהו. את זה עושים באמצעות פונקציית `fn` אונומית. הפונקציה האונומית הראשונה מחזירה מחרוזת טקסט והפונקציה האונומית השנייה מחזירה מספר.

אובייקטים

כבר למדנו על סוגי מידע פרימיטיביים בג'אווהסקריפט – מהרוות טקסט, מספרים, null, undefined, Symbol ובוליאני. בפרק הקודם למדנו על פונקציה והסברתי שמדובר בסוג מידע שאינו פרימיטיבי (כלומר סוג מידע מורכב) מסוג function. אם יוצרים פונקציה ובודקים את הסוג שלה באמצעות typeof מגלים שהוא המידע שלו הוא function.

סוג מידע נוסף פרימיטיבי הוא אובייקט, ומדובר בסוג המידע החשוב ביותר בג'אווהסקריפט. המטרה של אובייקט היא ליצור סוג מידע מורכב שיכול להכיל סוג מידע אחרים, פרימיטיבים או אפילו אובייקטים ומערכות (עליהם לדבר בפרק הבא). איך יוצרים אובייקט? הזורך הפשטה ביותר היא זו:

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

אם מרצוים_typeof על המשתנה שמכיל את האובייקט, מגלים שהוא סוג object.

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

אובייקט יכול להכיל בתוכו מידע לפי "מפתחות". מפתחות הם דרך להכנס מידע נוסף תחת שם מזוהה לאובייקט – כך שהיא לנו קל לראות את המידע זהה ולשייך 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 הזה id) והכנסת ערך, בדיק כmo משתנה. שמו לב לנקודה בין myObject ל-id. זהו אחד המבנים הבסיסיים של השפה.

אפשר להכנס עוד מפתחות, כמוון. למשל מהרוות טקסט:

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

גיישה למפתח נעשית באמצעות שם המשתנה שמכיל את האובייקט, נקודה ואו 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 אותם. בדיק-בדיק כמו משתנים! 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 שכשמדפיסים את האובייקט, המפתחות והערכים מסודרים באופן זהה:

```
key1: value1,
key2: value2,
```

וכן להלאה. 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 ליצירת האובייקט – ככלומר יצרה שלו כבר עם כל המפתחות מראש – מקובלת יותר וגם עדיפה מבחינת ביצועים.

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

אובייקט בג'אוהסקריפט הוא גמיש. מובן שאפשר להכניס לתוכו ערכים עם משתנים, למשל במקרה 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 מ משתנים שבמקרה זהם להלוטין לשמות המפתחות. נכוון `name: name` עלול לבלבול – אבל הראשון הוא המפתח והשני הוא הערך או, נכון יותר, שם הערך שהמפתח נכנס אליו. חשוב לציין

שומרגע ההצעה של המשתנה בפתח המתאים באובייקט, הקשר בין המשתנה לערך נעלם. המשתנה יכול להשתנות, אבל הערך יישאר כפי שהיא ברגע ההצעה. זה קורה כי למעשה נוצר אובייקט בזיכרון שמכיל את הערך ולא הפניה למשתנה שנמצא במקום אחר בזיכרון. הנה 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 – מתכוונים אוחבים מאוד להשווות את שמות הארגומנטים לשמות המפתחות. משחו בסגנון הזה:

```
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'אוּהסְקְרִיפְט מנוסים, אתם לא אמורים להיבהל מהקוד הזה – יש פונקציה שמקבלת שני ארגומנטים, מכניסה אותם לאובייקט ומחזירה אותו. זה משווה שאתם אמורים להכיר אחרי הפרק הקודם על פונקציות. את הקוד הזה תראו חוזר כמה וכמה פעמים במערכות מבוססות ג'אוּהסְקְרִיפְט, עד כדי כך שהכנסו לג'אוּהסְקְרִיפְט את היכולת לייצר מפתחות וערכיהם באופן אוטומטי לפי שמות הערכים. כך הפונקציה שלעיל תהיה שוקלה לו:

```
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 וטופסת תאוצה בשנים האחרונות. לא משתמש בה בדוגמאות, אבל אני זכרו שהוא קיימת כי נעשה בה שימוש רב, בעיקר על ידי מתכוונים מנוסים (כולל עבדכם הנאמן).

כאמור, אובייקט יכול להכיל כל נתון שהוא בתור ערך, כולל... אובייקטים אחרים! שימו לב לדוגמה 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, לחזו על השורה הזו והאובייקט המלא יוצג לפניכם.

זאת דוגמה כמעט אמיתית – יש אובייקט user ואובייקט מידע מורחב. מכניסים את האובייקט של המידע המורחב אל מפתח שנקרא moreData באובייקט user. כשהמצאים הדפסה של user בקונסולה אפשר לראות את האובייקט moreData מופיע במלואו תחת המפתח .moreDataprofileExtendedData

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 בוזה בעיקר כאשר שם המפתח נמצא בתחום משתנה מסווג מהרוות 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 כמחרוזת טקסט. על מנת לקרוא למפתח מתוך האובייקט משתמשים בסוגרים מרובעים, ובתוכם שמים את המשתנה שמכיל את שם המפתח.

אם תנסה לעשות משהו כזה (הוֹא המשתנה שמכיל את שם המפתח) תקבלו undefined.

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

לפייך לעולם לא משתמשים בכך זה 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, כמו כן, בכך או בסוגרים מרובעים כדי להסביר לאופרטור delete איזה מפתח הוא צריך למחוק.

הכנסת פונקציה כערך

כפי שציינתי קודם, כל סוג מידע יכול להיכנס לאובייקט כערך. הדגמתי זאת באמצעות סוג מידע פרימיטיבים, כמו מספר וטקסט, וגם הריאתי שאפשר להכניס אובייקט כערך לאובייקט אחר. מוכן שגם פונקציה יכולה להיכנס 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 במפתח alertMe ומוכנים לתוכו פונקציה שכל מה שהוא עושה הוא להדפיס בקונסולה את מחרוזת הטקסט "hi!". יצירת הפונקציה זהה במאה אחותו ליצור פונקציה רגילה והכנסתה לתוך משתנה. הקראיה לפונקציה בתוך אובייקט גם כן זהה לחלווטין לקריאה לפונקציה ונעשית כפי שקוראים לכל ערך אחר.

לפונקציה בתחום אובייקט יש חשיבות גדולה מאוד ואדון בה בהמשך. כרגע, מה שהחשוב הוא שתדעו שככל מבנה מידע יכול להיכנס לאובייקט, בין שמדובר בסוגי מידע פרימיטיביים ובין שבסוגי מידע מורכבים יותר.

אפשר להכניס פונקציה גם כתיב מקוצר (פונקציית חצ'ר, שעליה למדנו בפרקם הקודמים) באופן הבא:

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

אובייקט קבוע

אחד הדברים החשובים שצריך לזכור הוא שם מגדירים אובייקט קבוע, אין שום בעיה לשנות אותו. נכון, אין אפשרות לשנות את הסוג שלו (זאת אומרת, קבוע שמעכשו במקום אובייקט יכול קבוע מס'ר), אבל אפשר להוסיף לו תכונות או לשנות תכונות קיימות, ככלمر לשנות את השדות בתחום האובייקט ולא לעשות השמה לאובייקט אחר. מדובר בהבדל ממשמעותי:

```
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 myObject קבוע, אך למרות זאת אפשר לשנות את הערכים בתחוםו. אבל אם מנסים לשנות את הסוג אובייקט למשהו אחר (במקרה זה מס'ר), נתקלים בשגיאה.

יש כמה הבדלים משמעותיים בין סוגי מידע פרימיטיביים לבין אובייקטים וסוגי מידע מורכבים יותר. אחד השינויים הכח מבלבלים הוא עניין ההצבעה, שאינו כל כך מורכב כפי שהוא. כשכתבתי על משתנים פרימיטיביים, ציינתי שאפשר להעתיק את המשתנה למשתנה אחר. אם משנים את המשתנה الآخر, המשתנה המקורי נשאר כשהוא. הנה דוגמה:

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

אבל מה קורה כשהעושים משהו דומה באובייקט? אם מעתיקים אותו למשתנה אחר? אם עושים את זה ומשנים את המשתנה الآخر, רואים שהאובייקט העיקרי השתנה! נטו להרים 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, משפיע על האובייקט המקורי! זה קורה כי כسمעתיקים אובייקט שנמצא במשתנה A למשתנה object לא מבצעים העתקה אלא הפניה, וההבדל ביןיהםמשמעותי. הסיבה לכך היא שיקולי זיכרון. אם רוצים לבצע העתקה של אובייקט ולשנות אותו בלי שהוא ישפיע על האובייקט המקורי, צריך לעשות הליך שנקרא clone ואפשרי לביצוע באמצעות לולאות, ועל כך בפרקם הבאים.

תרגילים:

צרו אובייקט של computer שיש לו id, modelName ו-price

פתרונות:

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

הסבר:

יווצרים משתנה ומוכניסים 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 ואו הערך. 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}
```

בצם יוצרים כאן אובייקט ריק ואז מוכניסים 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,  
}
```

פתרונות:

```
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 שקווראים לו, `dayNumber`, והמטרה היא למצוא את מסpter היום. לצורך כך משתמשים במשפט switch case שלמדנו. 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 למשנה `dayName`.
את `dayName` ואת `dayNumber` מוכניסים לאובייקט ומחזירים אותו. זה הכלול.
כיוון שהשמות המפתחות זרים לשמות הערכים, אפשר ליצור את האובייקט באופן מוקוצר:

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

תרגילים:

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

פתרונות:

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

הסבר:

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

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

אלא צריך להשתמש בסוגרים מרובעים להגדירה. כל שנותר לעשות אחרי הוספה התוכנה והערך הוא להחזיר את האובייקט ולבדוק. זה הכלול.

תרגיל:

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

פתרון:

```
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.

מערכות

מערכות הם סוג מידע מורכב המשמש לאחסון מידע. הם דומים למדדי אובייקטים, אך המפתחות שלהם הם מספרים מ-0 עד אינסוף. אל המערכת אפשר להכנס כל סוג מידע שהוא. מקובל להתייחס אל הערכים שנמצאים במערך כאל איברים; האיבר הראשון נמצא במיקום 0, האיבר השני נמצא במיקום 1 וכן הלאה. זה דבר ש תמיד מבלב מתכנתים. כשאדבר על לוളאות תבינו את ההגיון של התחיל מ-0, אבל חשוב לשנן ולזכור שבכל הנוגע למערכות תמיד סופרים מ-0.

יצירת המערכת נעשית כך:

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

כאן יוצרתי מערך ריק. על מנת להכנס לתוך המערכת 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 הייצירה 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 ובבר בשלב הייצירה מכניסים לתוכו שלושה איברים לפי הסדר - למיקום 0, למיקום 1 ולמיקום 2. אחרי הייצירה מדפיסים בקונסולה את האיבר שבמקום הראשון (מיקום 0), משנים אותו ומדפיסים אותו שוב. שימושו לב: פניה לקבלת ערך מתוך מערך נעשית על ידי סוגרים מרובעים וגם האינדקס של האיבר.

אפשר כמובן להכנס גם אובייקטים לתוך מערכים וזה 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, ואו שולפים את האובייקט הראשון מהמערך ואיפלו לא את כל האובייקט הראשון, אלא מפתח נבחר מהאובייקט הראשון.

וכן, בדיקת כמי שמערך יכול להכיל כל סוג מידע שהוא, כולל אובייקטים, מערך יכול להכיל גם מערכים. לערך שמכיל מערכים נוספים קוראים מערך דו-ממדי. הינה דוגמה למערך כזה:

```
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 כדורים, כך יש מחסנית של נתונים. הכנסה של איבר חדש נקראת "דחיפה", ואפשר למשמש אותה באמצעות שימוש בפונקציה מיוחדת לסוג המידע מערך בלבד, שנקראת "push":

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

push היא פונקציה מיוחדת אך ורק על סוג מידע שהם מערכים, כלומר כולן שמחזירים array באמצעות האופרטור של typeof. ככלמנו על סוג מידע, הריאתי כל מיני פעולה שאפשר לעשות על מהרווזת טקסט או על מספרים; ובכן, push היא פעולה שאפשר לעשות אך ורק על מערך. מכניםים לתוכה ארגומנט את המידע שרוצים להכניס למערך בתווך האיבר החדש והאחרון – בין שמדובר במספר ובין שבמהרווזת טקסט, באובייקט, בערך ועוד. כל דבר יוכל כניסה ארגומנט push-וּוּוּץ איבר חדש בסוף.

הדחיפה מוסיפה איבר חדש למערך על גבי האיברים האחרים. אם היה איבר אחד במיקום 0, דחיפה תוסיף איבר חדש למיקום 1.

משיכת היא בדיקת הפך מדחיפה – היא מאפשרת למשוך את האיבר האחרון מהמערך ולקבל אותו כמשתנה. המשיכת נעשית באמצעות פקודה pop, שגם 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, והערך שעבר שינוי (מוותץ), או נכון יותר המחסנית, מתקצר באיבר אחד.

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

```
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 מכניסים את מהירות הטקסט z לתוך המערך באמצעות unshift, אז האיבר הראשון כבר לא יהיה a אלא z. האיבר האחרון עדין נותר c.

ולבסוף, שליפת 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 מהירות טקסט a והאיבר האחרון הוא מהירות טקסט c. שימוש בפונקציה shift מוחזרת האיבר הראשון. כרגע במערך האיבר הראשון הוא b והאיבר האחרון הוא עדין c. המערך קוצר 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 לסופו המערך	push

תמונה נוספת שקיימת וייחודה 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 במשהו טריקי. אם למשל יוצרים מערך ומכניסים אליו שני ערכים בלבד – אחד במקומות 0-0 (שהוא המקום הראשון, והוא זוכרים שהמקום הראשון במערך הוא 0) והשני במקומות 99-99 – 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'אוּהַסְקְּרִיפְט תיצור את כל האיברים האחרים בערך עד המיקום שהוכנס. כל איבר כזה יהיה מסוג `undefined` (שםו לב – לא `null` אלא `undefined`). מצד שני, ככה תמיד אפשר לקבל את הערך האחרון בערך, 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 במיקום 0. כמו באובייקט, מהיקת איבר בערך התבצע בעזרת האופרטור `delete` שבצム לוקח את האיבר ומכניס אליו `:undefined`:

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

שםו לב שכשרוצים למחוק את הערך השלישי, צריך לציין [2] כיון שהערך מתחילה תמיד מ-0. הבעיה בכך הוא היא שהערך נשאר בה גודל קבוע, והאיבר שעושים לו `delete` הופך ל-`undefined`. כדי למחוק איבר בערך ולקצר אותו יש להשתמש ב-`splice`. כמו המתודות הקודמות שלמדו לנו עליון, גם המתודה `splice` שיכת לערך בלבד. היא מקבלת כמה ארגומנטים. הרראשון הוא מזינה מקום בערך להתחיל והשני הוא כמה איברים למחוק. נניח שיש מערך של אבות ורוצחים להוריד את משה, כי הוא לא אחד האבות המקוריים.

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

אם מורידים את משה באמצעות `delete`, יהיה `undefined` בערך ובכיוול יהיו "ארבעה אבות" והשיר "אחד מי יודע" ישتبש לגמרי. אז מה עושים? משתמשים ב-`splice`. מה המקום של האיבר שרצו למחוק (משה)? המקום ה-2; זה אברהם, 1 זה יצחק ו-2 משה. כמה איברים רוצים למחוק? אחד. איך תיראה המהикаה? כך:

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

מערכות ומחוזות טקסט

כדי להגדיל את השמחה ואת הבלבול הכללי, ג'אוּהַסְקְּרִיפְט מחוזות טקסט חולקות לא מעט תוכנות עם מערכיהם והן מתנהגות בחלוקת מהמרקם כמו מערכיהם שבהם האיבר הראשון הוא האות הראשונה, האיבר השני הוא האות השנייה וכך הלאה. שםו לב לمثال לדוגמה זו:

```
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 להתייחס אליה כערך ולשלוף את האות הראשונה באמצעות [0] ואפשר גם לשלוף את האות האחרון בדיק כפוי שלופים את האיבר האחרון בערך. האופרטורים delete וגם push ו-pop לא יעמדו, 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.

פתרון:

```
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 ואו 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 לערך בלבד, אפשר לדוחף ערך בטור האיבר האחרון בערך. בתוך הסוגרים העגולים שלאחר ה-push מכניסים את הערך שרצים שייהה בערך.

תרגיל:

למערך הראשון שיצרתם, הוסיפו את הספרה 0 בתחילתה, 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:

הדחיפת של ערך כאיבר הראשון של המערך והזזה של כל האיברים נעשו באמצעות הפונקציה `unshift`, כמו push מקבלת את הערך שורצים שהוא ראשון בערך, במקרה זה 0.

תרגיל:

בערך 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:

הפונקציה `shift`, שיחודה לערך, מסירה את האיבר הראשון ומוסיפה את כל שאר האיברים.

תרגיל:

בערך 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:

האופרטור `delete` יכול למחוק משתנים וגם איברים בערך. במקרה הזה רציתם למחוק את האיבר החמישי ולפיכך היהם צריכים לכתוב `delete myArray[4]`; למה ? כיון שהערך מתחילה ב-0 והאיבר החמישי יהיה `myArray[4]`;

this ו-new

אפשר ליצור אובייקטים EVEN אם באמצעות המילה השמורה new. בג'אוהסקריפט משתמשים המון בטכניקה זו ליצירת אובייקטים שונים דרך פונקציות. מגדירים פונקציה אז, 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 הוכנתה למשתנה. את הפונקציה הזו מאתחלים באמצעות new. שמו לב שימוש מתחלים את הפונקציה. ברגע שימושים במילה new, הפונקציה מזוריה אובייקט.

הפונקציה שמתוכננת להיקרא באמצעות new נקראת פונקציה בנאית, ובאנגלית constructor. בתשעיה מקובל לכתוב שם של פונקציה בנאית באות גדולה 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 בנאית המשמעות של this היא האובייקט שייווצר כתוצאה מהפונקציה. this מצביע לאובייקט – ככלمر 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 בנאית שהוגדרה קודם, אבל במקום פונקציה ריקה, יש בתוך הפונקציה הכרזה על תכונת foo שיש לה ערך bar. האובייקט שנוצר מהפונקציה הבנאית הוא לא רק אובייקט ריק אלא אובייקט שיש לו foo. כל מה שמכריזים עליו באמצעות this קיבל 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 בניתה. הפונקציה הבניתה הcrize על שם פרט, על שם משפחה ועל תוכנות נוספות של הלוקה. אם תציצו אל האובייקט באמצעות הקונסולה, תראו שהוא מכיל את כל הארגומנטים שהגדרתם באמצעות ה-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 לקוב אחידים, יוצרים פונקציה בונה אחידה לאובייקט לקוב ומכניסים 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 בניתה אחידה לאובייקט משתמש. משתמשים במילה השמורה new על מנת להזכיר למשתנה בכל הפעלה אובייקט אחר, בהתאם למשתנים המכניסים 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 שモצתדת לאובייקט ואפשר להפעילה בקבילות מהאובייקט שהפונקציה הבניתה 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 מתודה בפונקציה הבנאית שנקראת `getFullName`. היא מוצמדת ל-`this` כמו הטענות, אבל היא פונקציה ולא מחזורת טקסט. במקרה זהה היאמחזירה את השם המלא של המשתמש המורכב מתוכנת השם הפרטי, שם המשפחה ורואה בינם.

אפשר להציג מושנים פרטיים בתחום הפונקציה הבנאית – פרטיים במובןuai-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. ברגע לחבריו, המשתנה הזה לא מוצמד ל-`this` אלא מוגדר ממש בתחום פונקציה. האם אפשר לגשת אליו מרחוק? לא. רק מה שמצוידים ל-`this` יהיה נגיש החוצה. מה שלא, מוגדר כפרט. לכן צריך לכתחזק 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`, כיוון שהיא פונקציה שימושת לקלט (get) של מושנים פרטיים שאין גישה אחרת אליהם. כמו שיש `get`, שזו פונקציה שקובעת את המשתנה הפרטי:

```
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 בדיקן צריך set ו-get אם אפשר פשוט לחושף את ה-id ב-this כמו שאר המשתנים? התשובה נעוצה במימוש. לפעמים רוצים לעשות ולידציה. למשל, רוצים לוודא שה-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 מכניסים אותו ל-id, ומעכשו המשנה id מכיל את הערך זהה. אם לא, מוחזרים שגיאה ולא משנים את ה-id המקורי.

או מה ההבדל בין הגדרה רגילה של אובייקט לפונקציה בנית? בדרך כלל משתמשים בפונקציה בנית על מנת להגדיר אובייקטים שיש להם תפקוד – ייחידת תוכנה שיש לה תוכנות ומוגדות. דבר על כך בהמשך. בדרך כלל קוד

מודרני של ג'אווהסקרייפט ארزو אובייקטיבים שבוססים על פונקציה בנית. ומה עם אובייקטיבים רגילים? בהם משתמשים בדרך כלל לאחסן מידע בלבד.

בالمושך תראו את התועלת שיש בפונקציה בנית כאשר מפתחים מערכות מורכבות יותר. בינוויים, זכרו שברגע שאתם רואים `new`, זה סימן שתקבלו אובייקט חדש. נדבר על פונקציות בניות בפרק על אובייקטיבים מובנים בג'אווהסקרייפט.

תרגילים:

צרו פונקציה בנית לאובייקט מכונית המתקבלת שם, צבע וນפה מנوع. לכל מכונית יש מספר זיהוי ייחודי המורכב מהיבור של דגם, צבע וນפה מנוע. למשל, אם שם היצרן של המכונית הוא `opel`, הצבע הוא `white` וນפה המנוע הוא 1,200, מספר הזיהוי שלו יהיה `opelwhite1200`. צרו לאובייקט מכונית פונקציה המחזירה את מספר הזיהוי. הדגם, הצבע וນפה המנוע הם נתונים שהשופים החוצה.

פתרונות:

```
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:

יווצרם פונקציה בנית המתקבלת שלושה ארגומנטים: שם, צבע וນפה מנוע. שלושת הארגומנטים הללו נחשפים החוצה באמצעות `this`. בתוך הפונקציה הבנית יוצרים `modelNumber`, שמורכב משלושת הארגומנטים אלו. לא הושפים אותו החוצה באמצעות `this`. השלב הבא הוא ליצור פונקציה שתחזיר אותו. הפונקציה נחשפת החוצה באמצעות `this` ומחזירה את המשתנה הפרטי.

תרגיל:

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

פתרון:

```
let carObjMaker = function (name, color, engine) {
    this.name = name;
    this.color = color;
    this.engine = engine;
    let modelNumber = this.name + this.color + this.engine;
    this.setModelNumber = 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
```

הסבר:

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

תבנית טקסט

מבנה טקסט מאפשר ליצור מחרוזות טקסט בклות ממשתנים שונים. למשל בפרק על מחרוזות טקסט שאפשר לחבר בין מחרוזות טקסט. ככלمر אם קיימים משתנה שיש בו "Hello" ומשתנה שיש בו "World", יתקבל משהו כזה אם תחברו אותם:

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

אם רוצים רוח בין שני המשתנים, צריך להוסיף אותו. למשל משהו כזה:

```
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 משהו כזה:

```
let price = 10;
let currency = '$';
let combined = price + currency;
console.log(combined); // "10$"
```

אפשר גם לא להכניס את סימן הדולר (\$) כמשתנה ולהחבר אליו ישירות אל המשתנה הראשון:

```
let price = 10;
let combined = price + '$';
console.log(combined); // "10$"
```

הבעיה היא שזה מסורבל מאוד, וככל שהlf הומן ותוכנות הג'אווהסקריפט התפתחו והפכו למורכבות יותר. היה אפשר למצוא בתוכנות מבוססות ג'אווהסקריפט תפלצות מהסוג זהה (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 מסורבלים מאוד. אבל מואוד. לפיכך נוצרה דרך להגדיר "מבנה" בג'אווהסקריפט או, נכון יותר, דרך ליצור מחרוזת 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 בגרש שעובר בדיקת כמו גרש רגיל' או גרשים כפולים " בכל מה שנוגע לטקסט. ככלומר, משחו כזה:

```
let myVar = `Hello world`;
```

בהחלה יעבדו, ו-myVar ייחס למחוזות טקסט לגיטימית לכל דבר. אבל לגרש המסלול יש יכולת שאין לגרשיים הרגילים, והיא להכיל תבנית. אם רוצים להכניס משתנה מסוים לתוך מחוזות הטקסט צריך להקיף אותו בדולר (\$) ובסוגרים מסולסים, והוא ייכנס למחוזות הטקסט בשלמותו. הינה 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 להלוטין לדבר הזה:

```
let welcomeString = 'Hello, ' + user.name + '. How are you doing? Good ' +  
user.localtime + '!';
```

מה נראה טוב יותר ומובן יותר? התשובה ברורה.

tabniot מטפלות באופן נאה מאד בירבי שורות, ואילו מחוזות טקסט רגילות לא מסוגלות להתחמಡ עם ריבוי שורות. אם נוסיף 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 שהתשובה ברורה.

בשנים האחרונות יש מעבר חד-משמעות אל שימוש בתבניות טקסט בגלואוסקריפט, ואני ממליץ גם לכם להשתמש בהן.

תרגילים:

נתון מערך שיש בו בוקר וערב.

```
const timeOfDay = ['Morning', 'Evening'];
```

הציגו באמצעות התבנית טקסט את המשפט "Good Morning" ללא שימוש באופרטור חיבור.

פתרון:

```
const timeOfDay = ['Morning', 'Evening'];
let welcomeString = `Good ${timeOfDay[0]}`;
console.log(welcomeString); // "Good Morning"
```

הסבר:

מחרוזת הטקסט שאותה מדפיסים מוקפת בגרש מסולסל ` מכל צד. הגרש מסמל תבנית טקסט. כיוון שרוצים את האיבר הראשון במערך, כלומר את:

```
timeOfDay[0]
```

מקיפים אותו בסוגרים מסולסלים שבתחילהם יש \$:

```
`${ timeOfDay[0] }`
```

ומשביצים 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, ומהזירה מחרוזת טקסט של שני הארגומנטים מחוברים. למשל, אם מעבירים 30 ו-\$ הפונקציה תחזיר 30₪.

פתרון:

```
function giveMeLocalAmount(amount, currency) {
    let answer = `${amount}${currency}`;
    return answer;
}
let NISAmount = giveMeLocalAmount(30, '₪');
console.log(NISAmount); // "30₪"
```

הסבר:

cotuibim פונקציה רגילה שמקבלת שני ארגומנטים, amount ו-\$-currency. שני הארגומנטים הללו נכנסים למחרוזת טקסט אחת שМОקפת בגרש מסולסל ` מכל צד. הגרש הזה מציביע על הימצואת של התבנית טקסט שקל משנה שנמצא בתוכה ומוקף בסוגרים מסולסלים וב-\$ – ערכו ייכנס למחרוזת הטקסט. מחרוזת הטקסטפה היא פשוטה:

```
`${amount}${currency}`;
```

כלומר, מחרוזת הטקסט היא המספר והערך צמודים. מהזירים את מחרוזת הטקסט זה. על מנת לבדוק אותה בודקים את הפונקציה – קוראים לה עם שני ארגומנטים, 30 ו-\$-סמל המטבע של השקל (МОקף בגרשיים כיון שהוא מחרוזת טקסט), ובודקים את התוצאה. הדרישה תראה שצדקתם.

לולאות

לולאות מאפשרות לעבור על האיברים של מערך או של אובייקט בקבוצות רבה. יש לא מעט לולאות בג'אווהסקייפט. חלקן מיועדות לערכים וחלקן לאובייקטים, וכל אחת מהן משמשת למטרה אחרת עם יתרונות וחסרונות משלها. לשם הבהרה – אוי משתמש במונח "lolalla" לכל איטרציה שהיא.

לולאת **for**

לולאות מאפשרות, בעצם, להריצן קטע קוד שוב ושוב ושוב, כמה פעמים שרוצים. מספר הפעמים תלוי בתנאי מסוים. כך למשל אם רוצחים שקווד ירוץ עשר פעמים, יוצרים משתנה ומציינים שהקווד הזה ירוץ כל עוד המשתנה קטן מ-10 ובכל ריצה מעלים את המשתנה ב-1. כשהמשתנה הגיע ל-10 הריצה תיעצר.

איך עושים את זה? כך:

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

קטע הקוד הזה נקרא "lolalla" (lolalla) והוא ירוץ עשר פעמים בדיקוק, מ-0 ועד 9. הנה ניתוחו: ביטוי ה-for מורכב מהמילה השמורה for ומסוגרים עגולים שבתוכם ביטוי ה-for, המורכב משלושה חלקים:

```
1  
for (let i = 0; i < 10; i++) {  
    console.log('Iteration number ' + i);  
}
```

החלק הראשון הוא הגדרת המונה. המונה הוא המשתנה שעולה, או יורדת, בכל פעולה של הלולאה. במקרה הזה קובעים את שמו **i** ומגדירים שהוא יהיה שווה 0.

החלק השני הוא עד متى תרוץ הלולאה. קובעים שהוא תרוץ כל עוד **i** קטן מעשר. אפשר להשתמש فيه בכל אופרטור השוואתי.

החלק השלישי הוא מה שקרה למונה בכל פעולה של הלולאה. במקרה הזה משתמשים באופרטור שמוסיף 1. ככל פעם שהlolalla רצתה, **i** עולה ב-1. בתוך הטעוריים המסלולים נמצא מה שקרה בכל פעם שהlolalla רצתה.

הבה נראה מה קורה ביריצה הראשונה:

ריצה ראשונה

```
i = 0      מעלים את i באחד      עומד בתנאי
↑          ↑                  ↑
for (let i = 0; i < 10; i++) {
    console.log('Iteration number ' + i);
}
```

ו שווה ל-0. נעשית בדיקה אם i קטן מ-10. כיוון שהוא קטן מ-10, מה שיש בתחום הלולאה עובד, הקונסולה מדפסה את i ומעלים את i ב-1. מגיעים ליריצה הבאה:

ריצה שנייה

```
i = 1      מעלים את i באחד      עומד בתנאי
↑          ↑                  ↑
for (let i = 0; i < 10; i++) {
    console.log('Iteration number ' + i);
}
```

פה כבר i שווה ל-1. נכוון, כתוב פה `let i = 0`, אבל זה תקף אך ורק לריצה הראשונה של הלולאה. בריצה השנייה הולולה תזוכר שלא צריך לאותחל את i אלא רק להעלות אותו. כיוון שהוא קטן מ-10 מרים את מה שיש בתחום הסוגרים המסולסים ואחרי כן מעלים את i ב-1. בריצה הבאה הוא יהיה שווה ל-2, ונמשיך לריצה הבאה.

ריצה שלישית

```
i = 2      מעלים את i באחד      עומד בתנאי
↑          ↑                  ↑
for (let i = 0; i < 10; i++) {
    console.log('Iteration number ' + i);
}
```

פה i שווה ל-2, כיוון שהוועלה ב-1 בפעם הקודמת. 2 קטן מ-10 והתנאי עדין תקף. הולאה מרים את מה שיש בתחום הסוגרים ומעלת את i ב-1. בריצה הבאה הוא יהיה שווה ל-3 וכך הלאה. עד שנגיע לריצה האחרונה:

ריצה עשירית ואחרונה

```
i = 9      מעלים את i באחד      עומד בתנאי
↑          ↑                  ↑
for (let i = 0; i < 10; i++) {
    console.log('Iteration number ' + i);
}
```

הሪיצה الأخيرة היא הריצה העשירית. i כבר שווה ל-9. מרייצים את מה שקרה בתחום הסוגרים המסולסים ומעלים את i ב-1. עכשו הוא שווה ל-10. בריצה הבאה התנאי לא מתקיים, 10 לא קטן מ-10 והלולאה נעצרת. שימו לב: כל ריצה או סיבוב של הלולאה נקראים "אייטרציה". זה המונח הסטנדרטי.

מוכן שאפשר לעשות לוולה שיורדת. שימו לב:

```
for (let i = 5; i > 0; i--) {
    console.log('Iteration number ' + i);
}
```

אם מרייצים את הלולאה זו מקבלים את הഫוסות 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 והציבו על 5.
2. קביעת התנאי שהלולאה תרצו כל עוד i גדול מ-0.
3. בכל אייטרציה i יורך ב-1.

הבה נראה מה קורה באיטרציה הראשונה:

```
ריצה ראשונה באיטרציה הבאה  
i = 5 עומד בתנאי יהיה שווה ל 4  
  
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

באייטרציה הראשונה מתחילה את **ז** וקובעים אותו על **5**. אם הוא עומד בתנאי? כן. **5** גדול מ-**0**. מדפיסים את מה שיש בתחום הסוגרים המסלולים ומורידים את **z** ב-**1** באמצעות האופרטור **--**.

```
ריצה שנייה  
באיורציה הבאה  
יהיה שווה ל 3  
עומד בתנאי  
i = 4  
  
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

באיירציה השניה נ כבר שווה ל-4. הוא עומד בתנאי, 4 גדול מ-0, לפיכך מה שכתוב בתוך הסוגרים הממולאים קורה (יש הדפסה בקונסולה) ו-ז מופחת ב-1. הלאה!

```
ריצה שלושת  
באיורציה הבאה  
יהיה שווה ל 2  
עומד בתנאי  
i = 3  
  
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

באיורציה השלישית ? שווה ל-3, עדין יש עמידה בתנאי. מרכיבים את מה שקרה בתוך הסוגרים המסלולים ואז מורידים את ? ב-1.

ריצה רביעית באיטרציה הבאה
'יהה שווה ל 1'
עומד בתנאי
 $i = 2$

```
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

באייטרציה הריביעית נ שווה ל-2. עدين גדול מ-0. מתרחשת הרצאה נוספת של מה שיש בסוגרים המסלולים והפחתה של 1 מ-

```
ריצה חמישית  
ואחרונה באיטרציה הבאה  
יהי שווה ל 0  
עומד בתנאי  
i = 1  
for (let i = 5; i > 0; i--) {  
    console.log('Iteration number ' + i);  
}
```

באיטרציה החמישית והאחרונה ? שווה ל-1. 1 גבול מ-0 ולפיכך מה שיש בסוגרים המסלולים ירוץ. ? יורץ ב-1 ובריצה דבאה יהיה 0.

באייטרציה שלא תתקיים יש בדיקה אם σ גדול מ-0. כיוון ש- σ שווה ל-0, התנאי לא מתקיים. 0 לא גדול מ-0 ולפיכך האיטרציה לא תרוויז. הלולאה נעצרת:

איטרציה שלא תקין

שימנו לב: מקובל מאוד לקרוא למשתנה של הלולאה בשם `i`, או `index`.

אמנם לולאות של הפחתה או הוספה הן הלולאות הקלאסיות, אבל אפשר באמת להתפרק. הינה למשל לולאה שמוסיפה לו-ב כל 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"
```

טרגול טוב יהיה לכתוב את הלולאה על נייר ולנסות להבין למה הפלט הוא כזה. הבה נתחיל. באיטרציה הראשונה, מן הסתם הוגדר כך שהיא שווה ל-0. כיון ש-0 קטע מ-12, ההדפסה בקונסולה מתרחשת ואז מעלים את 0-ב-4.

באיירתציה השנייה, *i* שווה ל-4. מן הסתם 4 קטן מ-12 ומה שיש בתוכה הסוגרים המסלולים יקרה. *i* עלה בעוד 4.

```
איטרציה מס' 2  
i = 4           8 = ? בריצה הבאה  
  
for (let i = 0; i <= 12; i = i+4) {  
    console.log('Iteration number ' + i);  
}
```

באיורציה השלישית, १ שווה ל-८. ८ עדין קטן מ-१२ והפעולה בלולאה תתקיים. १ עלה ב-४, ל-१२.

אייטרציה מס' 3

```
i = 8          i = 12 ביריצה הבאה
↑             ↑
for (let i = 0; i <= 12; i = i+4) {
    console.log('Iteration number ' + i);
}
```

זו האיטרציה האחרונה שתתקיים. `i` שווה ל-12 ועדיין עומד בתנאי. למה? כי התנאי הוא קטן או שווה. 12 שווה ל-12 ויש עמידה בתנאי. הפעולה תרוץ ו-`i` יעלה ל-16. ביריצה הבאה נראה ש-16 גדול מ-12, ולפיכך הפעולה לא תתקיים לעולם. בום!

אייטרציה מס' 4 ואחרונה

```
i = 12          בפעם הבאה, שלא תרוץ
↑             ↑ i = 16
for (let i = 0; i <= 12; i = i+4) {
    console.log('Iteration number ' + i);
}
```

דיברתי על סקופים בפרק על הפונקציות. גם בלולאות יש סקופ. כל מה שקורה בתחום הסוגרים המסלולים נחשב לסקופ שלו. זו גם אחת הвойדות של `let` לעומת `var`. שהיון נוהג בעבר `let` שומר על סקופ שלו בלולאה. אם תנסה לגשת אליו מחוץ ללולאה 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 לו-`i` מבוחרן, וזה מלביל 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 שלו שמתפוגג בכל אייטרציה.

לולאת for משחקת יפה מאוד עם מערכים. הבה נדגים. נניח שיש מערך שורצים להדפיס את כל תוכנו. איך אפשר לעשות את זה? כך:

```
const myArray = ['first', 'second', 'third', 'fourth'];
console.log(myArray[0]);
console.log(myArray[1]);
console.log(myArray[2]);
console.log(myArray[3]);
```

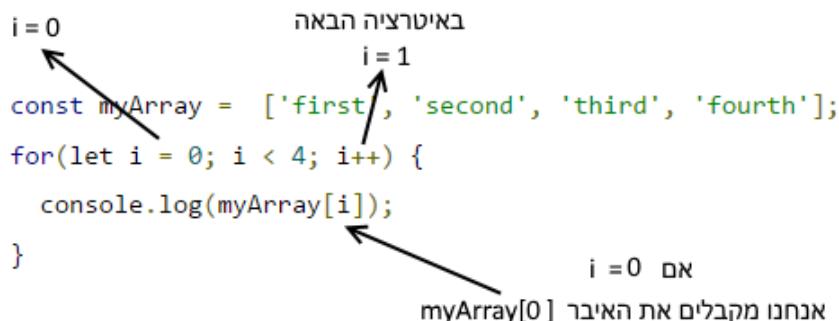
אבל זה אורך ומצבן, במילוי אם יש הרבה איברים. במקום זה אפשר לכתוב לולאת פשוטה:

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < 4; i++) {
    console.log(myArray[i]);
}
```

כידוע, *i* משתנה בכל איטרציה, מ-0 ועד 4 (0, 1, 2, 3). אפשר להציב אותו בתור מספר האיבר במערך ולקבל אותו!

הבה נבחן את האיטרציות:

איטרציה מס' 1



באייטרציה הראשונה *i* מאותחל ושווה ל-0. מן הסתם 0 קטן מ-4, ומה שיש בתחום הסוגרים המסוללים מתקיים. כיוון *sh-0=i* אפשר להשתמש בו. זה בדוק כמו לכתוב:

```
let i = 0;
console.log(myArray[i]);
```

פשוט במקרה הזה *i* כבר קיים. לא צריך להגיד אותו. מה שיודפס בקונסולה הוא *first*. עוברים לאייטרציה הבאה.

באייטרציה הקיימת נקבע שווה ל-1. התנאי מתקיים כי $1 < k < 4$, ומה שיש בתחום הסוגרים הממוסללים רצ. אפשר להדפיס את האיבר הראשון (שהוא האיבר השני, כי במערכות מתחילה מ-0). עוברים לאיטרציה הקיימת, שבה נ^{עולה ל-2.}

```
i = 2                                באיטרציה הקיימת  
                                         i = 3  
const myArray =  ['first', 'second', 'third', 'fourth'];  
for(let i = 0; i < 4; i++) {  
    console.log(myArray[i]);  
}  
                                         ↓  
                                         אם i = 2  
                                         myArray[2]
```

גם פה, כמו באיטרציות הקודמות, התנאי מתקיים. 2. קטן מ-4. אפשר להשתמש ב-n, שכרגע הוא 2, להדפים את האיבר השליישי במערך ולעבור לאיטרציה הבא, שבה n עולה שוב, הפעם ל-3.

```
i = 3                                באיטרציה הבאה  
                                         i = 4  
  
const myArray = ['first', 'second', 'third', 'fourth'];  
for(let i = 0; i < 4; i++) {  
    console.log(myArray[i]);  
}  
  
                                            i = 3 אם  
אנו מקבלים את האיבר myArray[3]
```

באייטרציה האחרונה ♡ שווה ל-3 ועדין עומד בתנאי. אפשר לקחת את ♡ ולהדפיס את האיבר הרביעי בסדרה. באיטרציה הבאה ♡ שווה ל-4, אבל זה כבר לא מעניין כי 4 לא קטן מ-4 וה坦אי לא מתקיים. זו הריצה האחרונה של הלולאה הזו והערך הודפס.

הבעיה בגישה זו היא שמניחים שגודל המערך הוא 4. אבל מה אם לא יודעים מה גודל המערך? זוכרים שאפשר להלץ את גודל המערך באמצעות שימוש בתכונה המוחדרת `length`? אפשר להשתמש בה בלולאה:

```
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < myArray.length; i++) {
    console.log(myArray[i]);
}
```

לולאה (ולא `myArray.length`) בדוק גודל המערך.

לולאה אינסופית

יש כאן לולאה בעייתית. מה הבעיה בה?

```
for (let i = 0; i >= 0; i = i++) {
    console.log('Iteration number ' + i);
}
```

הבעיה בלולאה זו היא שהוא לא תעצור לעולם. למה? כי היא תתקיים כל עוד ♡ גדול מ-0, ו-ו-ן גדול ב-1 כל הזמן ולעולם לא יהיה קטן מ-0. הלולאה הזו, שלא הסתיים לעולם, נקראת לולאה אינסופית. בדרך כלל, דברים כאלה יקרים אחרי המון איטרציות. אם מריםם את זה בדףן, מתיישבו הלשוניות תהיה לא יציבה ותקרוס. כשוכותבים לולאה (ולא משנה מאיזה סוג) והדףן קופא, סביר להניח שזאת הסיבה.

לולאת while

לולאה נוספת היא לולאת `while`. היא פשוטה יותר מlolאת `for` ומחייבת תנאי אחד בלבד עד שהוא נכון הלולאה מתיקיימת. למשל:

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

כל עוד התנאי מתקיים, הלולאה רצה.

```

כל עוד התנאי הזה שווה ל true
הולאה תרוץ
let i = 0;

while (i < 10) {
    console.log(i);

    i++;
}

```

בלולאות כאלה, כמובן, חייבים להגיד את `i` מחוץ ללולאה, אחרת בכל ריצה של הלולאה יתאפס משתנה ה-`i`.

התנאי יכול להיות כל תנאי שהוא, כולל תנאים מורכבים, ובגלו זה הלולאה הזו שונה בעיקרו מLOOPAT for שמקובל לכתוב אותה רק עם תנאי אחד. למשל הלולאה המורכבת הזו:

```

let i = 0;
let n = 20;
while (i < 100 && n > 0) {
    console.log('n' + n);
    console.log('i' + i);
    i++;
    n--;
}

```

היא עובדת כל עוד `n` חיובי ו-`i` קטן מ-100. ברגע שאחד משנייהם לא נכון, היא עצור. קצת מורכב ומלבלב, אבל כדי לזכור את זה. אחד השימושים שלה הוא אם (מספר מסוימת) רוצים ליצור לולאה אינסופית, ואז כותבים משהו כזה:

```

while (true) {
}

```

אבל צריכה להיות לכם סיבה טובה מאוד ליצור לולאה אינסופית.

לולאת do while

עוד סוג של לולאת `while` הוא לולאת `do`. סוג הלולאה זהה מאוד לא שכיח אבל כדאי להכיר אותו. הלולאה הזה ייחודית כי בניגוד לLOOPAT for ולLOOPAT while, היא תמיד מבצעת פעולה אחת ואז בזקמת את התנאי, במקומות לבדוק את התנאי ואז לروعן. כמובן, תמיד הלולאה רצה לפחות פעם אחת.

הינה דוגמה:

```
let i = 100;
do {
    console.log(i);
    i++;
}
while (i < 0); // 100
```

יש כאן מילה שモורה, do, ומיד אחריה סוגרים מסולסלים. מה שיש בתוכם תמיד יקרה, ורק אחר כך יגיע ה-h-while שבודק את התנאי. כאן, למשל, הלולאה תתקיים כל עוד זה הוא שלילי. לפני התנאי קבועים את זה על 100. לפि כל הלולאות שלמדנו זה אומר שהלולאה לא תרוץ לעולם, אבל כיוון שימושם ב-do while, הלולאה תרוץ לפחות פעם אחת.

לולאת forEach

הלולאות שלמדנו עד כה הן לולאות שעבודות עם תנאים, אבל בעולם הגאומטריפט מקובלות יותר לולאות שעבודות עם אובייקטים או עם מערךם באופן שלם יותר. למה צריך את זה? באחד התרגילים בפרק על מערכים רואיתם 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 בלולאת for או עלולים להסתבך כי אי-אפשר להיות בטוחים שהאיבר אכן מוגדר. במקומות להתחיל לשבור את הראש על גודל המערך ועל התאמתו ללולאת for, שלא לדבר על מה שקרה אם יש איברים ריקים, יש דרכים אלגנטיות יותר לעبور על כל האיברים בו. הדרך הראשונה והיודעת מכולן היא שימוש ב-`forEach`. מדובר במקרה שקיים אך ורק במבנה נתונים מסווג מערך (לא אובייקט) ומקבלת פונקציה אנונימית 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 מהרוות טקסט `first`. האיבר במיקום 1 הוא מהרוות טקסט `second` והאיבר האחרון, במיקום 2, הוא מהרוות טקסט `third`. כיוון שמדובר במערך, אפשר להציג לו את מתחות `:forEach`

```
myArray.forEach();
```

הmethodה זו היא פונקציה ש策ריכה לקבל ארגומנט. מה הארגומנט? פונקציה נוספת, שבדרך כלל היא אנונימית. על פונקציות חוץ אנונימיות למדנו בפרק על הפונקציות. הפונקציה שאותה מעביריםפה תזריך בכל איבר של המערך. הארגומנטים שלה הם האיבר עצמו והמספר שלו:

```
(value, key) => {
  console.log(`value: ${value}, key: ${key}`);
}
```

זה כמו לכתוב משהו כזה:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach(iteratorFunction);
function iteratorFunction(value, key) {
  console.log(`value: ${value}, key: ${key}`);
}
```

הקוד שלעיל זהה כמעט לחולtin לקוד של פונקציה אנונימית, אבל פה יוצרים עוד פונקציה ו"מלככים" את הסkop. מקובל מאוד להשתמש בפונקציה אנונימית וזה מה שעשווים כאן. הפונקציה האונימית רצתה על כל איבר. מאיפה באים הארגומנטים value ו-key שקראו להם? זה הכוח של לולאת forEach, היא שותלת 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:

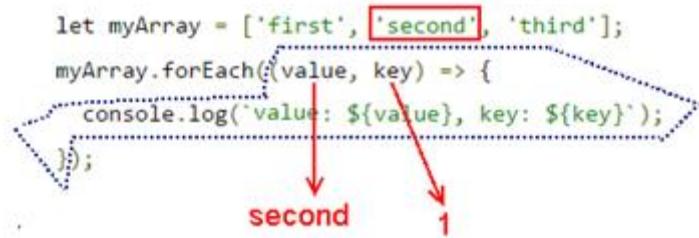
```
let myArray = [first, 'second', 'third'];
myArray.forEach((value, key) => {
  console.log(`value: ${value}, key: ${key}`);
});
```

first
0

מה שיש בפונקציה מופעל. זה הכוח של forEach. אל תבלבלו מהסינטקס המוזר עם החז', בסופו של דבר זו פונקציה. מה שיש בתחום הסוגרים הממולסים רץ ורק הארגומנטים מתחלפים. בריצה הראשונה של forEach, תור

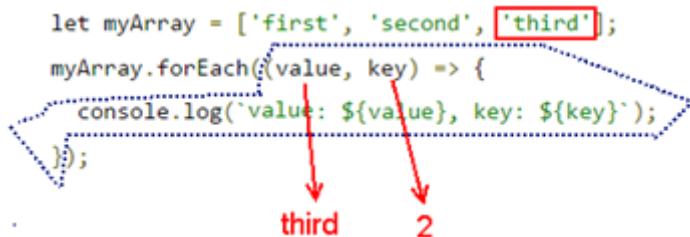
האיבר הראשון להיכנס לפונקציה האנוונית. יש שהוא האיבר של המערך ויש key שהוא המיקום של האיבר במערך. למה? כי מערך תמיד מתחילה מ-0.

אחרי שהאיבר הראשון רץ והפונקציה מסתיימת, מגיעת תור האיבר השני:



הֆונקציה רצתה שוב, אבל ה-key וה-value שלה שונים. הפעם אלו ה-key וה-value של האיבר השני. ה-key הוא מה שיש בתוך האיבר השני במערך, ובמקרה זה - מחזורות טקסט. ה-key הוא המיקום של האיבר השני במערך. למה? כי מערכים מתחילה מ-0.

בפעם השלישייה והאחרונה, האיבר השלישי ייכנס לפונקציה האנוונית:



שמות הארגומנטים נموון 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. הוא תרוץ לכל אורך איברי המערך. למה רוצים לשבור את הלולאה? בגלל אלף ואחת סיבות. בדרך כלל אם מփשים משהו ומוצאים אותו, אין צורך בהרצת כל הלולאה. בדיק בשביל זה יש את לולאת .for of

לולאת for of

לולאת for of פועלת כך: היא לא מחזירה את האינדקס של האיבר במערך, אבל כן מקבלים את האיבר. הנה נניח שיש מערך שהאיברים שלו הם אובייקטים של חפצים. יש את החפץ ויש את המחיר. משагה בסגנון הזה:

```
let orderArray = [{ name: 'pen', price: 11 }, { name: 'pencil', price: 5 }, { name: 'TV', price: 2345 }];
```

לא צריך להיבהל. מדובר במערך שבמוקום מסוימים או מהרוות טקסט יש בו אובייקטים. לכל אובייקט יש שתי תכונות – name ומיל מהרוות טקסט ו- price מכל מספר. לולאת for of 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, אותה הכרנו בלולאת for. אך במקום הסינטקס המוכר יותר, יש הגדרת משתנה שהוא בעצם האיבר התווך במערך, המילה השמורה of ושם המערך. בתוך הטעוריים המסלולים יירוץ 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

הנה 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 תכונה בשם name ותכונה בשם price. כיוון שאנו מדפיסים אותו, מה שראאים 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 שראאים בהדפסה הוא:

```
// "The price of pencil is 5."
```

בהרצתה الأخيرة, האיבר השלישי ייכנס אל תוך המשתנה order בבדיקה כמו האיברים שלפניו.

עכשו נעבור לשבירת לולאה. בואו נניח שהדרישה היא להציג את המוצר הראשון שהמחיר שלו נמוך מ-10. לא את כל המוצרים אלא את המוצר הראשון. דרישת כזו יכולה כמובן להיות אמיתית לחהלוטין. 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 העזרות שהלולאה אכן נשברת ולא ממשיכה 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 עונה לתנאי
    answer = order;
    break;
  }
}
```

יש הדפסה של:

```
// "I am passing pen."
```

אך התנאי אינו מופעל כיוון שהמחיר הוא 11 והתנאי הוא מחיר שקטן מ-10.

באייטרציה 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) {
    עונה לתנאי!
    answer = order;
    הלולה נשבורה
    break;
  }
}
```

יש הדפסה:

```
// "I am passing pencil."
```

כיוון שהתנאי מתקיים, המשפט break מופעל והלולה לא עוברת לאייטרציה השלישית. זה היתרונו הגדול של for – שהיא מזוגת את היכולת של while מבחינה שבירה ואת האלגנטיות של .forEach

lolata map

פעמים רבות רוצים לשנות את המערך עצמו. למשל, הבה נניח שלקווח שmagu ישראלי רואה את המחיר ורוצה להציג לו אותם ב שקלים, כולל להכפיל את המחיר פי ארבעה. איך עושים את זה? אפשר להשתמש בלולאת for,

בלולאת while, בולולאת forEach או בולולאת of – אך אם רוצים לשנות את המערך מקובל מאוד להשתמש ב-map, שמאחרו הקלעים לוקחת את המערך המקורי והופכת אותו לערך חדש. לולאת map מיועדת לשינויים מעריכים והוא פועלת באופן דומה ל-forEach, אלא שהפונקציה האנונימית יכולה להחזיר את הערך החדש שייכנס במקום הערך המקורי, ככלمر 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 מאד לlolאת forEach שעלייה כבר למדנו. ה-value שהפונקציה הפנימית מקבלת הוא האיבר שיש במערך. אבל אפשר לראות שכן הפונקציה מחזירה משהו. מה היא מחזירה? את האיבר החדש שעבר שינוי. האיבר החדש ייכנס במקום האיבר המקורי, 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;
});
```

נכון למקום הראשון במערך המקורי

אפשר להזיר מה שרצים, אפילו מחרוזת או אובייקט מורכב יותר. כל מה שמנסים ייכנס במקומות האיבר הראשון של המערך. באיטרציה השנייה משנים את האיבר השני, ומה שמחזירים נכנס במקומות האיבר השני. הינה:

```
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 }
});
```

ולסימן, באיטרציה האחרונה מקבלים את האיבר השלישי ומה 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 }
});
```

זהו. מה שיש במערך המקורי הוא לא מה שהוא פעם אלא האיברים החדשניים.

lolaa filter

הlolaa האחרונה היא lolaa שמחזיא איברים מהמערך. בעוד map רק משנה איברים, באמצעות lolaa אפשר ליצמצם את המערך. הנה 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 שמדובר להציג רק דברים שהמחיר שלהם גבוה מ-7. אפשר לעשות משהו כזה:

```
writingArray.forEach((value, key) => {
  if (value.price <= 7) {
    delete writingArray[key];
  }
});
```

כלומר, לקחת את המערך ולעבורי עליו עם lolaa. בכל איטרציה (מעבר) של lolaa בודקים את המחיר של האיבר הנוכחי. אם הוא קטן מ-7 או שווה לו, מוחקם אותו. האיבר הראשון, למשל, הוא pen, שהוא price שלו הוא 11. התנאי לא יתקיים כי value.price הוא 11 והוא לא קטן מ-7 או שווה לו. האיבר השני הוא pencil וה-he price שלו הוא 5. במקרה זה התנאי מתקיים. כיוון שלlolaa ה-fforeach נותנת גם את ה-key, הלווא הוא מספר האיבר, שבמקרה של האיבר השני הוא 1 (זוכרים שמערך מתחילה מ-0, נכון?). אפשר להשתמש באופרטור delete, שעליינו למדנו בפרק על מערכים, כדי למחוק את האיבר הזה, וכך הלאה. מה שנתקבל בסוף הוא מערך עם שני undefined, היכן שהיו כל האיברים שמחירים נמוך מ-7.

```
[  
  { name: 'pen', price: 11 },  
  undefined  
  { name: 'paper', price: 10 },  
  { name: 'brush', price: 12 },  
  undefined  
]
```

זה קצת מבאש, ואפשר לעקוף את זה בכל מיני דרכים או באמצעות filter. מדובר בlolaa שזהה ל-map אך מחזירה רק true או false. אם היא מחזירה true האיבר נשאר במערך. אם היא מחזירה false האיבר מתפוגג מהמערך Caino לא היה שם. ברגע lolaa, lolaa הזו לא משנה את המערך המקורי וצריך להחזיר את התוצאה למשתנה אחר.

```

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;
  }
});

```

הוּא מערך המכיל אובייקטים של מוצרים כתיבה. לכל אחד מהם יש שם ומחיר. כדי לסנן את כל האיברים שהמחירם שלהם נמוך מ-7, משתמשים בפונקציה ייחודית למערך שנקראת `filter`. הפונקציה זו מקבלת פונקציה אונומית שאotta כותבים כפונקציה חז. אני רק מזכיר - פונקציה חז היא דרך מסוימת לכתוב פונקציה אונומית (כלומר פונקציה שאון לה שם אלא עוברת כארוגמנט). הפונקציה האונומית מוחזירה `true` או `false`. אם היא מוחזירה `true` האיבר נשאר במערך. אם היא מוחזירה `false` הוא עף מהמערך. פונקציית `filter` מוחזירה את המערך החדש.

הבה ננסה לפרק את זה. בסכוב הראשון, מי שנכנס לתור הבדיקה של ה-`filter` הוא האיבר הראשון:

```

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) { price = 11
    return false;
  } else {
    return true;   הfonktsiya mchizra true
  }
});

```

התנאי לא מתקיים
הfonktsiya mchizra true
האיבר זוכה להשאר במערך

באייטרציה השנייה מגיעת תור האיבר השני:

```
const writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 }, // ← היבר הזה לא זוכה להיכנס לאייר החדש
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filteredArray = writingArray.filter((value) => {
  if (value.price <= 7) { price = 5 // התנאי מתקיים
    return false; // הפונקציה מחזירה false
  } else { // האיבר הזה לא זוכה להיכנס לאייר החדש
    return true;
  }
});
```

הוא כבר לא זוכה להיכנס, כיון שהמחיר שלו הוא 5 והтенאי הוא שכלל מה שקטן מ-7 או שווה לו מקבל false. וכך הלאה. כמו כל לולאה אחרת, הלולאה הזו עוברת על כל המערך. הפונקציה שהעברתנו מהזרה true או false לנבי false. האיברים שמקבלים true זוכים להיכנס למערך החדש, ואלו שמקבלים false – לא. בסופו של תהליך חייבים להחזיר את מה ש-filter מזרה אל משתנה חדש. המערך המקורי נשמר כמו שהוא:

```
];
let filteredArray = writingArray.filter((value) => {
  if (value.price <= 7) { // חיבים להחזיר את התוצאה אל מערך חדש
    return false;
  } else {
    return true;
  }
});
```

לולאה sort

lolalla חשובה נוספת המשמשת למילון מערכים. הלולאה זו לא משנה איברים ולא את גודל המערך, אלא פשוט מסדרת את האיברים במערך. בኒוגוד לאייטרציות אחרת, לא חייבים להעביר פונקציה אונומית שתעשה את הסדר. כביררת מחדל האיטרציה זו מסדרת לפי סדר הא'-ב' – היא ממירה את הערכים למחוזות טקסט ועורך השוואת ביניהם, וכך מבצעת את הסדר. הינה דוגמה פשוטה מאוד שמדגימה את זה:

```
const months = ['March', 'Jan', 'April', 'Dec'];
months.sort();
console.log(months); // ["April", "Dec", "Jan", "March"]
const numbers = [1, 42, 5, 7, 565656];
numbers.sort();
console.log(numbers); // [1, 42, 5, 565656, 7]
```

מובן שהעוניינים מתחילה לסתוך אם במערך יש אובייקטים ולא רק מספרים או מחוזות טקסט, אבל לא נכסה את אייטרציית sort במלואה בספר זהה.

לולאות על אובייקטים

כל הלולאות שלמדונו עד כה משகחות יפה עם מערכים ובאמצעותן אפשר לעבור על כל איבר ואיבר 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 להציג את כל נתונים האובייקט (בלי לדעת מה שמות התכונות שלו). איך עושים את זה? לולאת for או לולאת while לא יעזור ולולאות forEach או of מיועדות למערכים בלבד ויניכו שגיאות אם תנסו להשתמש בהן. יש שתי שיטות למעבר על אובייקטים. הראשונה "לפי הספר", שמעטם משתמשים בה, והשנייה הפופולרית יותר.

לולאות for in

מדובר בלולאה שעוברת על כל תכונות האובייקט בדומה ל-for-of. 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 `shmorah`, `for`, ומיד אחריה סוגרים עגולים. בסוגרים העגולים מגדירים משתנה שהוא המפתח. בתוך הלולאה יקבל המשתנה זהה את שם המפתח המקורי. אחר כך מכנים את המילה `shmorah` `in` ואת המשתנה שבתוכו נמצא המערך.

בתוך הסוגרים המסלולים 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]`

בואר נראה איך זה קורה. ראשית, חשוב להבין ש-`key` מגיע מהמפתחות של האובייקט. הוא לא מכיל את הערכים:

```

const userObject = {
    userId: 12,
    userName: 'barzik',
    age: 40,
};

for (let key in userObject) {
    console.log(`key: ${key}, value: ${userObject[key]}`);
}

```

בアイテרציה הראשונה שמתרחשת, `key` הוא המפתח הראשון שיש באובייקט, והוא מהרווז הטקסט `.userId`. באמצעות המפתח אפשר לחלץ את הערך. איך? למדנו בפרק על אובייקטים שאם המפתח נמצא בשם של משתנה אפשר לחלץ אותו באמצעות שימוש בסוגרים מרובעים, וזה בדוק מה שעושים. את המפתח ואת הערך מדפיסים באמצעות שימוש בתחום טקסט פשוטה, שגם עליה למדנו:

```

const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
  userObject[userId] = 12
}

```

באייטרציה השנייה, key הוא המפתח השני של האובייקט, במקורה זהה(userName). ושוב, גם כאן מחלצים את הערך באמצעות שימוש בסוגרים מרובעים כמו בפעם הקודמת:

```

const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) {
  console.log(`key: ${key}, value: ${userObject[key]}`);
  userObject[userName] = 'barzik'
}

```

באייטרציה الأخيرة, key הוא ה-age.

```

const userObject = {
  userId: 12,
  userName: 'barzik',
  age: 40,
};
for (let key in userObject) { key = age
  console.log(`key: ${key}, value: ${userObject[key]}`);
  userObject[age] = 40
}

```

הלואה זו ניתנת לשכירה על ידי break;

בסק הכלול פשוט ונחמד, נכון? אבל האם היא שנדיר למצוא מתכנת שימוש בשיטה זו, מפני שלולאת `for in` מביאה נתונים גם מאובייקטים אחרים שהם פרוטוטיפ – אב טיפוס – של האובייקט שלנו. כרגע זה נשמע סינית וודין לא דיברנו על זה, אבל על קצה המזלג אכן שבדרך כלל אובייקטים לא מגיעים כלוח חלק והם העתק של אובייקטים אחרים להם מפתחות נוספים. לרוב לא מעוניינים בפתחות של האובייקטים האחרים, ולפיכך לולאת `for in` עלולה להיות הרת אסון, וזה הסיבה שבkowski משתמשים בה. אם כבר משתמשים בה, מקובל מאוד לוודא שהתמונה שבודקים היא התמונה 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 `for` לשימוש באובייקטים היא בעייתית ונדר שמשתמשים בה.

Object.keys

הדרך השנייה והপופולרית ביותר היא להשתמש בפונקציה שנקראת `Object.keys` לחילוץ כל המפתחות של האובייקט כערך, ואו לעבור על המערך הזה באיזו לולאה שרוצים, כי זה מערך. איך ניגשים לערך באובייקט כמו בולולאת `for in`; ברגע שיש את המפתח יש גם את הערך. שימוש לב לדוגמה 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-`userObjectKeys`. ועליו עושים לולאת `forEach`. כיוון שמדובר במערך, אפשר לעשות עליו `.forEach`. הנדר ואלגנטiy.

נסכם בטבלה המסבירה על כל סוגי הלולאות:

שם לולאה	יתרונות	הסוג
לולאות למערכים		
for	פשוטה למימוש ניתנת לשכירה באמצעות <code>break</code>	לולאת <code>for</code> לולאה <code>for</code>
while	פשוטה למימוש ניתנת לשכירה באמצעות <code>break</code> אפשר לכתוב תנאים מורכבים	לולאת <code>while</code>
for each	לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיירים ריקים	לולאת <code>for each</code>
for of	פשוטה למימוש ניתנת לשכירה באמצעות <code>break</code> לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיירים ריקים	לולאת <code>for of</code>
map	לולאה שעוברת על המערך ויוצרת ממנה מערך חדש באותו גודל. אפשר לשנות את האিירים בקלות.	לולאת <code>map</code>
לולאות להמרת מערכים		
filter	מוחקת אিירים	לולאת <code>filter</code>
לולאות לאובייקטים		
for in	פשוטה למימוש ניתנת לשכירה באמצעות <code>break</code> לא צריך לדעת את גודל האובייקט אין לולאה אינסופית לא נכנסים לאיירים ריקים	לולאת <code>in</code>

<p>מקבלים רק את המפתח של האובייקט וצריכים לחלק את הערך בלבד</p> <p>אפשר להשתמש בlolalot של המערך ואז מקבלים את כל ההוראות של lolalot</p> <p>מקבלים את כל ההוראות של lolalot שבחורתם</p>	<p>אפשר להשתמש בlolalot של המערך ואז מקבלים את כל ההוראות של lolalot שבחורתם</p>	<p>שימוש ב- Object.keys</p>
---	--	------------------------------------

תרגיל:

במציאות לולאות for, הדפסו 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 לולאה פשוטה שמתחילה כאשר i שווה ל-0. היא אמורה להימשך כל עוד i קטן מ-3. באיטרציה הראשונה כאמור i שווה ל-0. בשניה 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 ששווה ל-0 והתנאי הוא שהlolalot תרוץ כל עוד המספר קטן מאורך המערך, במקרה זה 5. משתמשים במונה i שזמין על מנת להדפיס את האיבר המתאים במערך. למשל, באיטרציה הראשונה i שווה ל-0. התנאי מתקיים כי 0 קטן מ-5 (אורך המערך). משתמשים ב- i על מנת להדפיס את האיבר שנמצא במקום 0. באיטרציה השנייה, i שווה ל-1, התנאי מתקיים כי 1 קטן מ-5 (אורך המערך) ומשתמשים ב- i כדי להדפיס את האיבר שנמצא במקום 1, וכך הלאה.

תרגיל:

נתון מערך של אובייקטי הזמנות:

```
const invoices = [{ id: 1, price: 10 }, { id: 2, price: 32 }, { id: 3, price: 40 }];
```

הדפיסו את סכום ההזמנה במערך:

```
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:

যוצרם לוֹלָה שעוברת על המערך. המערך הוא לא של ערכים פרימיטיביים אלא של אובייקטים, אבל גם אובייקטים אפשר לשלוף באמצעות `[i]`, כשה-`i` משתנה לפי מספר האיטרציה. לפני שהlolאה רצה, יוצרים משתנה בשם `amount` ששוּה 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 הזה האובייקט נמצא באיבר במערך ולא במשתנה עם שם משלהו, אז אפשר לגשת למחיר באמצעות:

`invoices[i].price`

כש-`i`, להזכירם, משתנה לפי האיטרציה.

תרגיל:

צרו לוֹלָה while שמקבלת מספר וסופרת ממנו עד 0.

פתרונות:

```
let i = 10;
while (i >= 0) {
    console.log(i);
    i--;
}
```

הסבר:

לוֹלָה 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.

תרגילים:

צרו לולאת while שמקבלת מספר וסופרת ממנו עד 0. אם המספר הוא 0 או שלילי עדיין תהיה הדפסה אחת של המספר, למשל 1- זהו.

פתרונות:

```
let i = -1;
do {
    console.log(i);
    i--;
} while (i >= 0);
```

הסבר:

לולאת do while היא אידיאלית אם רוצים פעולה אחת לפחות, גם אם התנאי לא מתקיים. במקרה זה קודם הלולאה עובדת לפחות פעם אחת ואז התנאי נבדק. כיוון שהמספר הוא שלילי, התנאי לא מתקיים והלולאה לא תמשיך לרגע, אבל היא עבדה פעם אחת והדפסה את המספר.

תרגילים:

נתון מערך:

```
let myArray = [
    { userName: 'Moshe', age: 20, },
    { userName: 'Yaakov', age: 25, },
    { userName: 'Ran', age: 40, },
]
```

מדובר בפרופילים של משתמשים באתר היכרויות. כתבו פונקציית forEach שתחדפיס בקונסוללה 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 בפונקציה forEach שמקבלת פונקציה אונונימית מסווג חז. הפונקציה זו נראית כך:

```
(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 שבחרתי 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 תכונת age שלו, ומדפסים את מי שגילו 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 ומערך ומחזירה אך ורק מערך משתמשים שהגיל שלהם שווה למספר הזה 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, }]
```

הסבר:

יווצרם פונקציה שמקבלת שני ארגומנטים – מספר מסויים ומערך. משתמשים בפונקציה filter על מנת לסנן את המערך ולהשאיר שם רק את המשתנים שרצו. איך נראה פונקציית filter? בדיק ככה:

```
(value) => {
    if (value.age < requestedAge) {
        return false;
    } else {
        return true;
    }
}
```

הפונקציה פשוטה יחסית ולא צריך להתבלבל מההץ. ב-value יש בכל פעם איבר אחר מהמערך. אם פונקציית החץ מחזירה true הוא נשאר במערך. אם לא, הוא עף החוצה בובשת פנים. במקרה זהה מוחזרים false על כל מי שהגיל שלו קטן מהמשנה requestedAge, שאותו מקבלים מהפונקציה.

אל תשכחו שהיבטים להחזיר את מה שמתקבל מה-filter אל משתנה אחר, כיון שפונקציית filter לא משנה את המערך המקורי.

כל מה שנותר לעשות הוא לבדוק את הפונקציה. מעבירים לה 40, לדוגמה, ואת המערך של המשתנים ורואים שמקבלים משתמש אחד שהוא בן 40, כיון ש-40 לא קטן מ-40, הוא מקבל true בפונקציית הфиילטר.

ג'אוהסקרייפט בסביבת דפדפן

עד כה, כל התרגולים היו עם הקונסולה בלבד. סביבת העבודה היא בדפדפן ועם עורך טקסט, אבל בניגוד לדבר היחידי שעשיהם בדפדפן היה בקונסולה. הגיע הזמן להתחיל למש את הידע שלנו ולראות מה אפשר לעשות בעזרתו למציאות.

כיום ג'אוהסקרייפט רצה במגוון אדריכל של סביבות, אבל בעבר הסביבה היחידה שבה היא רצה הייתה הדפדפן. כמובן, באתר אינטרנט השתמשו בה כדי ליצור התנהוגיות אינטראקטיביות עם המשתמש, אнимציות, אפקטים, וידאו ועוד. באופן עקרוני, כל הידע שלמדו עד עכשיו הוא הבסיס הנדרש לצירוף כל הדברים האלה בדפדפן. יכול להיות שלולאות, אובייקטים ופונקציות נראים לא מעניינים לעומת אнимציות וסרטונים, אבל הם מה שפועל מאחורי הקלעים כדי לאפשר את כל הדברים היפים בדפדפן.

הספר הזה מלמד ג'אוהסקרייפט ולא HTML, שהוא עולם ומלאו. בפרק זה נלמד HTML באופן בסיסי בלבד, רק מספיק כדי להבין איך ג'אוהסקרייפט קשורה לכך. לשפת ג'אוהסקרייפט 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 שלכם מכך. כשנכנסים אל דף כלשהו ברשות, ולא משנה אם מדובר באתר חדש או באתר משחקים או סרטים – רואים דף HTML. דף זה הוא בעצם דף טקסט. בדיקו כמה קוד ג'אוהסקרייפט שאפשר לכתוב ב-HTML, גם קוד HTML הוא קוד שאפשר לנתח בכל מקום. אם אתם רוצים לראות את קוד ה-HTML של כל דף, לחצו על הכפתור הימני בעכבר ואו על "הראה מקור" או "view source". תראו מיד המונח-המונח טקסט עם סימנים כאלה < וכאלו >. הדברים שנמצאים בתוך החצים נקראים "אלמנטים של HTML" ויכולם להכיל טקסט או אלמנטים אחרים. אלמנט בסיסי של HTML נראה כך:

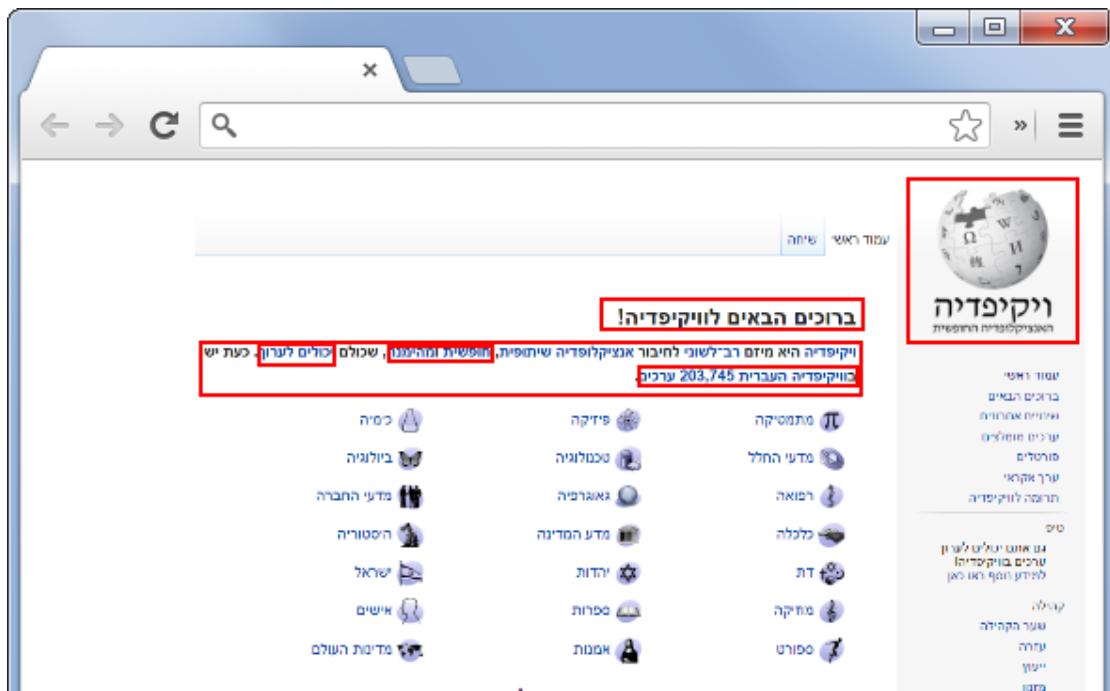
```
<elementName>Element Content</elementName>
```

אלמנט מורכב מתגיית פתיחה עם סוג האלמנט, תוכן האלמנט, במרקחה זה טקסט פשוט, ותגיית סגירה שמורכבת מלוכסן ושם האלמנט.

אם מדובר באלמנט שנקרה "אלמנט שנさんが 말いました", הוא נראה כך:

```
<elementName />
```

בגدول, כל דבר שראויים בדף אינטרנט כלשהו עשוי מאלמנטים של HTML. כל דבר. למשל:



כאן רואים את דף הבית של ויקיפדיה. כל אובייקט, קישור או טקסט בו הם אלמנטים בריבועים. למשל הלוגו של ויקיפדיה הוא אלמנט מסווג תמונה. הטקסט "ברוכים הבאים לוויקיפדיה!" הוא אלמנט מסווג `h1`, ככלומר כותרת ראשית. יש גם אלמנטים בתוך אלמנטים. טקסט הפתיחה הוא אלמנט ויש לו אלמנטים בנים של קישורים (סימנתי כמה 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 משנה, `p` הוא פסקה, `a` הוא קישור, `div` הוא אלמנט כללי וכך הלאה. יש עשרות סוגים של אלמנטים.

לכל אלמנט יכולות להיות תכונות (attributes). למשל, אם יש קישור, אחת התכונות של האלמנט היא המיקום שלו – הtekst שmailto: Amnon שmailto: Amnon מציין את העבר מעל הקישור. הינה דוגמה למה שרואים אם כותבים בתוך מסך 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 משהו בתנהגו.

יש אלמנטים שמכילים טקסט כמו קישור או פסקה:

```
<p>This is paragraph.</p>
```

ויש כאלה שלא, כמו תמונה. למשל:

```

```

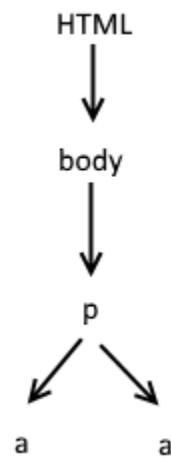
הבדל הוא שהאלמנטים ללא הטקסט נסגרים בלוכסן בסופם ולא בשם האלמנט. ככלומר ממש לא כך:

```
</img>
```

כאמור, יכולים להיות אלמנטים בתחום אלמנטים. הינה לדוגמה פסקה המכילה 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:



כלומר ה-HTML עצמו שנחשב אבא, תגיית ה-`body`, ואו 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'אוועהסקרייפט.

מוזהים של תגיות HTML

אלמנטים של HTML יכולים להיות מזוהים, משחו שבעזרתו אפשר לזהות את האלמנט לצרכים שונים. בדיק כמו בני אדם שיש להם שם פרטי ושם משפחה, גם לאלמנטים יש שני מזוהים. המזהה הראשון הוא כללי וקוראים לו `class`. לא להתבלבל עם קלאס של ג'אוועהסקרייפט. במקרה של HTML, `class` הוא מקביל לשם משפחה, ובבדיקה כי שיש כמה אנשים בעולם עם שם המשפחה שלי, 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 לאוthon אלמנט. המזהה השני נקרא `id` והוא מזהה ייחודי. אסור לתת את אותו `id` לשני אלמנטים באותו דף. ה-`id` הוא ייחודי ומוגדר לכל אלמנט, אפשר לתת `id` אחד בלבד לכל אלמנט.

אפשר כמובן להצמיד גם מזהה `class` וגם מזהה `id` לאלמנט אחד. במא משמשים? תלוי במה שרוצים לביצוע ובמבנה הדף. לאלמנטים מיוחדים שרוצים להגיע אליהם בקלות ג'אוועהסקרייפט נותנים `id` בלבד.

גישה אל תגיות באמצעות ג'אוועהסקרייפט

לתרגול, צרו דף HTML לדוגמה ותציבו בו ג'אוועהסקרייפט. הדף זהה זהה לדף שהסבירתי עליו בפרק על בניית סביבה העבודה, אבל חשוב להזכיר על דבר אחד – דפדף מרים את קוד הג'אוועהסקרייפט ומרנדר את האלמנטים בסדר מקבילי מלמעלה למטה. לצורך התרגול חשוב מאוד שקובצי הג'אוועהסקרייפט והקוד של הג'אוועהסקרייפט יהיו בתחום הדף, כיון שגם הקוד יחפש את האלמנטים של ה-HTML או יתיחס אליהם, הדפדף יהיה חייב להכיר

אותם לפני שהוא מרים את הקוד – ככלمر קובץ ה'אואהסקרייפט' שמתיחסים לאלמנטים חיבים להיות מתחת לאלמנטים. זו הסיבה שבגללה יש להקפיד, לפחות כרגע, שקוד ה'אואהסקרייפט' 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 שמוקף ב-HTML בתגיות <!--> נחשב להערה ולא מודפס, בדומה להעתת ג'אואהסקרייפט. היכן שיש הערת HTML – זה המקום שבו כתבים את האלמנטים. תגית הסקרייפט תכיל קישור לקובץ ה'אואהסקרייפט', ולפיכך היא חייבת להיות מתחת ל-HTML.

אלמנטים של HTML מתורגמים לאובייקטים של ג'אואהסקרייפט

כל אלמנט שיש ב-HTML יכול להיות מתורגם לאובייקט ג'אואהסקרייפטי כשר למהדרין. כאשר קוד ג'אואהסקרייפט חי בתחום HTML הוא יכול לגשת אל ה-HTML הזה באופן ישיר בלי צורך בטעינה. ה-HTML מתורגם לסוג של ישות שנקראת DOM, ראשי התיבות של Document Object Model. הישות הזאת נמצאת בזיכרון זומינה דרך קוד ה'אואהסקרייפט'. נשמע אבסטרקטי מדי? הנה נראה. אחרי שמצויבים את ה'ג'אואהסקרייפט' בתוך ה-HTML, יוצרים בתוכו אובייקט h1, שהוא בסגנון הזה:

```
<body>
  <!-- HTML elements will be here -->
  <h1 id="myHeader">Headline</h1>
</body>
```

בקובץ של ה'ג'אואהסקרייפט' מכנים את הקוד הבא:

```
let header = document.getElementById('myHeader');
header.innerHTML = 'Hello world';
```

אם טוענים את העמוד אפשר לראות שהטקסט שכותרת הוא לא "Headline" אלא "Hello world". איך עושים את זה? ראשית משתמש באובייקט ה'גלוובלי' document. האובייקט הזה קיים בכל ג'אואהסקרייפט שנמצאת בתוך HTML ומכיל את כל ה-DOM. הוא אובייקט כמו כל אובייקט אחר ויש לו מתחות.

אחד המתוודות היא `getElementById`, שמקבלת ארגומנט אחד. איזה ארגומנט? את ה-`id`. במקרה שלנו ה-`id` הוא הערך שהושם ב-`attribute id` של האלמנט `h1`. המתוודה מחזירה את הייצוג של האלמנט ב-DOM עם כל המתוודות שלו. ומעכשו? הגיגה שלמה. אחד המאפיינים הוא `innerHTML`, שמאפשר לשנות את תוכן האלמנט. אפשר גם ליצור אלמנטים חדשים לחולוטין ולהכניס אותם למסך:

```
let p = document.createElement('p');
p.innerHTML = 'I am a paragraph';
document.body.appendChild(p);
```

כאן למשל משתמשים במתודה `createElement` כדי ליצור אלמנט מסווג פסקה או תגית `<p></p>`, להכניס אליה תוכן ואו להציג אותה אל ה-`body` של המסך באמצעות `appendChild`. עד שימושים במתודה `appendChild`, הייצוג של האלמנט נמצא בזיכרון. עד שהוא לא מחובר ל-DOM אין לו ייצוג ויזואלי.

AIROURIM UM HTML VAG'AOHOSKRIPUT

אפשר גם להאזין לAIROURIM שמתרחשים בדף. AIROUR שאל העבר על אלמנט והמון דברים אחרים. בדוגמה כזו אתמקד באירוע של קлик. הנה ניצור ב-HTML כפטור שיש לו `id`:

```
<!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 `g'AOHOSKRIPUT` תופסם את אלמנט הCPFTR כרגע עם `document.getElementById` עם מכנים אותו למשנה:

```
let button = document.getElementById('myButton');
```

עכשו נרצה ש בכל פעם שיקליקו על הCPFTR יראו משהו בקונסולה. את זה עושים בעזרת AIROUR DOM. AIROUR DOM מופעל על ידי אינטראקציה עם המשתמש שלוחץ על הCPFTR, והטיפול באירוע מתבצע באמצעות הפונקציה `addEventListener` שגדירים אותה כמתפלת באירוע. מותוודת `addEventListerner` זמינה בכל אלמנט. היא

מקבלת שני ארגומנטים. הראשון הוא סוג האירוע, ובמקרה זה: `click`. השני הוא פונקציה שפועלת בכל פעם שיש קליק. משתמשים במקרה הזה בפונקציית `vez` בסיסית שכבר הופיעה בדוגמאות רבות קודם:

```
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 שבכל פעם שלוחצים על הכפתור, בקונסולה רואים "קליק". אבל למה להסתפק רק בקונסולה? אפשר ליצור אלמנט שכותב 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 זה קוד שאתם כבר מכירים, שיזכר אלמנט. הקליק של המשתמש הוא בעצם הקלט והפלט. הוא הרינדור של האלמנט. אפשר ליצור קלט משוכלל יותר באמצעות אלמנט ה-`input` שנראה HTML. האלמנט זה יוצר שדה טקסט והוא נראה כך:

```
<input id="myInput" />
```

אפשר לגשת אליו כמו כל אלמנט HTML. תופסם את האלמנט באמצעות `getElementById` וואז ניגשים אל תכונת `value` שלו:

```
let input = document.getElementById('myInput');
let content = input.value;
```

הצעד הבא יהיה לקחת את מה שיש ב-`value` 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);
});
```

אסביר 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 נכונים אחר כבוד למשתנים המתאים כדי שיתאפשר להתייחס אליהם. בשלב הבא מצמידים אירוע קליק לכפתור. האירוע עובד בכל פעם שיש להיזכר. מדובר בפונקציה טהורה, ככלומר צו שהוא ג'אווהסקרייפט בלבד, שבמקרה זהה יוצרת אלמנט מסווג פסקה. ב-HTML פסקה היא תגית `<p>`. היא לוקחת את מה שהמשתמש הכניס לתוכה שדה הטקסט, מכניסה אותו לתוכנת innerHTML של אלמנט הפסקה וממצמידה את אלמנט הפסקה אל גוף ה-HTML, וכך אפשר לראות אותו.

מה שחשוב להבין הוא שמדובר כאן בג'אווהסקרייפט לכל דבר אך בשילוב ה-DOM – כלומר אובייקטים ויריעים שימושיים מהאלמנטים שיש ב-HTML. ברגע שימושים ב-`document` כדי ליצור אלמנט או לקרוא לו, הוא מקבל גם תוכנות כמו innerHTML ואפשר להציג אלייו איריעים מיוחדים כמו "קליק" באמצעות `.addEventListener`.

הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווהסקרייפט

בכל הדוגמאות עד כה הריתי איך מצמידים את האלמנט שיצרנו ישירות ל-`document.body`, שהוא בעצם תגית ה-`<body>` שיש ב-HTML. זו תגית יהודית שמופיעה רק פעם אחת במסמך HTML. אפשר להציג אלמנטים לכל אלמנט אחר!

أدגים באמצעות תגיות `` ו-``. מדובר בתוצאות של רשימה ב-HTML, שנראית כך:

```
<ul>
  <li>פריט ראשון</li>
  <li>פריט שני</li>
  <li>פריט שלישי</li>
</ul>
```

הבה ניצור דף אינטרנט שבו רשימת קניות שהמשתמש יכול להכניס אליה פריטים. ה-HTML כולל חלון של שדה טקסט, כדי שהמשתמש יוכל להכניס פריטים, כפטור של "הכנס" וכמו כן שלד 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'אוהסקריפט** יהיה פשוט. בדוק כמו בדוגמה הקודמת, מגדירים את שלושת השחקנים במשחק: הcptnor שmaps את כל הפונקציה של ההוספה, שדה הטקסט שבו משתמש מכניס את מה שהוא רוצה להכנס לרשימה והרשימה הריקה.

```

const button = document.getElementById('myButton');
const input = document.getElementById('myInput');
const list = document.getElementById('myList');

```

ועכשיו להצמדת האירוע לכפתור. את זה עושים באמצעות המתודה `addEventListener`. מעבירים שני ארגומנטים. הראשון הוא סוג האירוע (קליק), והשני הוא ארגומנט של הפונקציה שתעבד כשהארוגמנט יפעל. מעבירים אותה כפונקציית חץ:

```

button.addEventListener('click', (event) => {
});

```

הקוד שבתוֹך הפונקציה הוא פשוט. יוצרים אלמנט חדש מסוג `li`, מאכלסים את ה-`innerHTML` שלו בערך המתබל משדה הטקסט – `value`, ואז מצרפים 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 הזה אינו מלמד CSS או HTML. אבל קל ללמידה CSS, ובഗודל אפשר לשנות בעורתו על העיצוב ועל המיקום של כל אלמנט HTML. לכל תכונת CSS יש ערך שלה. למשל, אם רוצים לקבוע את רוחב האלמנט, התכונה היא `width` והערך הוא (למשל) `10px`, שמשמעותו רוחב של 10 פיקסלים. תכונת `background-color` קובעת את צבע הרקע, והערך יכול להיות צבע (כמו `red`) או ערך הדצימלי של צבע כמו `#ff0000`.

על מנת לשנות תכונת CSS או ליצור תכונה חדשה משתמשים בתכונת `style`, שקיימת בכל אלמנט DOM. לשם ההדגמה אוצר אלמנט HTML מסוים `div` עם `id` מסוים ב-HTML. אלמנטים מסוג `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 הדפסן לא תראו כלום. הנה נניס קצת צבע לאלמנט זהה באמצעות ג'אוועסקריפט:

```

const div = document.getElementById('myDivId');
div.style.height = '100px';
div.style.width = '100px';
div.style.backgroundColor = 'red';

```

יצרים תכונות של גובה, של רוחב ושל צבע רקע על מנת ליצור ריבוע אדום. זה ממש נחמד. אפשר להכניס, לשנות או לקרוא כל תכונה של style באופן חופשי כמוון. שימו לב שמדובר בהשפעה על תגיית ה-style של האלמנט ולא בשינוי קובץ CSS הקשור למסמך. לרוב לא מקובל לשנות צבעים באופן ישיר בג'אוועסקריפט, אלא ליצור class בשיינו קובץ CSS שמכיל את התכונות המבוקשות ואו להוסיף את הקלאסים הללו לאלמנטים או להוריד אותם באמצעות ג'אוועסקריפט.

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');

```

לעתיל נשמע לכם ג'יבריש – זה הזמן ללמידה CSS או HTML או CSS בספר זה. כאמור, אני לא מכשה HTML או CSS בספר זה.

סלקטורים של DOM

סלקטור הוא שם כללי למתודות שモוצאות אלמנט. עד כה השתמשנו ב-`document.getElementById`, `document.querySelector` ו-`document.querySelectorAll` שמהם. אך יש סלקטורים אחרים שאינם מבוססים על `id`. הסלקטור של `id` הוא ייחודי כי יוצאים מנקודת הנחה שתמיד יש אלמנט אחד שיש לו את ה-`id` זהה. במקרים שיש סלקטורים המבוססים על `class` או על תכונה אחרת, יתאפשר מערך של תוכאות שאפשר להתייחס אליהם כלם יחד, אלא שבמערך הזה יהיו אלמנטים של DOM.

הבה נדגים באמצעות הסלקטור `getElementsByClassName`. הסלקטור הזה מחזיר מערך של כל האלמנטים שיש להם `class` מסוים. קיים ה-HTML הזה:

```
<div class="rectangle">Rec 1</div>
<div class="rectangle">Rec 2</div>
```

(שימוש לב שלשם הנוחות אני לא מציב כאן את ה-HTML המלא כולל ה-`body` וה קישור לקובץ `glossary.js`).

על מנת לבחור את כל האלמנטים שיש להם את ה-`class` הזה ולצבעו 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 כmo על כל מערך רגיל בג'אווהסקריפט, במקרה הזה באמצעות `for of`, אבל אפשר להשתמש בכל לולאה שהיא. למדנו על לולאת `for of` בפרק על לולאות, אז אחות אמרורים להבין מה קורה פה. עוביים על כל אייר בלולאה ונותנים לו גובה, רוחב וצבע. אם רוצים לחתת 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 שף 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-דף. לשמורצים ג'אווהסקריפט בסביבת

דף – גם ה-DOM וגם היג'אזהסקריפט חיים בתוך הדף והו מנהל את עצם ה-DOM ואת המנווע של ה-JS ודווגע לחבר בין השניים. למעשה, מפני שאתם כבר מכירם את היסודות התיאורטיים של השפה, אתם אמורים לשלוט גם בג'אזהסקריפט בסביבת דף. מדובר באותו שפת ג'אזהסקריפט בדיק. אפשר להعبر כארגומנט רשיימה של קלאסים עם הפרש של רווח ביניהם.

אפשר גם לבחור אלמנטים לפי שם התגית. למשל, הקוד הזה בוחר את כל התגיות שהן מסוג div וצובע אותן באדום. העיקרון הוא אותו עיקנון 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 `querySelectorAll`, שבוחר את האלמנטים לפי סלקטורים של CSS. כאן נדרשת הבנה ב-CSS אבל כל סלקטור מורכב של CSS יכול להיות שימושי גם בג'אזהסקריפט. 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` שלו הוא `rectangle` והם ילדים של `div` עם `class` `good_parent`, אפשר להשתמש בסלקטור של CSS שהוא:

`.good_parent div.rectangle`

אם המילים "סלקטור של CSS" נשמעות לכם כמו סיינט, סימן שאתם צריכים לחזור על ה-CSS שלהם. הספר הזה אינו מכסה HTML ו-CSS רק:

```
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 שambilגינים שיש סלקטורים שמחזירים מערך של אלמנט DOM במקומם אלמנט DOM אחד, ושהמערך הזה הוא מערך ג'או-הסקרייפטי לכל דבר ועניין אלא שהוא מכיל אלמנט DOM – נגמר הסיפור.

ל-selectorn יש גם גרסה של פונקציה בשם querySelector שמחזירה רק את התוצאה הראשונה של הселקטור ולא מערך שלם. 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 להלוטין ל:

```
const div = document.querySelectorAll('.good_parent div.rectangle');
//[div.rectangle, div.rectangle]
const firstDiv = div[0];
firstDiv.style.backgroundColor = 'red';
firstDiv.style.height = '100px';
firstDiv.style.width = '100px';
```

לשם הנוחות אצף רשימה של סלקטורים נפוצים ותפקידם:

תפקיך	שם הSELKATOR
בוחר אלמנט אחד ויחיד לפי ה-ID מחזיר אלמנט אחד	document.getElementById
בוחר כמה אלמנטים לפי קלאס אחד או יותר (הקלאסים מועברים כמח祖ות טקסט עם רווח) מחזיר מערך של אלמנטים	document.getElementsByClassName
בוחר כמה אלמנטים לפי שם האלמנט - סימן לכל האלמנטים ב-OM 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 להכניס כמה סלקטורים עם פסיק ביןיהם מחזיר מערך של אלמנטים	document.querySelectorAll
בוחר את האלמנט הראשון שמציה לסלקטור של CSS-ה-אלמנט אחד	document.querySelector

אירועים נוספים

עד כה למדנו רק על אירוע קליק, אך יש אירועים 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 אמצעים 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 יעלה עליו. את זה עושים באמצעות הצמדת אירוע בשםmouseover. 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';
});
```

addEventListener מקבל שני ארגומנטים. הראשון הוא שם האירוע והשני הוא הפונקציה שעובדת כשהאירוע מופעל. משתמשים בפונקציית `vez`, 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 והסביר. במקרים אפקטיבים כאלה נעשים באמצעות CSS בלבד.

אני רוצה להתעכ卜 קצת על הארגומנט שפונקציית החז מקבלת, והוא ה-`event`. ככל-mdeno על פונקציות חז ועל קולבקים בפרק על הפונקציות, הסבירתי שככל קולבק יכול לקבל כמה ארגומנטים מי שמבצע אותו מעביר. במקרה הזה הארגומנט שמקבלים הוא אובייקט האירוע עצמו, שמכיל عشرות פרטיים. למשל, `Alt`, `Shift`, `Ctrl` או `Ctrl` היו לחיצים, מה המיקום המדויק של האירוע, באיזו שעה הוא התקיים וכו' וכו'. כך למשל אפשר להפעיל את אירוע הקליק רק אם הCapsLock נלחץ 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 בשלוב כפתור Alt להזנתה לצבע צהוב. אובייקט ה-event (אפשר לקרוא לו בכל שם כMOVEN) הוא אובייקט כמו כל אובייקט. אפשר לעשות עליו console.log כדי לתחות על קנקנו ואפילו מומלץ להביע בו. אפשר לעשות אותו דברים מעניינים מאוד.

אפשר להפעיל כמה אירועים על אותו אובייקט. כאן למשל גורמים לריבוע להפוך לצהוב כשהעכבר נמצא עליו ולהיות שוב אדום כאשר הוא ממשיך הלאה. האירוע הראשון שימושים בו הוא mouseenter, שמופעל ברגע שהעכבר נכנס אל האלמנט, והאירוע השני שימושים בו הוא mouseleave, שמופעל ברגע שהעכבר יוצא 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';
});
```

שים לב שיש משמעות לאירועים. למה לא השתמשתי ב-mouseover? כיון שהוא מופעל בכל פעם שהעכבר נמצא מעל האלמנט, ככלומר המונ פעים – בכל פעם שהעכבר זו והпозה שלו היא מעל האלמנט. האירוע mouseenter מופעל רק פעם אחת – כשהעכבר נכנס לתוך האלמנט. זה הכל.

יש המונ אירועים, מירועי מקלדת ועד אירועי עכבר. יש גם אירועים של אלמנטים (למשל אירוע שמוופעל אם מישו מונה את ה-CSS של האלמנט) ואירועים הקשורים ל-clipboard. הינה רשימה של כמה אירועים עיקריים:

הסבר	אירוע
אירוע של הקלקה באמצעות העכבר	click
סמן העכבר נכנס אל תחום האלמנט	mouseenter
סמן העכבר יוצא מתחום האלמנט	mouseleave
המשתמש לוחץ על כפתור העכבר בתחום האלמנט	mousedown
המשתמש שחרר את כפתור העכבר בתחום האלמנט	mouseup
המשתמש מגלגל את גלגלת העכבר כאשר סמן נמצא על האלמנט	wheel
לחיצה על כפתור במקלדת	keydown
שחרור של כפתור במקלדת	keyup

המשתמש מבצע פוקוס על האלמנט (למשל באמצעות הטאב או העכבר)	focus
המשתמש יוצא מפוקוס על האלמנט (למשל באמצעות הטאב או הקלקה על אלמנט אחר)	blur

פעוף של אירועים

פעוף (באנגלית bubbling) הוא שם של תופעה שחווב להכיר באירועים, והוא מתרחשת כאשר יש יותר מכמה אלמנטים. למשל כמו בדוגמה זו:

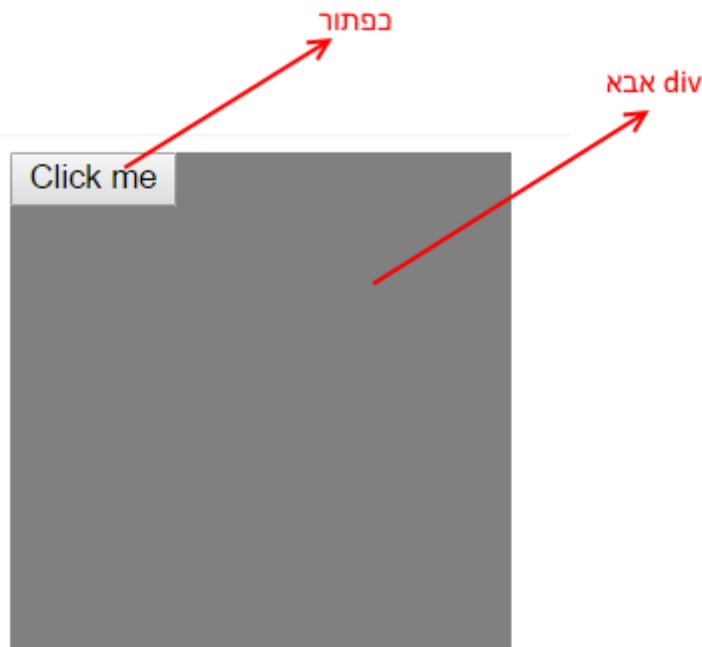
```
<div id="parent">
  <button id="button">Click me</button>
</div>
<div id="result"></div>
```

כאן יש שלושה אלמנטים. אלמנט div אחד עם id של parent. מתחתיו יש שבתוכו יש כפתור עם id של button. מתחתיתו יש div שכותוב בו .result.

קוד הבא רץ באותו דף. מקבלים את שלושת האלמנטים כאלמנטים של DOM באמצעות getElementById בהתאם ל-id של כל אחד מהם, צובעים את ה-parent באפור ונותנים לו מדימויים:

```
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 כזה:



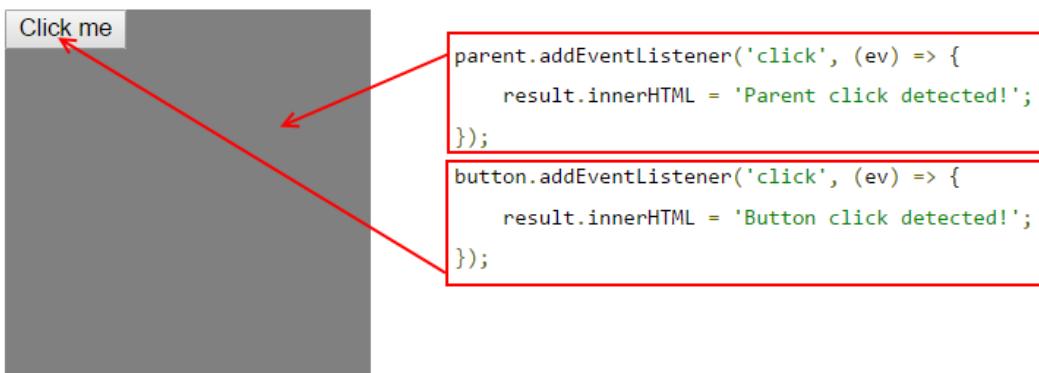
מה קורה אם מctriddim לאלמנט האב אוiroう? תזוכרת: אלמנט האב הוא ה-`div` שמכיל את הכפתור. בדוגמה זו לאלמנט האב יש `id` בשם `parent` (דיברנו על אלמנטים אבות וילדים מוקדם יותר בפרק).

```
parent.addEventListener('click', (event) => {
    result.innerHTML = 'Parent click detected!';
});
```

זה קל – אם לוחצים על האב, רואים שבדף התוצאה כתוב: Parent click detected. מה יקרה אם לוחצים על הכפתור? אותו הדבר! למה? כי האירוע מפעפע גם לאלמנטים של הילדים. האירוע מתחילה בכפתור ומפעפע למעלה עד שיש אלמנט שמתפל בו (ובמקרה זה האב). כמובן, אם אני אלמנט ויש אירוע על האלמנט שמכיל אותו (או על האלמנט שמכיל את האלמנט שמכיל אותו וכו'), אז אירוע יפעע כלפי מעלה עד שיופיע באירוע כלשהו (או עד שיגיע ל-`body`). לוחצים על הכפתור, אין בו אירוע להציגו. האירוע עולה למעלה. האם באלמנט האב יש אירוע? כן! אז אירוע מופעל ממשיך לעלות למעלה, והוא מפעיל את כל האירועים באלמנטים האב השוניים.

זה נחמד ו שימושי, אבל העסק 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 יקרה כשהלחצו על הכפתור? האירוע של הכפתור יתרחש, וכן ב-`div` של התוצאה יודפס `Parent click detected`, אבל מיד אחר כך, האירוע יפעע למעלה ויפעל את האירוע של `parent`, שבתורו ידפיס מיד ב-`div` של התוצאה `Button click detected`. לעומת זאת האינדייקציה של הלחיצה על הכפתור ב-`div` ה-`parent` לא תראות את האינדייקציה של הלחיצה על הכפתור ב-`div` ה-`button`. זה קצת בעייתי אם אנחנו רוצים לדאות בליחיצה על הכפתור `Button click detected`, ורק כשלוחצים על ה-`div` של האב לקבל `Parent click detected`.



מה יקרה אם לא רוצים שהאירוע יפעע למעלה? באירוע שבו רוצים לעצור את הפעוע חיבים להפעיל את:

```
event.stopPropagation();
```

שזמין 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 בדיקת מה שרוצים שהוא יעשה. לחיצה על האב תדפיס שלחתם על האב. לחיצה על הcptor תפעיל את האירוע של cptor, אך האירוע לא יחלחל להלאה והאירוע של האב לא יופעל.

הצמדת אירועי ג'אוועסקריפט לאלמנטים ב-HTML

השיטה של `addEventListener` נחשבת לעדיפה בהרבה, אבל אפשר גם להציגם אירועים של ג'אוועסקריפט ב-HTML באמצעות תכונת `onclick` לקליקים. למשל משזה:

```

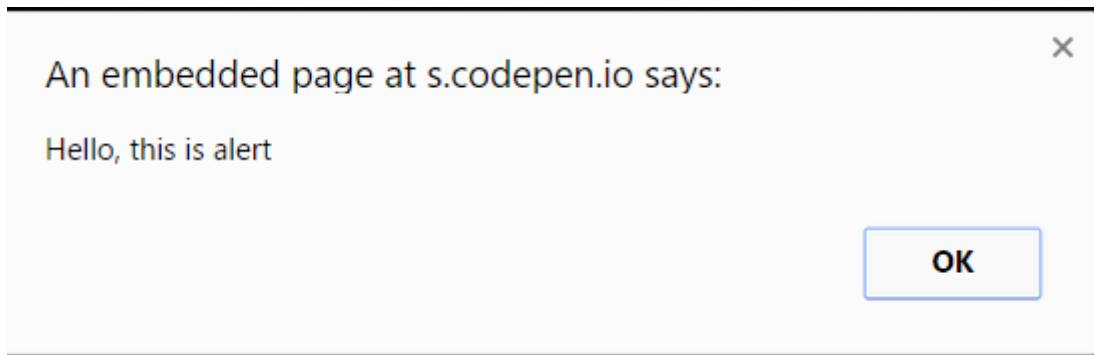
```

יצירת alert ו-prompt

עוד דרך מושנת לתקשר עם משתמשים היא באמצעות alert. מדובר בפונקציה גלובלית שזמין בכל ג'אווהסקריפט שהיה בסביבת דף. משתמשים בה כך:

```
alert('Hello, this is alert');
```

מה שהמשתמש יראה הוא חלון לא גרפי שיקפוֹץ עם הודעה.

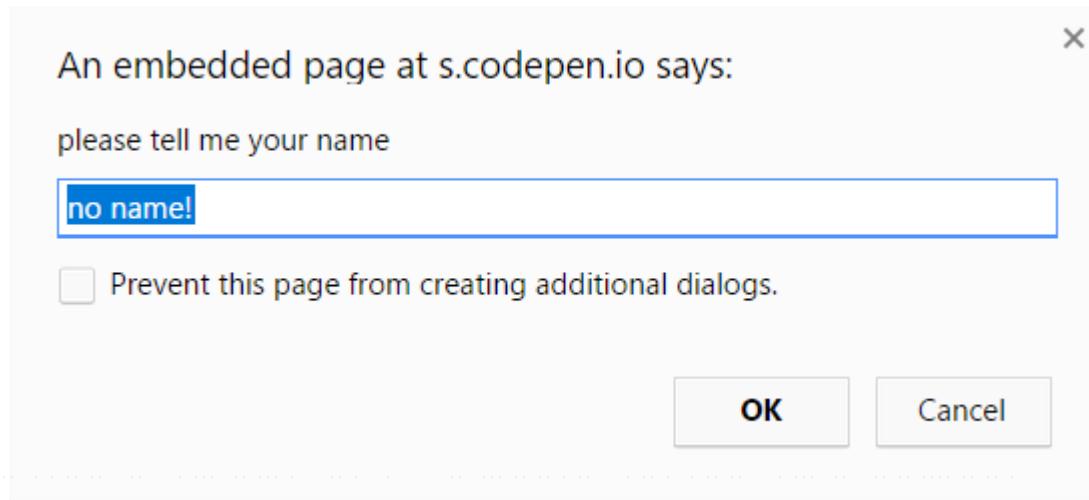


עיצוב החלון תלוי בדף, בעיקר מפני שהוא בחלון של מערכת הפעלה. כאמור, מדובר בדרך מושנת להציג הודעה למשתמש וכיום ממעטם להשתמש בה. מומלץ מאוד להציג למשתמש הודעות בדרך אחרת, כפי שהראיתי קודם – הקצתה div מיוחד והדפסת הודעה עליו.

דרך נוספת לתקשר עם המשתמש הפונקציה הגלובלית prompt, שגם היא זמין בכל ג'אווהסקריפט שהיה בסביבת דף. הפונקציה זו מקבלת שני ארגומנטים. הארגומנט הראשון מכיל את המחרוזת למשתמש כשם פעילים אותה, מוצגת למשתמש המחרוזת שהוכנסה בארגומנט הראשון עם שדה טקסט שבו הוא יכול להכניס מחרוזות טקסט משלו, ואיתה פונקציית ה-prompt מוחזירה. הארגומנט השני הוא ערך ברירת המחדל שモצג למשתמש.

```
const result = prompt('please tell me your name', 'no name!');  
alert(result);
```

הקוד הזה למשל יציג את השאלה:



אם המשתמש יכנס שם, מיד אחר כך יוקפץ alert עם השם שהוא הכניס. גם כאן מדובר בדרכן מינשנת מאד ועדייף לא להשתמש בה. אם רוצhim לתקשר עם המשתמש, עושים זאת ב-`input`. 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, שהיה בחתית 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 דג'אווהסקריפט מכניסים את חקודה הבאה:

```
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 למשנה `myObj`. מעכשו יש אלמנט `DOM` שיש לו מתחודות משלו, וכל אחת מהן משפיעה עליו. אחת החשובות שבהן היא `innerHTML`, שמכניסה את התוכן לאלמנט שנוצר. כל מה שנותר לעשות אחרי שיצרים את האלמנט ואת התוכן שלו הוא להכניס את האלמנט לגוף ה-`HTML` באמצעות `.appendChild`.

תרגילים:

צרו דף אינטרנט שבו שדה טקסט וcptor. אם המשתמש מכניס מספר לשדה הטקסט ולחוץ על הכפתור, הדף מציג רשימה ממוספרת 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 להכנס מספר, רשימה ריקה וכפטור להפעלה הפונקציה שתיקח את המספר ותכנס פרטיים לרשימה. ב-HTML יהיה קישור לקובץ ג'אוועסקריפט שבו יהיה הקוד. אלו שלושת השחקנים.

בקוד הג'אוועסקריפט מגדירים את שלושת השחקנים באמצעות `document.getElementById`. מרגע זה מקבלים גישה לתוכנות שלהם ב-DOM.

```
const button = document.getElementById('myButton');  
const number = document.getElementById('myNumber');  
const list = document.getElementById('myList');
```

מצמידים אירוע קליק שמקבל שני ארגומנטים באמצעות `addEventListener`. הראשון הוא שם האירוע והשני הוא מה שקורה כשהוא מופעל. שימו לב שהשתמשתי כאן בפונקציית חץ:

```
button.addEventListener('click', (event) => {  
});
```

בפונקציה עצמה מגדירים את המספר שמקבלים מהמשתמש:

```
const numberValue = number.value;
```

ברגע שיש מספר, מפעילים לו לאת `for` שתפקידו לפיו:

```
for (let i = 0; i < numberValue; i++) {  
}
```

בתוך הלולאה יוצרים אלמנט `li` (`li` הוא אלמנט של איביט ברשימה) ומכניסים אותו לרשימה. מאכליים את תוכנו באמצעות `+i`. זה הכלול.

תרגום:

צרו כפטור ואלמנט מסוג `div`. להיזה על הכפטור תהפוך את האלמנט לצהוב, ברוחב של 100 פיקסלים ובגובה של 100 פיקסלים.

פתרון והסבר:

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 כפתור והשני הוא div. מקפידים לקשר בין ה-HTML לבין ה-ג'אווהסקריפט.

בג'אווהסקריפט מגדירים את שני השחקנים הראשיים – הכפתור וה-div. עושים את זה באמצעות getElementById. לאחר מכן מזמנים אל הכפתור אירוע מסוג קליק, שモפעל בכל פעם שלוחצים על הכפתור. באירוע עצמו משתמשים ב-`style`, שזמין לאלמנטים של DOM על מנת לקבוע את הגובה, את הרוחב ואת הצבע.

תרגילים:

בஹמך לתרגיל הקיים, כתבו קוד שיגרום לכך שללחיצה על הכפתור תקבע את האלמנט בצהוב, להיזכה נספח תצבע אותו באדום, להיזכה נספח תצבע אותושוב 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>

```

```
</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 בתרגיל הקודם. שני השחקנים העיקריים הם div ו-button. בקוד ה-ג'אווהסקריפט מגדירים את שני השחקנים כאלמנטים של DOM באמצעות getElementById. מגדירים משתנה בוליאני בשם activated ומגדירים אותו כ-true. בסופי כל מתחלים את הריבוע הצהוב. קובעים את הרוחב, את הגובה ואת הצבע ההתחלתיים. מצלמים אירוע קליק לכפתור באמצעות addEventListener, שכבר הכרנו בתרגילים קודמים. הפונקציה ש-addEventListener מפעילה בכל קליק מרכיבת מעט. ראשית בודקים אם activated הוא true. אם כן, צובעים את האלמנט בצהוב ומשנים את ה-activated ל-false. אם activated הוא false, הוא נצבע באדום ומשנים את ה-activated ל-true.

ליבאגיניג

בכל כלי מפתחים שהוא – בין שמדובר בכרום ובין שבפיירפוקס או באgel – יש אפשרות להפעיל כל שנקרא "דיבאג". כל מפתחים הוא רכיב תוכנה שנמצא בכל דפדפן ללא צורך בהורדה או בהפעלה. מדובר בשם כללי לכל שימוש למפתחים לבחון את הקוד שלהם. בדף הכללי זה עוזר לנו לתרור תקלות בג'אווהסקריפט ולהבין מה קורה בנבכי הקוד. עד כה השתמשנו בעיקר console.log על מנת להדפיס דברים בקונסוללה. הכללי הזה חוסך את התהליך של הדפסת פלט בקונסוללה ומאפשר לראות בדיקות בג'אווהסקריפט.

בדי לתרגל שימוש בדיבאגר, נניח שבפרויקט שלכם יש קובץ HTML סטנדרטי וקובץ ג'אווהסקרייפט שמקושר אליו, בדיק כmo שהסברתי בפרק על התקנת סביבת העבודה. קובץ ה-HTML נראה כך:

```
<!doctype html>
<html>

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

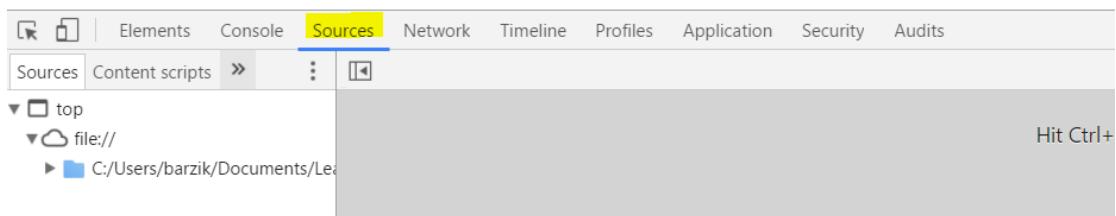
<body>
  <script src="./source.js"></script>
</body>

</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 בדף על מנת לוודא תקינות. אם הכל נכון, הבה נפתח את הדיבאגר. בכרום, לחצו על F12 אם אתם עובדים בחלונות או על i + Cmd + Shift + F12 אם אתם במק. עכשו מצאו את לשונית ה-sources



מצד שמאל אפשר לראות את כל עץ הפרויקט. פותחים אותו ומחפשים את קובץ הג'אווהסקרייפט source.js. ברגע שלוחצים עליו אפשר לראות את כל הקוד מצד ימין.

```

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 זה העץ מצומצם מאוד כיון שהוא כולל תקיה אחת בלבד. אם העץ מורכב, אפשר ללחוץ על p (או p במק) ולהקליד את שם הקובץ.

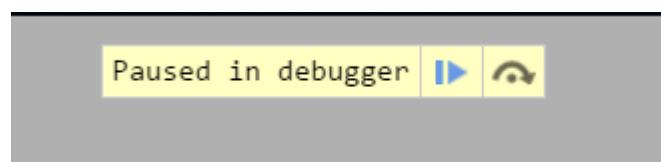
הדייבאגר עובד באופן עקרוני עם נקודות עצירה (breakpoints). אפשר לבחור נקודה (או כמה נקודות) בסקריפט שבה הוא יעצור ויאפשר לבחון את מה שקרה. הנה נראה! נלחץ על השורה השלישי, ממש היכן שנמצאת לולאה for. מיד יופיע סימן כחול המאשר שנקודות העצירה קיימות:

```

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

```

עכשו חיבים להפעיל את הסקריפט מחדש. את זה עושים על ידי טעינה של הדף שוב. שימוש לב שבהרבה מקרים לא צריך להפעיל את הסקריפט מחדש. אם מציבים את נקודות העצירה באירוע שמוופעל בלחיצה על קליק, אין צורך בהפעלה מחדש. כאן מדובר במקרה נדיר כיון שהסקריפט הזה כבר רץ ולא יירוץ אם לא נטען את הדף מחדש. טעינה מחדש של הדף לא תציג שוב את התוכן המקורי, אלא יופיע לפתע דף אפור ובו אינדייקציה שהדייבאגר מופעל:



בדיבאגר עצמו תוכלו לראות שעצרתם בנקודה העצירה:

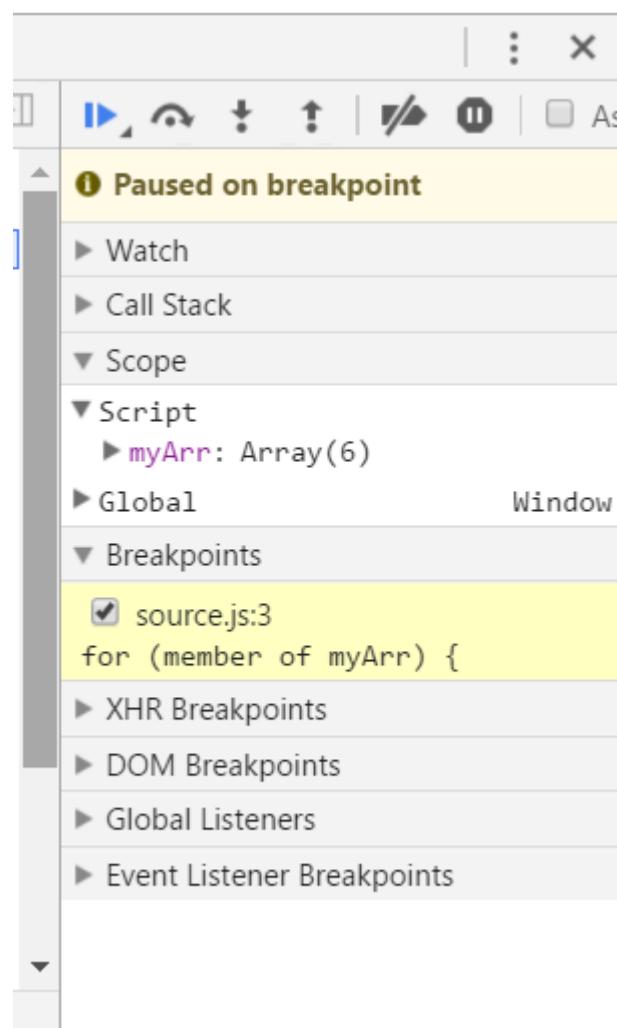
The screenshot shows the Chrome DevTools Sources tab. The left sidebar lists files: 'top', 'file://', and 'C:/Users/barzik/Documents/Le...'. Under 'C:/Users/barzik/Documents/Le...', there are 'index.html' and 'source.js'. The right panel displays the contents of 'source.js' with line numbers 1 through 8. Line 3 is highlighted with a blue background, indicating where the script is currently paused.

```

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

```

מצד ימין אפשר לראות פרטים על המצב הנוכחי:



ב-`scope` אפשר לראות את כל המשתנים הקיימים באותו `scope`, ובמקרה זה רק `myArr`. אפשר גם לראות את כל האלמנטים הגלובליים (כמו `document` שלמדנו עליו בפרק על HTML וג'אווהסקריפט) ואובייקטים מוכנים אחרים שעוד נלמד עליהם.

אם עוברים לكونסולה, אפשר להקליד פקודות שיעבדו בנקודת הזמן זו של הסקריפט.

על דיווח המצביע שורה קודים שבם אפשר להפעיל את הדיבאגר:



בלחץ הראשון אפשר להמשיך להריצין את הסקריפט עד נקודת העצירה הבאה, אם יש כזו. אם אין כזו, הסקריפט ירוץ עד הסיום. בלחץ השני אפשר לעשות over step – קלומר להריצין את הסקריפט לשורה הבאה. הלחץ השלישי והרביעי, stepin\out, שמוראים לכניסה אל פונקציות ולא נדון בהם בשלב זה. אפשר גם למחוק את כל נקודות העצירה.

הבה נלחץ על over step. תוכלו לראות שהתקדמתם צעד אחד מעבר לנקודת העצירה וכרגע אתם בתחום הוללה:

```
const myArr = ['a', 'b', 'c', 'd', 'e', 'f'];
for (member of myArr) {
  const el = document.createElement('p');
  el.innerHTML = member;
  document.body.appendChild(el);
}
```

Debugger paused

- Watch
- Call Stack
- Scope
- Block
 - el: undefined
- Script
 - myArr: Array(6)
- Global

Breakpoints

- source.js:3

YHR Breakpoints

עכשו יש בסkop שני חלקיים – גם זה של הבלוק, קלומר של ה-for of. אפשר לראות שיש את ה-el, אבל הוא עדיין לא מוגדר. הוא יהיה מוגדר רק בשורה הבאה. בוואנו נתקדם אליה באמצעות לחיצה נוספת על הפקד over step או על F10:

The screenshot shows the Chrome DevTools debugger interface. The top navigation bar includes tabs for Sources, Network, Timeline, Profiles, Application, Security, and Audits. The Sources tab is active, displaying the file 'source.js' with the following code:

```
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
```

The line of code at **line 5**, `el.innerHTML = member;`, is highlighted with a blue selection bar, indicating it is currently being executed or has just been executed. To the right of the code editor is the debugger sidebar, which displays the following information:

- Debugger paused**
- Watch
- Call Stack
- Scope
- Block
 - el: p
- Script
 - myArr: Array(6)
- Global
- Breakpoints
 - source.js:3

כעת אפשר לראות שה-`el` הוא אלמנט מסווג `c`. אם לוחצים על החץ הקטן שלצד ה-`el` אפשר לראות את האובייקט של ה-DOM במלוא תפארתו, לצד כל התכונות המיעילות של ה-DOM והתכונות שבאות ייחד עם הגדרתו כאובייקט, וזה המונ.

בגadol, איןפה יותר מדי, אך מדובר באחד הכלים החשובים ביותר למתכנתים וכדי מאוד ללמידה להשתמש בו – ואז להשתמש בו – כבר בהתחלה. היכולת המריהיבה של הדיבאגר לסייע במציאת תקלות היא חשובה מאוד, והוא גם נותן תמונה מלאה על מה שתרחש בקוד – כולל המשתנים המופיעים בסkop או בקלוז'רים השונים (ואם שכחتم מה זה, חזרו לפרק על הפונקציות).

שים לו: מطبع הדברים עיצוב של דפדף משנתה כל הזמן וייתכן שצילומי המסך המופיעים בה לא מעודכנים מספיק. אם כן, לא צריך להילחן – העיקרון הוא אותו עיקרון גם אם לשונית קוראים בשם אחר או אם האיקון נראה קצת אחרת.

אובייקטים גלובליים ואובייקטים מוגדרים

הראיתי בכמה הzdמניות שקיים אובייקטים שיש גישה אליהם. הדוגמה הטובה ביותר היא `document`. על אף ש-`document` היא לא מילה שומרה כמו `if` או `function`, יש גישה אל אובייקט ה-`document` המכיל מתודות ותכונות שונות. יש גישה אל `console`, למשל, על פונקציית הלוג שלו. מדובר באובייקטים גלובליים הזמינים לג'אוהסקריפט. ב"אובייקטים גלובליים" אני מתכוון לאובייקטים שמוגדרים ברמה הגלובלית ומשם מחללים 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);
  }
}();
```

בדוגמת הקוד הזה, למשל, `console` הוא אובייקט שזמין מיד ברמה הגלובלית, בסקופ של הפונקציה האנונימית המריצה את עצמה (שלמדנו עליה בפרק על הפונקציות) וגם בסקופ של לולאת `for`. אם דורסים את `console` בرمאה 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 שומרה אלא באובייקט גלובלי, אפשר לעשות את זה בקלות. זה אומר שככל `console.log` בסקופ של הפונקציה האנונימית וככל ה-`log` שהוצבו באותו סקופ באמצעות closure (שגם עליו למדנו בפרק על הפונקציות), לא יעבדו כיוון שפונקציית `log` תהיה ריקה ב-`console`. כמובן, הטעסט שנעביר ב-`log` פשוט לא יעבוד. כל ההameda הארוכה זו נועדה להסביר שיש הבדל משמעותי בין מילה שומרה, שהיא עצם עצמותיה של ג'אוהסקריפט, לבין אובייקטים גלובליים שהם מכובדים וחוובים, אך עדין רק אובייקטים שמצויים לכללי השפה. יש אובייקטים גלובליים שזמינים אך ורק לג'אוהסקריפט שעבוד בסביבת HTML, כמו `document` שהכרנו קודם, ויש אובייקטים גלובליים שזמינים אך ורק לג'אוהסקריפט שעבוד בסביבת שרת, כמו `window`, `Screen`, `URL` process.

למשל. אובייקטים גלובליים רבים זמינים לכל ג'אווסקריפט שהוא, כמו console. כל האובייקטים האלה נקראים Web APIs.

לכל אובייקט גלובלי יש תכונות או מethods. על מנת לבדוק אותן, אפשר לחפש אותן בתיעוד ברשות או פשוט להסתכל בקונסולה. בדוגמה הבאה פותחים את כל המפתחים בכרום, עוברים לكونסולה, מקלידים console ולחיצים על Enter. בבת אחת רואים את כל המתודות שיש לكونסולה.

```
> 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-Web API מאפשר לקבל המון מידע מהטכננים השונים. אחד האובייקטים השימושיים ביותר הוא אובייקט window, שיש לו תכונות ומethods רבות. למשל window.location, שמחזיר את ה-URL,(Clomar הכתובת, של הדף שבו הסקריפט רץ. כך אפשר להעביר את המשתמש למקום אחר. הקוד הזה, לדוגמה, מעביר את המשתמש לאתר גוגל.

```
window.location = 'https://google.co.il';
```

יש המון אובייקטים גלובליים כאלה, וצריך רק לזכור מהם לא מגיעים ממש ממקום אלא מוחרים לסקופ הגלובלי על ידי ג'אווסקריפט עצמה. מי שרצח לראות את רשימת האובייקטים המלאה של Web APIs מזמין לגשת לאתר המקיים והטוב ביותר של תיעוד ג'אווסקריפט: Mozilla Developer Network (MDN), שם יש פירוט מكيف יותר על כל האובייקטים. עם כל שינוי והתעדמות של הדפכנים נוספים אובייקטים. למשל, בשנת 2015 נוספו אובייקטים גלובליים שמאפשרים תקשורת באמצעות ג'אווסקריפט עם מצלמת הרשת, וכך כל קוד ג'אווסקריפט

יכול לגשת אל מצלמת הרשת, לאחר בקשת רשות מהמשתמש, ולהשתמש בפideal המציג ממנה למטרות שונות.

הרישימה, כאמור, נמצאת באתר MDN: <https://developer.mozilla.org/docs/Web/API>

בדומה לאובייקטים גלובליים, יש בג'אווהסקריפט גם אובייקטים מובנים. ההבדל הוא שכן האובייקטים לא מוזרקים מהדף או מסביבת העובודה אלא מוגדרים בתוכניות של השפה עצמה, ככלומר הם חלק מהשפה. נשמעו אולי שמדובר בהבדל דק, אך הוא ממשוני מאד. אובייקט document מגיע מהדף. הוא לא חלק رسمي מהשפה אלא סוג של API שהדף מימוש. אובייקט מובנה מסוג Math, למשל, הוא אובייקט שmagיע מהשפה עצמה ומופיע בהגדרותה שלה. בעוד אובייקט גלובלי כמו console.error יכול להיות מיושם בדף כרום בצורה אחת, בפירופוק בצורה אחרת ובסביבת שרת בצורה שונה, אובייקט מובנה PI שמחזיר את ערך הפאי מתנהג באופן זהה להלוטין בלי קשר לדף או אם ג'אווהסקריפט רצה בסביבת שרת. חלק מהאובייקטים הם אובייקטים של ממש וחלקים ממומשים כפונקציה (שגם היא אובייקט).

גם את האובייקטים המובנים אפשר לדרוס. כאן למשל דורסים את האובייקט Math וגורמים ל-PI להחזיר ערך של 5:

```
console.log(Math.PI); // 3.141592653589793
(() => {
  let Math = {
    PI: 5,
  }
  console.log(Math.PI); // 5
})();
```

בהמשך הפרק נעבור על כמה אובייקטים מובנים חשובים.

parseInt

הפונקציה הגלובלית parseInt ממירת טקסט למספר. הפונקציה מקבלת שני ארגומנטים. הראשון הוא מחרוזת הטקסט והשני הוא הבסיס. המלצת הרשミת היא להשתמש בשני הארגומנטים, אבל בפועל צריך רק ארגומנט אחד. כך זה עובד:

```
const convertedNumber = parseInt('10230');
console.log(convertedNumber);
```

אפשר לראות שהารוגמנט הראשון הוא מחרוזת טקסט, אבל מיד אחרי שמעבירים אותו ב-parseInt רואים בקונסולה ש-convertedNumber הוא מספר. הפונקציה תעבור גם אם יש המן רווחים:

```
const convertedNumber = parseInt(' 10230      ');
console.log(convertedNumber);
```

אבל אם היא מקבלת משהו שהוא לא יכולה להתרומות אליו, היא מוחזירה NaN (כאמור, הקיצור ל-*Not a Number*):

```
const convertedNumber = parseInt('Ahla Mispar 10230');
console.log(convertedNumber); // NaN
```

ראוי לציין שבמקרה ההפוך פונקציית parseInt דוגא עובדת. אם היא נתקלה במספר שיש אחריו טקסט, היא נוננת 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 ומריצה אותו כailo מדובר בג'אוועסקריפט. כמובן היא מעריצה אותו (evaluation) כאילו הוא ג'אוועסקריפט ומהזירה את התוצאה.

בדוגמה זו למשל לוקחים את מחרוזת הטקסט `1 + 4` ומריצים אותה כאילו היא ג'אוועסקריפט. התוצאה של זה דבר היא `5`, וזה בדיק מה שמתקיים:

```
let result = eval('1 + 4');
console.log(result); //5
```

אפשר לנקח 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 בסkop הגלובלי. כיוון שפונקציית eval ממש מריצה את מחרוזת הטקסט כמו ג'אוועסקריפט, משתנה `a` מקבל ערך. כל מה שמוקף ב-`eval`, לא משנה מה אורכו, יירוץ כמו ג'אוועסקריפט. כתיבה של מהו כזה:

```
let a;
let result = eval('a = "Hello";');
console.log(a); // Hello
```

דוגמא לכתיבה של מהו כזה:

```
let a;
a = "Hello";
console.log(a); // Hello
```

אחד המשפטים הכי נפוצים בקרב מתכנתים הוא eval is evil. באופן עקרוני צריך להשתמש ב-`eval` רק אם אתם יודעים肯定amente מה אתם עושים. הסיבה היא ענייני אבטחת מידע – בדרך כלל מחרוזת הטקסט מגיעה בדרך זו או

אחרת ממשתמש, ומתן אפשרות למשתמש להריץ מה שהוא רוצה בקוד הוא פותח אדריך לביעות אבטחה ולביעות אחרות. לפיכך למדנו כאן את eval, אבל רק לצורך ההסבר מדוע מומלץ לא להשתמש בה לעולם...

Math

Math הוא אובייקט גלובלי שמאפשר לבצע פעולות הקשורות ל... מתמטיקה. אל תיפלו מהכיסאפה. יש לאובייקט לא מעט מתודות ותכונות, שאחת מהן (i) הכרנו בתחילת הפרק. רוב המתכנתים פוגשים את האובייקט הזה כאשר הם רוצים ליצור מספר רנדומלי, למשל כמספרים משתמשים שם משתמש חדש או כאשר מדדים משתמשים, משחקים, אנימציות 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 ולבודק:

```
let result = Math.random();
console.log(result);
```

האם המספר הזה רנדומלי? לא ממש, כיון שהוא מבוסס על הזמן ועל האלגוריתם שלוקח את חליקין הזמן ויוצר מספר רנדומלי. אבל התהיליך הזה רנדומלי מספיק כדי שיתקבל מספר אקראי. נשאלת השאלה, איך לזכהים את המספר הזה והופכים אותו למספר הרנדומלי שרצים לקבל – למשל מספר מ-1 עד 10. האמת היא זהה די קל – רק צריך להכפיל את המספר שיוצא ב-10 ואז לערל את התוצאה. למשל, אם המספר הרנדומלי שיצא הוא 0.9181803844895979 מכפילים אותו ב-10 ומעגלים כלפי מטה. התוצאה תהיה 9. איך מעגלים? בammedootot תכמה נוספת של Math שמשמשת ליצירת עיגול, שנקראה round. כך:

```
let result = Math.random();
result *= 10;
result = Math.round(result);
console.log(result);
```

בגרסאות ישנות יותר של ג'אווה סקריפט, לפני שהייתה קיימת האופרטור * שעלינו למדנו באחד הפרקים הקודמים, היו משתמשים במתודה pow (קיצור של power) לביצוע חזקה. המתודה קיבלה שני ארגומנטים, המספר שעליו החזקה פועלות 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'אווה סקריפט מחושבים כמלישניות (אלפיות השנה) ומותאמים לשעון של המחשב. בעידן המודרני, שבו השעונים מסונכרנים באופן אוטומטי עם הרשת, התאריך מדויק למדי כך. התאריך הוא מספר המילישניות ש עברו מ-1.1.1970 והוא מוסכם על כל מערכות המחשב כ-epoch time, הזמן שבו כביכול מערכת הchèלה היזונקס

הראשונה לפועל. כמעט כל שפות התכנות המוכרות משתמשות במוסכמה זו. חלון סופרות את השנהוות בעברו וחלון את המילישניות. בג'אווהסקריפט סופרים מילישניות.

כשנדרשים לחשב תאריכים בג'אווהסקריפט, צריך לחשב אותו עם Date. ברגע לאובייקטים הגלובליים שלנונו עד כה, Date הוא פונקציה בנאי שמתחלים עם המילה השמורה new. על פונקציה בנאיית ועל new למדנו בפרק על this ועל new. לשם הזכורת, new מבהיר אובייקט שלם. הפונקציה הבנאיית Date מקבלת ארגומנט של הזמן שמעוניינים לחקור. כאשר אין ארגומנט, הזמן הוא ה epoch time. ברגע שמתחלים את Date, נוצר אובייקט שמצוון לתאריך האתחול של האובייקט – קלומר היום, ברגע זהה בדיק.

הבה נדגים:

```
const dateObject = new Date();
let epochElapsedTime = dateObject.getTime();
console.log(epochElapsedTime);
```

כאן יוצרים date ללא ארגומנט, קלומר תאריך האתחול הוא 1.1.1970. משתמשים במתודה getTime כדי לקבל את מספר המילישניות שעברו מהתאריך שבו אוחROL ה-*date* (במקרה הזה 1.1.1970 עד היום. נסו ותראו!

אם רוצים לאוחROL את אובייקט ה-*date* לתאריך מסוים, אפשר להכניס ארגומנטים של תאריך. הארגומנט הראשון הוא שנה, השני חודש, השלישי יום, הרביעי שעה, החמישי דקה, השישי שנייה והשביעי והאחרון מילישניה. אפשר גם להכניס רק ארגומנט של שנה וחודש, ואז האובייקט יאותחל ביום הראשון של החודש. שנה וחודש הם המינימום הנדרש, אבל אפשר להוסיף עוד ארגומנטים, עד המילישניה. כאן למשל מאתחלים את האובייקט בתאריך המלא 16.09.1977, 14:30, 20 שנים ו- 500 מילישניות. אחרי שמתחלים את האובייקט אפשר להשתמש בgetTime כדי לבדוק כמה זמן עבר מהתאריך של epoch time עד התאריך הזה. שזה 245,849,420,500 מילישניות בדיק.

```
const dateObject = new Date(1977, 9, 16, 14, 30, 20, 500);
let myBirthdayElapsedTime = dateObject.getTime();
console.log(myBirthdayElapsedTime);
```

זה נראה מעט אבסטרקט. הבה נראה שימוש אמיתי באפשרות הזאת. נניח שמקבלים את גילו שלcko והרוצים לדעת בן כמה הוא בדיק. 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 currentDaObject = new Date();
let gap = currentDaObject - customerDateObject;
console.log(gap);
```

יש כאן את אובייקט התאריך של הלכה ואת אובייקט התאריך הנוכחי, ופישוט מחסירים אחד מהآخر. אם השנה עכשו היא 2018 ו נולדת ב-1978, חישוב הגיל הוא פשוט: $2018 - 1978 = 40$. כאן במקום בשנים סופרים במילישניות, אבל קשה לומר לבן אדם, "הין! אתה בן 1,305,505,055 מילישניות!" – צריך להשתמש בפורמט של בני אדם. פה נכנסה המתמטיקה לסייע, ונעשה חישוב פשוט שבו מחלקים את הפער במילישניות במספר המילישניות:

שיש בשנה:

```
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:

밀ישניות בשנה	1000
שניות בדקה	60
דקות בשעה	60
שעות ביום	24
ימים בשנה	365

מספר המילישניות בשנה הוא לפיכך: $1000 \times 365 \times 24 \times 60 \times 60 = 31,536,000,000$, כלומר 31,536,000,000 מילישניות, וזה המון. אבל אם מחלקים את מספר המילישניות במספר הזה, מגלים את מספר השנים. מובן שיצא מספר עם הרבה ספרות אחרי הנקודה, אך בדוק לשם כך קיים `Math.round`.

נוהג לשמר תאריכים כמעט תמיד כמספר המילישניות (או מספר השנה, ואז להכפיל באלף כשמגיינים לג'אוועהסקריפט) ולא כמחרוזות טקסט. זה נראה מעט מסורבל בהתחלה, אבל בזכות האובייקט `Date` ממש קל לעשות את זה.

JSON

לא מעט פעמים נדרשים להבהיר נתונים אל סקריפט מסוים או מנו, למשל לקבל נתונים משרת ולבוד אם הם או לטען או לשמר קובץ אם עובדים בג'אווהסקריפט בצד השרת. ג'אווהסקריפט בנזיה לעבור היבט עם פורמט נתונים שנזכר JSON, ראשי תיבות של JavaScript Object Notation. בגדול, הפורט זהה הוא אובייקט ג'אווהסקריפט שעליו למדנו כבר בעבר, אך בשינויים קלים:

1. מירכאות כפולות בלבד.
2. מירכאות סביב כל מחרוזת טקסט, גם סביב התוכנות.
3. אין פסיק בסוף התוכנה האחורונה.
4. אסור לכתוב הערוות בתוך JSON.
5. אין מתודות.

למשל, הקוד הבא הוא קוד ג'אווהסקריפט 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 ההבדל? כפי שנכתב לעיל – מירכאות כפולות סביב כל מחרוזת טקסט וגם סביב כל התוכנות ואין פסיק בסוף התוכנה האחורונה.

כאמור, JSON אמור להיות אינטואיטיבי לכל מי שמכיר ג'אווהסקריפט מספיק זמן. למעט השינויים הקלים הללו, מדובר בפורט פשוט וקל, וכמעט כל השפות בשירותים השונים עובדות אליו, מה שהופך אותו למש נוח. אבל כדי לעבוד אליו צריך להמיר אובייקט ג'אווהסקריפט ל-JSON ולהפוך. עושים את זה בклות באמצעות אובייקט JSON הגלובלי, שיש לו שתי מתודות – האחת הופכת אובייקט ג'אווהסקריפט ל-JSON תקין והאחרת ממירה JSON תקין לאובייקט ג'אווהסקריפט שאפשר להשתמש בו:

```
let someObject = {  
    name: 'Ran',  
    birthdate: 245849420500  
}  
let JSONobject = JSON.stringify(someObject);  
console.log(JSONobject) // {"name": "Ran", "birthdate": 245849420500}
```

בדוגמה שלילית לוקחים אובייקט ג'אוּהסְקְרִיפְט והופכים אותו באמצעות `JSON.stringify` לאובייקט JSON מן המניין בפורמט של מחרוזת, ומכאן השם של המתודה: `stringify` – להפוך למחרוזת. את האובייקט הזה אפשר לשגר לשרת באמצעות פקודה AJAX, לשמר על השרת או להעביר אותו בכל צורה אחרת. הוא נחשב מחרוזת טקסט לכל דבר. בצד השרת, כל שפה יודעת להתמודד עם JSON, שהפך דה פקטו לפורמט המרכזי של הרשת. שימושו לב שאובייקט JSON הוא מחרוזת טקסט לכל דבר. ככלומר דבר כזה:

```
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 יעבוד ויזורק הودעת שגיאה. על מנת לעבד עם מחרוזות טקסט שהיא אובייקט JSON כשר למהדרין, צריך להמיר אותה לג'אוּהסְקְרִיפְט. 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 מחרוזת טקסט. במקרה הזה הוא הוקלץ יונית, אבל במצבים מסוימים או בכל צורה אחרת. באמצעות `JSON.parse` הוא הומר לאובייקט רגיל שלו אפשר לעבוד כרגע, למשל להחליף את שם הלוקוח ל-Moshe-כפי שנעשה כאן.

setTimeout

מדובר בפונקציה גלובלית שמאפשרת לעכב את הקוד או לתזמון אותו. בדרך כלל בקוד רגיל אין בה שימוש, אבל בלי מעט דוגמאות משתמשים בקוד הזה. כמו כן, בקוד ג'אוּהסְקְרִיפְט שבודק UI יש חשיבות רבה לתזמון או לעיבוב, אבל בדרך כלל השימוש בפונקציה זו נחשב לתכנות גרוע. אם התוכנה שלכם צריכה עיכוב (`timeout`), סימן שיש בעיה בסקריפט. אז למה לומדים את הפונקציה הזאת? בעיקר כי צריך אותה לשם לימוד קוד אסינכרוני בהמשך. בנוסף על כן, זו פונקציה שכאמור נמצאת בדוגמאות רבות ברשות בכל מה שקשרו לאספקטים מתקדמים יותר של ג'אוּהסְקְרִיפְט (כמו גנרטורים – שהם חלק מתקדם בשפה שלא הגיעו אליו בספר הזה), אבל אפשר לגשת ל-MDN שהזוכר קודם כדי לבדוק, אחרת לא הייתה שום סיבה לכתוב עליו בספר הזה. אז מומלץ להתוודע לسانיטקס שלו ולהכיר אותה, אבל מאוד לא מומלץ להשתמש בה בקוד אמיתי בלי סיבה ממש-טובה (סיבה טובה היא יצירה אণימציות או משחקים, למשל).

חמושים באזהרות האלו, בוואו נלמד את הפונקציה של `timeout`. מדוכר בפונקציה גלובלית שזמיןה כמעט בכל הסביבות – דפדפן או סביבת שרת. איך היא עובדת? זו פונקציה שמקבלת שני ארגומנטים, פונקציה אונומית (המורכמת כפונקציה חז) שרצה מיד אחרי זמן העיכוב, וזמן העיכוב במיילישנות.

הינה הפונקציה:

```
console.log('start the code');
setTimeout(() => {
  console.log('After 5 seconds');
}, 5000);
console.log('end the code');
```

פונקציית החז, שמועברת כารוגמנט הראשון, תרוץ מיד לאחר שזמן העיכוב, שמועבר כארוגמנט השני, יסתתיים. הקוד לפיך ידפיס את מה שיש בקובנוללה בשורה הראשונה – "start the code", ויריץ את פונקציית העיכוב שתחכה חמיש שניות. הסקריפט לא יעצור ויריץ את השורה האחרונה "end of code". רק לאחר חמיש שניות, שנון 5,000 מילישניות, יוצג "After 5 seconds".

<code>start the code</code>
<code>end the code</code>
<code>undefined</code>
<code>After 5 seconds</code>

סדר ההרצה

- 1 `console.log('start the code');` ארוגמנט ראשון - פונקציה אונומית
- 2 `console.log('end the code');`
- 3 `setTimeout(() => {` ארוגמנט שני - מספר המילישניות
- 4 `console.log('After 5 seconds');`
- 5 `}, 5000);`

מה שחשוב להבין הוא ששאר הסקריפט רץ, וברגע שהזמן שהוקצב יסתתיים, מה שהועבר בפונקציית החז ירוץ מיד. לא כל הסקריפט עוצר; ההרצה של שאר הסקריפט מתקיימת כרגע.

פונקציית `setTimeout` מוחזירה מספר חיובי שבו אפשר להשתמש אותה בהמשך הקוד. לאיזה צורך זה שימוש? אם רוצים מסיבה כלשהי להשתמש להשميد את ה-`setTimeout`. ההשמדה של `setTimeout` נעשית באמצעות פונקציה מובנית אחרת שנקראת `clearTimeout`, והיא מקבלת כารוגומנט את המספר המזהה של ה-`setTimeout` שהוא 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 דוגמה הקודמת, אבל הפעם מקבלים את ה-ID שלו ומשתמשים בו כדי להרוג מידית את ה-`setTimeout`, שלוולם לא יתבצע.

תרגילים:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד 100 (לא כולל 100).

פתרונות:

```
function getRandom() {
  let result = Math.random();
  result *= 100;
  result = Math.round(result);
  return result;
}
const randomNumber = getRandom();
console.log(randomNumber);
```

הסבר:

يُ 创建 فونكציה عم شم كلشه. 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 ישיצא הוא 0.512, התוצאה תהיה 51.2. כיוון שהחיבים להחזיר מספרים שלמים, לוקחים את המספר, מעגלים אותו באמצעות `Math.round` ומוחזרים אותו כמו כל פונקציה רגילה.

תרגילים:

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 שהטווה הוא בין 0 לארגומנט.

תרגילים:

צרו פונקציה שמקבלת גיל משתמש לשנים (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:

יצרים 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 הפער. הוא מתקבל במילישניות, וכדי להסב אותו לשנים מחלקים אותו במספר המילישניות שיש בשנה. עכשו יש את מספר השנה שהוא הפער בין גיל המשתמש לתאריך הנוכחי. אם מספר השנה גדול מ-21 מחזירים true, ואם מספר השנה קטן מ-21 מחזירים false. איזה יופי.

תרגילים:

צרו פונקציה שמקבלת מספר X ומדפיסה בקונסולה את המשפט "רצתי לאחר X שנים".

פתרונות:

```
function delayFunction(int) {  
    const delaySeconds = int * 1000;  
    setTimeout(() => {  
        console.log(`After ${int} seconds`);  
    }, delaySeconds);  
}  
delayFunction(2);
```

הסבר:

יצרים פונקציה בשם delayFunction שיש לה ארגומנט אחד. את הארגומנט זהה מכפילים ב-1,000 כדי לקבל מספר במילישניות ומכניסים אותו קבוע delaySeconds. קוראים לפונקציה הגלובלית setTimeout. הפונקציה הזו קיימת בכל ג'אויסקייפ כחלק מהשפה, לא צריך ליצור אותה. היא מקבלת שני ארגומנטים. הראשון הוא פונקציה שרצתה אחרי מספר המילישניות שמעבירים בארגומנט השני. בארגומנט הראשון מעבירים פונקציה אונימית מסווג חז, שעלה לנו בפרק על הפונקציות, שכל מה שהיא עושה הוא להדפיס בקונסולה "אחרי X שנים". שימו לב שההדפסה נעשית עם תבנית שגם לנו 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 כמו גודלה של טקסט שרצים לעבוד איתה – למשל לבדוק אם מילים מסוימות נמצאות או לא ומצוות שם וכו'. אם רוצים לדוגמה לבצע אימיל, אפשר להשתמש בביטוי רגולרי פשוט כדי להבין אם מדובר באימייל או לא. הדרך הכי טובה 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 מציינת לכלל מסוים או לחיפוש והחלפה של טקסט. בכל פעם 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 (לוכנים) בלבד, אף על פי שהוא סוג אובייקט, וככה אפשר לזרזותו 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;
```

מקובל מאד להזכיר מיד אחרי הביטוי הרגולרי אופציות. במקרה זה האותיות `i` ו-`g` מסמלות אופציות – גם הביטוי הזה תקין לחהלוטין. יש לא מעט אפשרויות שונות שאפשר להזכיר. הנה טבלה קצרה של אפשרויות לדוגמה:

מה הוא עוזה	התו המיצג את ה-modifier
ביטוי רגולרי לא יתיחס להבדל בין אותיות הראשיות לריביות. <code>abc</code> יהיה זהה ל- <code>ABC</code> או <code>ABc</code> וכן להלאה	i
ביטוי רגולרי יתחשב בריבוי שורות בסימון של התחלת או סוף	m
כאשר יש כמה התאמות בחיפוש, הביטוי הרגולרי יחזיר את כלן במקום אחת בלבד	g

אפשר ליצור ביטוי רגולרי גם באמצעות פונקציה בניתה:

```
const regularExpression = new RegExp('abc', 'ig');
```

אם משתמשים בפוקנציה בניתה, אפשר להשתמש במשתנים בביטוי הרגולרי.

כל ההגמות להלן ייעשו בדרך הראשונה והפשטה יותר, אבל אתם מוזמנים לנסות גם את הדרך השנייה, באמצעות הפונקציה הבנאית, כאשר אתם מתרגלים.

הביטוי הרגולרי הראשון הוא `/abc/` שאומר פשטו כמשמעותו – כל ביטוי שכולל את האותיות `abc` ללא רווח ביניהן ולא אותיות אחרות, כמו למשל המחרוזת `abc`. נראה איך ג'אווה סקריפט עובדת עם ביטויים רגולריים ואחר כך עמוקיק לתוך הביטויים ומשמעותם, ולשם כך הביטוי הבסיסי הזה יספיק. כאשר יוצרים ביטוי רגולרי, יש לו באופן אוטומטי את המתוודות של הביטוי הרגולרי (בדיקות כמו `shlamarck` יש את המתוודות של `forEach` למשל).

המוודה הראשונה שנלמד היא `test`, שמאפשרת לבדוק שהביטוי הרגולרי נמצא במחוזות טקסט כלשהו. נניח שרוצים לבדוק אם יש בביטוי `abcdefg` את הביטוי הרגולרי `/abc/`:

```
const regularExpression = /abc/;
const textString = 'abcdefg';
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 מחוזות טקסט שאותה בודקים. בהים האמיתיים, מן הסתם, מחוזות הטקסט הזו מועבר על ידי המשתמש או על ידי שירות חיצוני כלשהו, אבל כאן, לשם הדוגמה, נכתב אותה לתוך משתנה קבוע. כיוון שהמשתנה `regularExpression` הוא ביטוי רגולרי, יהיה לו מתוודות של ביטוי זהה. מתוודה `test` מקבלת כארגומנט מחוזות טקסט ואם הביטוי הרגולרי נמצא במחוזות הטקסט, מוחזרה `true`. זה בדיקת מה שהיא עשוña כאן, כיוון שיש `abc` ב-`abcdefg`.

אם ננסה את זה במחוזות אחרה, שלא כוללת את הצירוף `abc`, יתקבל `false`:

```
const regularExpression = /abc/;
const textString = 'Nothing is here';
const result = regularExpression.test(textString);
console.log(result); //false
```

מוכן שבביטוי רגולרי יכול להיות הרבה יותר מורכב מזה. אפשר לסמץ, למשל, שרוצים שהביטוי יהיה נכון רק אם מחוזות הטקסט מתחילה ב-`abc`. לדוגמה, הביטוי `abcdef` יעבור את הבדיקה, אבל הביטוי `defabc` לא יעבור. את זה עושים באמצעות התו `^` שמשמעותו "התחלת", אם הוא לא נמצא בתוך סוגרים מרובעים. הביטוי `/^abc/` יהיה נכון רק לגבי ביטויים כאלה:

```
const regularExpression = /^abc/;
const textString = 'abcdef';
const result = regularExpression.test(textString);
console.log(result); //true
```

אתם מוזמנים לקחת את הדוגמה הזו ולנסות לשנות את מחוזות 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 התחלה, יש גם סוף. התו \$ מסמל את סוף המשפט שהביטוי הרגולרי מסתים בו:

```

const regularExpression = /abc$/;
const textString = 'defabc';
const result = regularExpression.test(textString);
console.log(result); //true

```

הביטוי הרגולרי זה יהיה תואם אך ורק את מה שנוגמר ב-abc.

אפשר לשלב ולסמן גם התחלה וגם סוף, מה שאומר שرك הביטוי abc בלבד יתאים:

```

const regularExpression = /^abc$/;
const textString = 'abc';
const result = regularExpression.test(textString);
console.log(result); //false

```

כל ביטוי אחר לא יתאים, כי תחমנו את abc בהתחלה ובסוף.

אפשר גם לציין אותיות חופשיות (wildcards). למשל, אם רוצים ביטוי שמתחליל ב-a ומסתיים ב-bc ובאמצע יש רק אחת אחת, אפשר להשתמש בביטוי ".(" (נקודה), שמשמעותותו אחת מכל סוג:

```

const regularExpression = /^a.bc$/;
const textString = 'a1bc';
const result = regularExpression.test(textString);
console.log(result); //true

```

אפשר לציין גם ביטויתו אחד לפחות באמצעות הסימן +. + משמעתו 1 או יותר. 1 או יותר ממה? ממה שמניע לפניו. למשל:

+ – הכוונה היא לאות a אחת או יותר.
+ – הכוונה היא לכל סימן אחד או יותר.

כלומר הביטוי:

$^a.+bc$$

תקין גם ל-a1bc וגם ל-a12bc וכן הלאה. ולא רק מספרים נכללים כאן, אלא גם אותיות, רווחים, סימנים מיוחדים וכל דבר אחר. נקודה () היא סוגתו כללית.

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`. במקרה יש מספר (`[0-9]`) אחד או יותר (`+`) משמעו `-`.

אפשר להשתמש גם בתווים. למשל, כאן אפשרים גם אותיות קטנות וגם מספרים:

`^a[0-9a-z]+bc$`

הביטוי הזה מתחילה ב-`a` ומיד אחריו מופיע הביטוי `[0-9a-z]`, שמשמעותו כל אות בין `0` ל-`9` או בין `a` ל-`z`, כולל אותיות קטנות ומספרים. כמה פעמים? `"+"` – ככלمر פעם אחת או יותר.

אפשר להשתמש באיזה טווה שרצוים, למשל `A-Z` או `0-5`.

לא חייבים להשתמש ב-`-`, יש עוד כמה ביטויי כמוות:

* – כוכבית, `0` עד אינסוף.

? – סימן שאלה, `0` עד `1`.

אפשר לסכם את הביטויים הרגולריים בטבלה הבאה:

סימן	פירוש הסימן
<code>^</code>	התחלת – מה שאמור הגיע בתחילת הביטוי
<code>\$</code>	סוף – מה שיש בסוף הביטוי
<code>.</code>	כלתו
<code>[]</code>	בתוך הסוגרים המרובעים יהיו טווים של אותיות
<code>+</code>	אחד לפחות מהביטוי המופיע לפני
<code>?</code>	אפס או ביטוי אחד מהביטוי המופיע לפני
<code>*</code>	אפס או יותר מהביטוי המופיע לפני

תחום הביטויים הרגולריים הוא עולם ומלאו, וספרים שלמים מוקדשים לו. חשוב לדעת שיש דבר כזה ומשתמשים בו לניטתה טקסט. שליטה בביטויים רגולריים היא מעבר לתחומו של ספר זה, אבל אתם חייבים לדעת שיש דבר כזה ולשלוט בו באופן בסיסי ביותר, מכיוון שמדובר בחילק משפט JS שמתכונתים משתמשים בו.

בעברית הביטויים הרגולריים קשים יותר. אי-אפשר לכתוב טקסט בעברית אלא צריך להשתמש בייצוג היווני-קדמי שלו. יוניקוד הוא הקידוד שבו מקובל להשתמש בשנים האחרונות והוא מונע את כל הגיבריש. כל צורות הכתב בעולם נמצאות ביוני-קוד, וכל אות ואות בכל מערכת כתוב מיוצגת על ידי מספר סידורי. כך למשל האות א' ביוני-קוד היא 05D0U, וזה מה שצריך לכתוב, וכן, יש גם יוניקוד ל... ניקוד. כאמור, הנושא הזה לא מכוסה בספר זה.

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:

যוצרם קבוע בשם regularExpression ומוכניסים אליו את הביטוי הרגולרי ahla, שימושתו מחרוזת טקסט של ahla. הביטוי הרגולרי מוקף ב-/-. כך מסמנים ביטויים רגולריים. לאחר מכן יוצרם קבוע שבו יש את מחרוזת הטקסט שאויה בודקים.

השלב הבא הוא להשתמש במתודה test. כיוון שלכל ביטוי רגולרי יש אותה באופן אוטומטי, יש את המתודה הזו גם ל-regularExpression שמכיל ביטוי רגולרי. המתודה הזו מקבלת טקסט כารגומנט, וזה בדיקת מה שהווער לה.

תרגילים:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אחת 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 רוצחים? אחת או יותר. לפיכך משתמשים ב-+. הביטוי:

[A-Z]+

משמעותו אחת לפחות מהטווות Z-A, כוללן אות גדולה אחת. לאחר מכן נסחים את הביטוי הרגולרי, אפשר לבדוק אותו בכל מחרוזת טקסט. יוצרים אותו באמצעות // ומכוונים אותו לקבוע בשם regularExpression. לוקחים מחרוזת טקסט (כגון בדוגמה: password) ומשתמשים במתודה test, שקיימת באופן אוטומטי בכל ביטוי רגולרי, על מחרוזת הטקסט הזו. המתודה מחזירה true אם המחרוזת מציינת לביטוי הרגולרי. כיוון שיש בה אות אחת גדולה, היא מציינת ויתקבל true.

תרגום:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אותן גדרות לפחות אחת לפחות ומספר אחד לפחות. כשהאות היא לפני המספר. למשל passworD1.

פתרונות:

```
const regularExpression = /[A-Z] [0-9] /;
const textString = 'passworD1';
const result = regularExpression.test(textString);
console.log(result); //true
```

הספר

כמו בתרגיל הקודם, עיקר הבעיה פה הוא לנסח את הביטוי הרגולרי: במקרה זה:

[A-Z]

הטווה הוא Z-A (клומר את גדולה).

[0-9]

הטווה הוא 0-9 (כלומר מספר).

הביטוי המשולב משמשו מחרוזת טקסט שיש בה את גודלה אחת לפחות ומיד אחריה מספר אחד. אחרי שמנסחים את הביטוי הרגולרי, יוצרים אותו בג'אווהסקריפט בין // ומכנים אותו לקבוע regularExpression. קבוע זה,

מהרגע שיש בו ביטוי רגולרי, יש את מתודת `test` שמקבלת כารגוומנט מהרווז טקסט, במקרה זהה את הסיסמה שהזונה, ובודקת אם היא מציינת לתנאי.

טיפול בשגיאות

כמתכני ג'אווהסקריפט מנוסים, בוודאי יצא לכם לראות שגיאות בקונסולה במהלך העבודה. השגיאות הללו נוצרות בדרך כלל אם טועים בסינטקס, למשל בקריאה למשתנה שלא ממש הוגדר. אם למשל כתבים:

```
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 המקום שללה. עד עכשו התיחסתי אל שגיאות כאלו משחו בלתי רצוי, אבל הרבה פעמים כן רוצים שגיאות. שגיאות והטיפול בהן הם חלק מכל מערכת ופונקציה חשובה. אם למשל כתבים פונקציה שבודקת סיסמה והמשתמש מכניס סיסמה בעברית בלבד, זו שגיאה שהיבטים לתפוס ולטפל בה. אם כתבים פונקציה שקוראת קובץ אבל הקובץ לא שם, חיברים ליצור שגיאה עבור מי שכותב את הפונקציה.

יצירת שגיאה היא ממש פשוטה. שגיאה היא אובייקט שנוצר מפונקציה בנאית ויוצרים אותו כך:

```
const myError = new Error('Oy Vey');
throw myError;
```

ראשית יוצרים את השגיאה עם הטקסט המוחדר לה. במקרה זהה "Oy Vey". אחרי שיש אובייקט שגיאה אפשר בכל שלב של הסקריפט לזרוק אותו. ברגע שיש שגיאה, כל קוד שירוץ אחריה יפסיק לרוין מידית, ממש כמו שגיאת הרצה מהסוג המוכר (והאהוב?), ככלmor הקוד הזה:

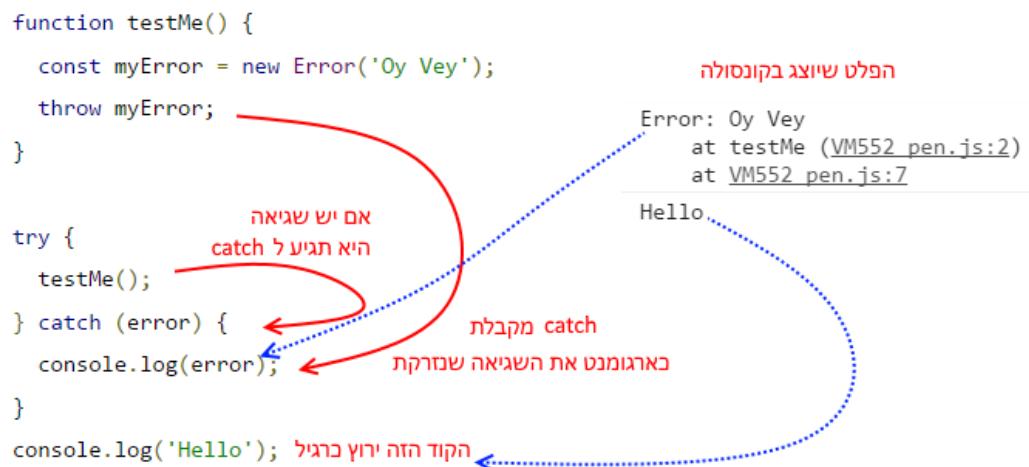
```
function testMe() {
    const myError = new Error('Oy Vey');
    throw myError;
}
testMe();
console.log('Hello'); // Will never run
```

ההדפסה של Hello בקונסולה לא תרוץ לעולם, אלא אם כן מדובר במקרה נדיר שבו חיבים להפイル מערכת. בדרך כלל יוצרים שגיאה כחלק מהניסיונו לניהל משהו. שגיאה שמפעילה את כל הסקריפט לא ממש תעזר כאן. בגלל זה לרוב זורקים את השגיאות לתוך בלוק שנקרא try-catch.

בלוק של try-catch הוא בעצם דרך לזרוק שגיאות ולטפל בהן בלי השבתה של הסקריפט כולו. בוגדול הוא מורכב מהמילה השמורה try שאחריה סוגרים מסוללים, ובhem הקוד רץ. אם לא תהיה שגיאה – טוב ויפה. אם תהיה שגיאה, קוד השגיאה ייכנס לבלוק 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');
```



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. במקרה זה, testMe תמיד זורקת שגיאה. בחרתי להדפיס את השגיאה בבלוק ה-catch. ההדפסה של>Hello מוחזק ל-try-catch, תודפס כרגיל.

כל קוד שעטופ ב-try ויזורק שגיאה, אפילו שגיאת קומפיילציה, לא יפריע למזהך התקין של הקוד וייה אפשר לניהל את דרך הטיפול בשגיאה באמצעות ה-catch. האם תדפיסו הודעה למשתמש? אם תנסו להריץ מחדש את הפונקציה? הכל אפשרי.

בדרך כלל מקובל שמודולים זורקים שגיאות וסומכים על מי שפעיל אותם שיטפל בהן לפי מיטב הבנתו. נניח למשל שיש אובייקט כזה:

```
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 סיסמות והוא אובייקט מודול שנבנה בעזרת פונקציה בנאית, מהסוג שכבר למדנו עליו. במקרה זה, בתוך האובייקט עצמו בודקים את הסיסמה באמצעות ביטוי רגולרי. אם הסיסמה לא מכילה אותן גדולות, זורקים שגיאה מסווג אחד. אם הסיסמה קצרה מדי, זורקים שגיאה מסווג אחר. כל זריקה של הסיסמה עוצרת את הסקריפט בתחום האובייקט.

אם מישמים את האובייקט כמו שצראיך, צריך לעטר אותו ב-try וב-catch:

```
try {
  new passwordTester('12345678');
} catch (error) {
  console.log(error);
}
```

בתוך ה-catch אפשר לנצל את השגיאה, למשל לזרפים אותה למשתמש, למשל. מקבלים את האובייקט של השגיאה, והטקסט מופיע בתחום אובייקט השגיאה תחת השדה message. אפשר להזapis את הטקסט של השגיאה, כמובן, או לעשות בו מה שרצו.

כמעט לכל מודול חיצוני או אפילו פנימי בגלאוסקריפט יש מדיניות שגיאות שא-י-אפשר לתחפש ולנהל. לא את כל השגיאות אפשר לנצל בצורה ממש טובה, והבחירה מה לעשות היא של המتنכת; לעיתים מעדיםם להשביט את כל הסקריפט, לעיתים מציגים הודעה שגיאה, לעיתים מתקנים את הקלט ומריצים שוב – מה שיש בתחום ה-catch הוא באחריותכם.

וז גם הסיבה שבגללה זה לא נחשי להברקה גדולה לעטר את כל הקוד ב-try-catch. ככה רק מונעים הדפסת שגיאות, אבל מפספסים את הנקודות החשובות של ניהול שגיאות – ניהול השגיאה הוא חשוב אם יודעים מאייה היא מגיעה ומה לעשות אליה. אם הסקריפט מורכב, בשלב העליון (try-catch) אי-אפשר לעשות כלום עם השגיאות ורקשה לדעת מהיכן הן הגיעו. לפיכך, מומלץ שתעטפו את חלקו הטעון שלם ב-try-catch רק אם יש לכם תוכנית מסודרת בנוגע למה צריך לעשות במקרה של קrise או במקרים מסוימים צופים שבהם ייעופו שגיאות.

finally

כתחספת ל-`try` יש את `try-catch-finally`. זהו בלוק קוד נוסף שניתן להוסיף לביטוי `try-catch` והוא תמיירן אחריו, בין `try` לבין `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 תריצו את הקוד הזה, תראו ש-`try` תמיד מודפס בקונסולת, גם אם תשנו את הסיסמה למשהו שעובר. בדרך כלל משתמשים ב-`finally` על מנת לבצע פעולה ניקיון. למשל אם מבצעים טעינה של משאב, לפני ה--`try` תהייה הצעה של איקון טעינה וב-`finally` תהייה 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 מילим. יש כאן פונקציה שמקבלת שני ארגומנטים ובודקת במשפט תנאי פשוט אם אחד מהם הוא לא מספר. אם הוא לא מספר, יוצרים שגיאה וזרקים אותה. שתי השרות האלו הן לב התרגיל:

```
const numberError = new Error('What?!? no number?? :(');
throw numberError;
```

כאן זרקים את השגיאה שיש בפונקציה. אפשר לראות שהשגיאה זו תודפס גם אם אין כאן כלל שגיאת קומפילציה. לאו והסקריפט בהחלט יכולה לחבר מחרוזת טקסט ומספר (היא פשוט תמיר את המספר לטקסט), אבל במקרה זה, הפונקציה לא תאפשר זאת ותזרוק שגיאה.

תרגיל:

נתון משתנה שהוא מספר. כתבו פונקציה שמקבלת אותו, ואם מדובר במספר שלילי היא זורקת שגיאה. עטפו את הפונקציה שכתחבם ב-`try-catch`. אם הפונקציה תזרוק שגיאה, 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
```

הסבר:

הפונקציה פשוטה למדי, היא מקבלת ארגומנט. אם הארגומנט הוא שלילי, זרקים שגיאה באמצעות:

```
const numberError = new Error('Negative Number');
throw numberError;
```

כדי לנחל את השגיאה, עוטפים את הקוד ב-`try-catch`. אם יש שגיאה, מאפסים את המשתנה. זה הכל. מה שיש בתוך `try` יזרוק את השגיאה אך לא יפריע למלאך התקון של שאר הסקריפט. מה שיש בתוך `catch` ינהל את השגיאה. במקרה הזה דרך ניהול השגיאה היא לאפס את המשתנה.

מבנה נתונים מסוג ו-Map

בפרק על מערכים למדנו שאפשר לאחסן בהם נתונים. אבל יש בעיה – במערכות, המפתח הוא מספר ואי-אפשר לקבוע אותו. אפשר לאחסן מידע אובייקטיבים, אך לא נוח לעשות זאת כיון שקשה לעבוד עם איטרציות אובייקטיבים, קשה למחוק ערך לפי המפתח שלו וקשה לספר את מספר הערכים. ג'אויסקייפ מספקת שני מבנים נתונים מתחכמים יותר לאחסן נתונים, וכך אפשר ליהנות מה יתרונות של אובייקטיבים אבל גם מהפונקציות של המערכים.

Map

מבנה הנתונים הראשון מאפשר להכניס ולאخזר ערכים בפורמט key value פשוט שבפשוטים. אין ערכים כפולים. בגודל, הוא בדוק כמו מערך, אלא שבמקום מספרים יש שם מפתח וערך. על מנת ליצור Map צריכים להשתמש בפונקציה הבנית new, שהזירה אובייקט שיש לו את התכונות ואת המתוות של Map.

למשל, כך יוצרים Map בסיסי עם המפתחות city, username ו-phone:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
```

הפונקציה set מאפשרת להכניס ערכים ל-Map והערכים יכולים להיות, כמובן, כל דבר, לא רק מחרוזות טקסט כמו זה אלא גם מספרים, מערכים ואפילו Map אחר. בזמן היצירה אפשר ליצור את ה-Map עם מפתחות. למשל:

```
let myMap = new Map([[ 'userName', 'Ran' ], [ 'city', 'Petah Tiqwa' ]]);
```

איך מוחזרים את השמות? באמצעות метод 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 בתוכנת size. שימושו לב שבשונה מערכים, כאן משתמשים ב-size ולא ב-length. הנה דוגמה:

```
const size = myMap.size;
console.log(size); // 3
```

מחיקה נעשית באמצעות המודה delete, שמקבלת כארוגומנט את המפתח של הערך שורוצים 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

```

שימו לב שבניגוד לערך, שם אם מוחקים איבר האורך שלו נותר כשהיה, כאן האורך משתנה בהתאם לערכים.

על מנת למחוק את כל המפה, צריכים להשתמש במתודה `clear` שלא מקבלת שם ארגומנט:

```

console.log(myMap.size); // 3
myMap.clear();
console.log(myMap.size); // 0

```

אפשר להשתמש בולולאת `forEach` ממש פשוטה על מנת לעבור על כל ה-`Map`, ממש כמו מערך, אלא שכן 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 of` עובדת יפה עם `Map`.

בגدول, מבנה הנתונים `Map` אמור להכיל כל מבנה נתוניים שהמפתחות שלו הם לא מספרים, ויש המון כאלו, מאובייקט של משתמש, דרך אובייקט של פעולה שהמשמש עושה ועוד כל דבר עצמו. היתרון האדריך של `Map` הוא שאפשר להיות בטוחים שיש רק מפתח אחד מכל סוג. אין חזרות ולא צריך לנהל את הדופלייקציות. נוסף על כך מרוויחים יותר של איטרציות, ממש כמו במערכות.

Set

מדובר במבנה נתונים דומה ל-`Map`, אך מעט פשוט יותר, שיש בו ערכים בלבד ללא מפתחות. בניגוד למערכות, שבhem המפתחות הם מספרים, או לאובייקטים ול-`Map`, שבהם יש מפתחות שיכולים להיות מהרוות טקסט רגילה, ב-`Set` יש רק ערכים - ככלמים אין ערכים כפולים. אם מנשים להכניס ערך ל-`Set` והוא כבר קיים, הוא פשוט "יידרס". `Set` נוח מאוד לאחסן נתונים כמו תגיוט, רשימת מצרכים וכל נתון שהוא שחייב בו הוא רק הערך שלו.

יצירת `Set` נעשית גם היא באמצעות פונקציה בניתה, כמו יצירת `Map`:

```
const tags = new Set();
```

מהרגע הזה יש Set שאפשר להכניס לתוכו ערכים. שימו לב שהכנסתי את ה-Set לתוכה קבוע. מהנקודה הזו איפואפשר להכניס קבוע משהו אחר, אבל מן הסתם אפשר להכניס ערכים לתוכה של ה-Set.

איך מכניסים ערכים? באמצעות המתודה add, שקיים לכל משתנה מסווג Set. הארגומנט היחיד שיש בה הוא הערך:

```
tags.add('tag1');
tags.add('tag2');
tags.add('tag3');
console.log(tags); // Set(3) {"tag1", "tag2", "tag3"}
```

ל-Set יש בדיקות איטרציות שיש לערכים ול-Map, כולל אפשרות לנתח 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
});
```

נחמד, נכון? אפשר לבדוק אם יש ערך מסוים ב-Set בעזרת מетод has, שמחזירה true או false:

```
tags.has('tag1'); // true
tags.has('Not here'); // false
```

כמו ב-Map, גם כאן מוחקים ערכים באמצעות delete ובודקים את גודל ה-Set בעזרת תכונה ה-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 מетод clear. כן, גם כאן יש דמיון ל-Map:

```
tags.clear();
console.log(tags.size); // 0
```

כאמור, Set שימושי מאוד לאחסנת ערכים שאין צורך במפתחות שלהם, כגון רכיבי מתכוון, תגיות, שמות וכו'. יכול להכיל כל נתון, ולא רק מחרוזות טקסט.

תרגילים:

צרו Map שיכיל עיר, רחוב, מספר בית ומיקוד.

פתרונות:

```
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:

ויצרים קבוע בשם address מסוג Map. שמו לב שהשתמשתי במילה השמורה new על מנת להפעיל את הפונקציה הבנائية Map. מהרגע שיש Map בתוך הקבוע, אי-אפשר לשנות את הסוג שלו, אבל כמו אובייקט שנכנס לתוכו קבוע – אפשר להוסיף ל-Map ולהוריד ממנו.

זה בדוק מה שעושים – ברגע שהקבוע address הוגדר כ-Map, אפשר להשתמש בMETHOD 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.

פתרונות:

```
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 בנאייה מסוג Set. מכניסים את ה-Set החדש לתוכו קבוע. זה לא אומר שאי-אפשר להכניס דברים ל-Set או למחוק דברים מתוכו, אבל אי-אפשר לשנות את names לסוג אחר. קל ונחמד.

משתמשים בMETHOD add, שקיים בכל Set להכנסת ערכים בלבד. אתם זוכרים שב-Set יש אך ורקערכים ולא מפתחות? זו הייחודיות של Set לעומת Map. הצגת מספר הערכים נעשית באמצעות METHOD size. זה הכלול.

תכנות אסינכרוני – קולבקים

כדי להבין מה זה תכנות אסינכרוני, כדי להבין מה זו אסינכרוניות. כשהושבים על תוכנה, בעצם חושבים על תוכנה סינכרונית. למשל התוכנה הבאה:

1. לטען את מערך א'.
2. לטען את מערך ב'.
3. למיין את מערך א'.
4. למיין את מערך ב'.

מה קורה אם שלב 1 ושלב 2 (טיענת המרכיבים) הם ארוכים? בכל הדוגמאות, מנوع הילאוסקריפט פשוט חיכה. שלב 1 מתבצע, אחריו זה שלב 2, אחריו שלב 3 ואז שלב 4.

תכנות אסינכרוני עובד קצת אחרת. בעצם כתובים את התוכנה לפי שלבים. הרוי אין היגיון להוכיח לטיענת מערך ב' אם מערך א' כבר נמצא בזיכרון ואפשר למיין אותו בזמן שמערך ב' נטען. תוכנה אסינכרונית נראה כך:

1. טוענים את מערך א'. כשהוא נטען (אם אין פוליה אחרת שמתבצעת) – ממיינים אותו.
2. טוענים את מערך ב'. כשהוא נטען (אם אין פוליה אחרת שמתבצעת) – ממיינים אותו.

קל לראות שההילך יעל בהרבה. טוענים את מערך א' ואת מערך ב'. ברגע שהראשון נטען, ממיינים אותו. זה לוקח הרבה פחות זמן. לא מביצים כלום במקביל – אבל בזמן שימושים למשהו שייטען, יכולים לעשות דברים אחרים. כדי לשים לב שזה שונה מפעולות דברים במקביל.

תכנות אסינכרוני הוא אחד מהפתרונות החזקים של הפלטפורמות השונות שמריצות לילאוסקריפט (מדפסן ועוד שרת), ולילאוסקריפט תומכת באסינכרוניות באופן מושלם, בטח ובטח בשימושים מודרניים. כשהותביבים סקריפט, יש לנו פועלות שMRIIZIM והtoutzaה שללה מגיעה מאוחר יותר. הנה נציגים זאת ונשתמש שוב בדוגמה של תוכנה פשוטה של המרת מטבחות. הסקריפט הפשט זה מקבל קלט מהמשתמש (למשל סוג המטבח ו对他), ניגש לשירות היצוני כלשהו באמצעות האינטרנט, על מנת לקבל את שער ההמרה, מכפיל את שער ההמרה בכמות המטבח ומציג את התוצאה למשתמש.

סדר פעולה	מה קורה בפועלה
1	קבלת קלט מהמשתמש – שם המטבח ו对他
2	גיישה לשרת היצוני כדי לקבל את שער ההמרה
3	קבלת הנתון מהשרת היצוני, חישוב הסכום והצגה 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 ולהחות עד שתגיע תשובה. גם אם השרת המרוחק הוא מאד-מאוד מהיר והrésה של המשתמש סופר-יעילה, עדין הפעולה הזו תיקח כמה מילישניות לפחות, ואז יש בעיה. אי-אפשר לבצע לשלב 4. האם עוזרים את פעלת הסקריפט והמשתמש "ננעלא"? או שמשיכים את פעלת הסקריפט לדברים אחרים? למשל, המשתמש אולי רוצה גם לבצע המرة לשער אחר או להפעיל סקריפט אחר.

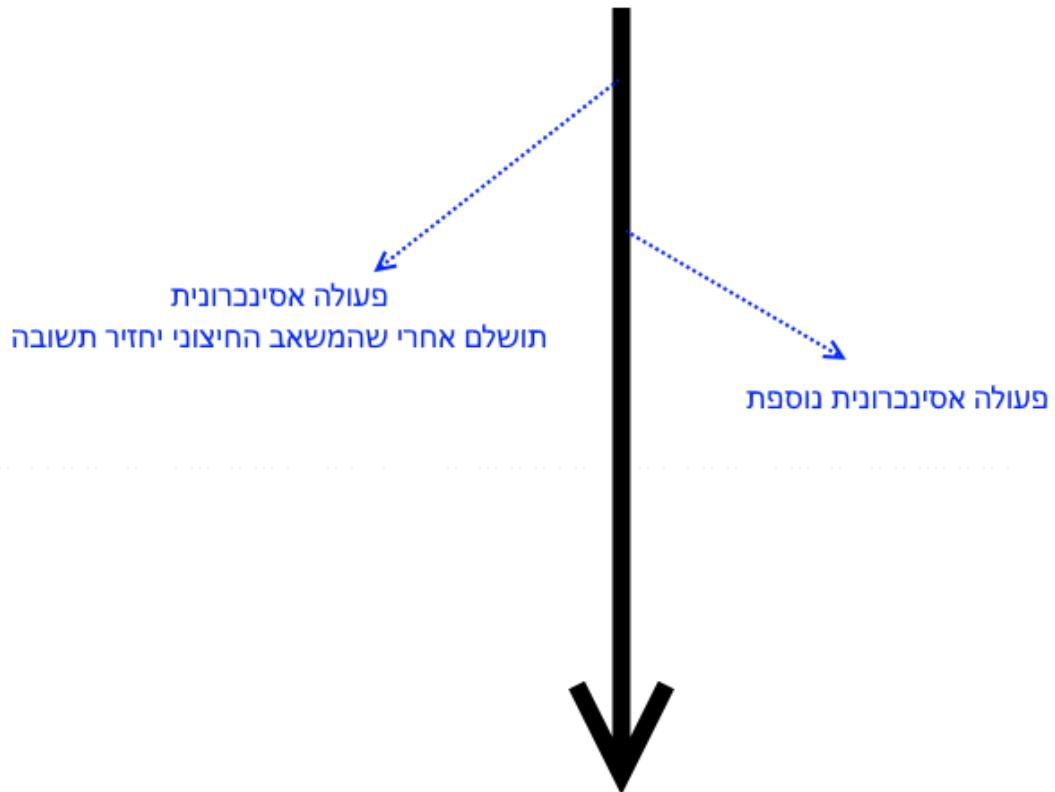
יש שתי אפשרויות:

האפשרות הסינכרונית – כל שלב נועל את השלב הבא. מ-1 עוברים ל-2. עד ש-2 לא מושלם לא עוברים ל-3 ובתча ובתча שלא עוברים ל-4. זו בעצם האפשרות הקלסית שאורכה למדנו עד עכשו. בקוד הבא:

```
console.log('1');
console.log('2');
console.log('3');
console.log('4');
```

2 יודפס אך ורק אחרי 1 ו-3 תמיד יודפס אחרי 2. הקוד רץ לפי סדר הכתיבה/קריאה. זה נקרא סדר סינכרוני. כל פעולה בג'אוהסקריפט היא סינכרונית וכך תמיד ירוין מההתחלת עד הסוף.

האפשרות האסינכרונית היא מעט מורכבת יותר. אם שלב 2 חייב רק לשלב 3, אפשר לומר לסקריפט, "שמע נא, מר בחור, שלח את הקראיה לשרת. ברגע שהשרת יחזיר עם תשובה, תציג את הפלט למשתמש. אבל עד שזה יגיע, אל תחכה לי. רוץ הלאה לשלב 4 ולשלבים שאחריו". ככלומר הקוד האסינכרוני לא חוסם ולא בולם את המשך הרצת הסקריפט. הוא כן חוסם את מה שצורך.



קוד אסינכרוני מופיע סדריפטים מודרניים. באפליקציות או אתרים כיום יש המון פעולות כאלו. למשל, רישום לדיסק קשיח, קריאה למשאב חיצוני, הפעלת מצלמת וידיאו או מיקרופון. במקומות לנעול את המשתמש, מאפשרים לו לעبور הלאה. בדיקות כמו שבגוגל דוקס, למשל, השמירה נועשית ברקע ואפשר לכתוב מסמך בלי הפרעה או "געילה". בדיקות כמו שבספייסבוק, לדוגמה, אם רוצים ליזום שיחת וידיאו, עד שהמשתמש השני מקבל את הבקשה אפשר לגלוּל בפיד או לעשות לייק. כל מה שכתבת לעיל נכתב בג'אווהסקרייפט ומשתמש בקוד אסינכרוני.

בתיאוריה זה טוב ויפה, אבל איך מתרגמים את זה למציאות? יש כמה שיטות ליישום אסינכרוניות. השיטה הראשונה היא שיטת הקולבקים. זו שיטה שקל ללמידה ולהבין, ולמדנו עליה בפרק על הפונקציות. בואו נחזר ונסביר מה זה קולבק. על מנת להמחיש את השיטה נזכיר הדמייה של פונקציה שקורסאת לשירות. הפונקציה הזו משתמש ב-setTimeout,

שאותו 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 עיכוב של מילישניה אחת. אפשר לעמוד פנים שהתשובה הזו הגיע מרוחק. בפונקציה אמיתית, לא יהיה `setTimeOut` אלא תהיה קריאה אחרת, אבל ככל מקרה הקריאה הזו תיקח זמן. אפילו מילישניה אחת (שהזה מעולמת) היא עדין זמן.

אם מנסים לקרוא לפונקציה הזו, מקבלים `undefined`. למה? כי הפונקציה עדין לא רצתה בכלל, אף על פי שמדובר בעיכוב של מילישניה אחת בלבד!

```

2 function getService() {
    setTimeout(() => {
        return 'Result from remote service';
    }, 1);
}

1 const answer = getService();
3 console.log(answer); // undefined

```

כלומר, כשהנמצאים בשלב 3, שלב 2 עדיין לא הספיק להתרחש. יש דרכם סינכרוניות לטפל בכך, אבל הבה נטפל בכך באופן אסינכרוני. אם רוצים שהשלב 3 יתרחש רק לאחר שלב 2 יישלם, הדבר היחיד הוא להכניס את שלב 3 לפונקציה ולגרום לפונקציה בשלב 2 לקרוא לה באופן אוטומטי. הדבר הזו נקראת "קולבק".

איך גורמים לפונקציה `getService` להריץ פונקציה אחרת? מעבירים לה ארגומנט שהוא בעצם פונקציה וקוראים לפונקציה הזו עם מה שמנגע מהשרת המרוחק:

```
function getService(cb) {
    setTimeout(() => {
        cb('Result from remote service');
    }, 1);
}
```

למדנו את זה לעומק בפרק על הפונקציות, אבל ארכייב שוב. ארגומנטים יכולים להיות פונקציות. ביגואו-סקריפט פונקציות הן אובייקטים. כמו שאפשר להעביר מחרוזת טקסט, מספר, מערך וכל דבר אחר, אפשר להעביר פונקציה. ברגע שהיא בתוך משתנה, קוראים לה כמו פונקציה רגילה ומעבירים לה איזה ארגומנט שרווצים. במקרה הזה, הארגומנט הוא סתם מחרוזת טקסט מומצת. בקריאה לשרת הוא יכול להיות מחרוזת טקסט ממשמעותית או אובייקט מידע.

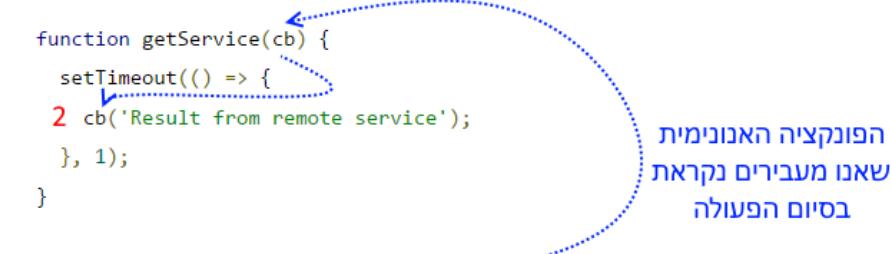
הfonקציה מקבלת ארגומנט של פונקציה וקוראת לו ak ורק כשהפעולה מסתיימת, ובמקרה זה, אחרי מילישניה אחת:

```
function getService(cb) {      ה콜בק מופעל  
  setTimeout(() => {      רק בסוף הפעולה  
    cb('Result from remote service');  
  }, 1);  
}
```

עכשו מפעילים את הפונקציה. צריך רק לקרוא ל getService, כשהארוגומנט הוא הפונקציה שתווסף מייד לאחר שהפעולה של getService תושלם. מעבירים ארגומנט פונקציה אונונימית מסוג ח' שתדפיס את מה שמעבירים אליה, זה הכל. שימו לב שה cb זה הארגומנט אותו מעבירים והוא אומר להיות פונקציה, כמובן:

```
function getService(cb) {  
  setTimeout(() => {  
    cb('Result from remote service');  
  }, 1);  
}  
getService(  
  (answer) => {  
    console.log(answer)  
  }  
);
```

```
function getService(cb) {  
  setTimeout(() => {  
    cb('Result from remote service');  
  }, 1);  
}  
  
1 getService(  
3 (answer) => {  
  console.log(answer)  
}  
);
```



הfonקציה האונונימית
שאנו מעבירים נקראת
בסיום הפעולה

ככה עובד קוד אסינכרוני עם קולבקים. הפעיה מתחילה כשיוצאים ממוחוזת הדוגמה אל מוחוזת המיצאות. נניח שיש אפליקציה שבה הלוקה מכנים מספר בשקלים ובזוק כמה מנויות של חברה זרה הוא יכול לנקוט. צריך לעשות את הפעולות הבאות:

1. לקבל את הקלט מהлокה – מחיר בשקלים וסוג מניה.
2. לראות מה שער הדולר באמצעות קריאה לשירות חיצוני ולבזוק כמה דולרים השקלים האלו שווים.
3. לקבל את מחיר המניה באמצעות שירות חיצוני אחר ולראות כמה מנויות אפשר לנקוט.
4. להציג את המידע לлокה.

כלומר, אם הלוקה רוצה לרכוש ב-40 שקל מנויות של חברת אינטל, קוראים לשירות מסויים שיראה מה שער הדולר. נניח שהוא 4, ולлокה יש 10 דולרים. אחרי כן קוראים לשירות אחר שיראה את מחיר המניה, נניח שהוא 5 Dolars. הפלט שיוצג לлокה הוא 2 מנויות. 40 לחלק ל-4, או בתרגיל: 40:4:5.

איך ממשים את זה? הנה השירותים שمدמים קריאה לשער הדולר וקריאה למחיר המניה:

```
function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}

function getStockPrice(stockSymbol, cb) {
  setTimeout(() => {
    cb(5);
  }, 1000);
}
```

השירות המזוייף הראשון `getCurrencyRate` דומה לזה שתרגנו – הוא קורא לקולבק שמעבירים לו ומהזיר 4. השירות המזוייף השני הוא `getStockPrice`. מעבירים לו שני ארגומנטים – הארגומנט הראשון הוא סימן המניה והשני הוא קולבק. השירות המזוייף לא משתמש בסימן המניה, אבל הוא כן קורא לקולבק 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
    });
  });
);

```

או מבצעים קרייה ל-`getCurrencyRate` כמספרים פונקציית חן ארגומנט. פונקציית החן קיבל את שער הדולר מהשירות המזוייף ותקרה מיד לשירות שמחזר את מחיר המניה. שער הדולר ומהירות המניה יסייעו לחישב כמה מנויות אפשר רכוש. במקרה זה – שתיים.

אפשר לראות שהמבנה הזה מסובך מדי. קולבק בתוך קולבק.

```

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
    });
  }
);

```

콜בק ראשון

콜בק שני

המצב הזה עוד יכול להסתבר ככל שהסקריפט מורכב יותר, ואו בהחלט אפשר לראות שלושה וארבעה קולבקים ואפילו יותר מקרים זה בתוך זה. מצב זה נקרא "גיהינום הקולבקים" (callback hell). איך פותרים את זה? בעזרה promises, כפי שנלמד בפרק הבא.

תרגילים:

נתונה פונקציית `getCurrencyRate`, שמדמה החזרה של שער הדולר באמצעות שרת חיצוני.

```
function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}
```

קראו לפונקציה זו והציגו את התוצאה שלה בקונסולה.

פתרונות:

```
getCurrencyRate(
  (answer) => {
    console.log(answer)
  }
);
```

הסבר:

קוראים לפונקציה `getCurrencyRate` עם ארגומנט אחד. איזה ארגומנט? פונקציה אנונימית מסוג ח' שמקבלת ארגומנט. הארגומנט זה יודפס. איך זה עובד בפועל? פונקציית `getCurrencyRate` מצפה לקבל ארגומנט שהוא קולבק. היא מדמה קרייה לשרת ומפעילה את הקולבק עם ארגומנט שהוא התשובה. הפונקציה האנונימית שיצרהם מקבלת את התשובה ומחזירה אותה.

תרגילים:

צרו שירות מזויף המדמה קרייה לדיסק הקשיח וקריאהקובץ. השירות המזויף מקבל שם קובץ וקולבק. הוא תמיד יחזיר טקסט שבו כתוב "`this` is from filename", ושם הקובץ יהיה השם שמועבר לו. קראו לשירות המזויף והדפיסו את התוצאה בקונסולה.

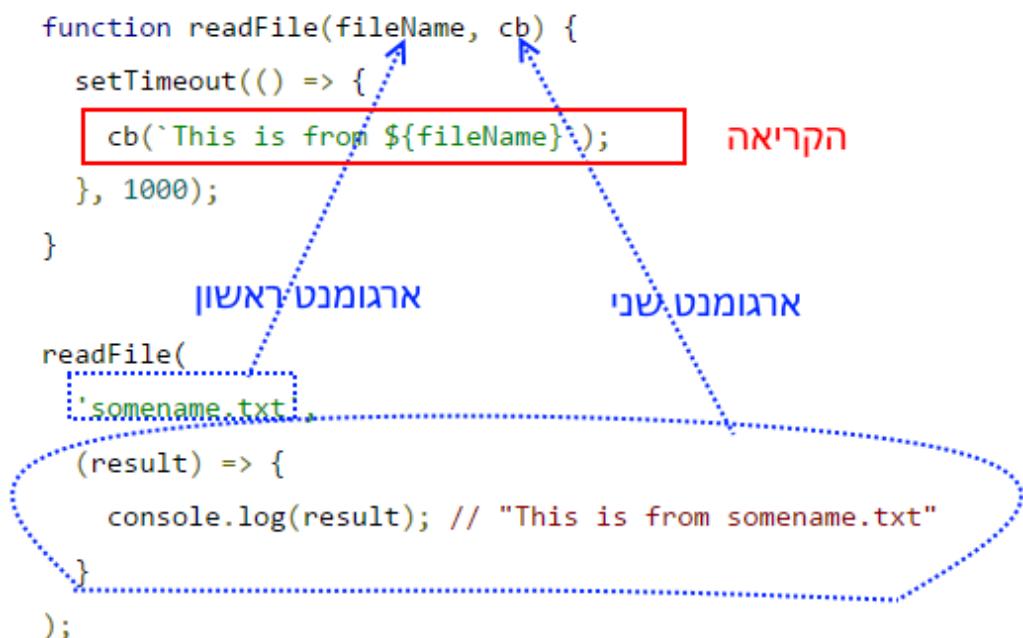
פתרונות:

```
function readFile(fileName, cb) {
  setTimeout(() => {
    cb(`This is from ${fileName}`);
  }, 1000);
}

readFile(
  'somename.txt',
  (result) => {
    console.log(result); // "This is from somename.txt"
  }
);
```

הסבר:

השירות `readFile` הוא שירות מזוייף מהסוג שכבר יצרנו. הוא מקבל שני ארגומנטים, שם הקובץ ו콜בק. את הקולבק הוא מפעיל לאחר עיכוב של שנייה ומשתמש בשם הקובץ כדי להעביר אותו כารוגמנט ל콜בק. Unless you're reading this in Hebrew, you might be wondering what's happening here. The code is using a closure to capture the value of `fileName` at the time it was defined, even though it's being used inside a function that has already been executed. This is a common pattern in JavaScript called a "closure".



Promises

כפי שראינו, קולבקים הם שיטה ייעילה מאוד לעבוד עם קוד אסינכרוני, אבל יש להם חסרונות, ובראשם הסרבול הגדלן מאד של הקוד או "גיהינום הקולבקים", וגם חסרונות אחרים – כמו למשל חוסר היכולת לתפוס שגיאות (את עוין ה-`try-catch` למדנו באחד הפרקים הקודמים). כדי לנחל את השגיאות קיימים פרומייסים (promises, "הבטחות" באנגלית). העיקרון מאחוריהם הוא פשוט: פונקציה אסינכרונית מוחזירה "הבטחה" שהיא יכולה להפר או לקיים. אם למשל יש פונקציה שקוראת לשירות החיצוני, היא יכולה לקיים את ההבטחה בכך שתחזיר את התוצאה מהשירות המרוחק או להפר אותה אם השירות המרוחק נכשל. במקרה שקוראת לפונקציה אפשר להחליט מה קורה אם ההחלה מתאפשרת ומה קורה אם לא.

איך יוצרים promises? הנה נשתמש בשירות המזוייף מהפרק הקודם. השירות מחייב שנייה ואז מוחזר 4. הוא מדמה שירות שמחשב שערית מטבעות וקורא לשרת החיצוני. הקוד הוא פשוט מאד:

```
setTimeout(() => {
    // Return 4
}, 1000);
```

ובכן שזו יכולה להיות קריאה לשרת אחר, קריאה למ Lager נתונים או לכל מקום שהוא. בסופו של דבר, מדובר בפקודה שמקבלת קולבק שפועל אחרי 1,000 מילישניות. למדנו על כך בפרק על `setTimeout` ועל גם בפרק הקודם.

מה שעושים הוא ליצור אובייקט הבטחה בעזרת פונקציה בנאית. האובייקט מקבל ארגומנט אחד שהוא קולבק, שיש לו שני ארגומנטים – פונקציית `resolve`, שלה קוראים כשההבטחה מצלה, ופונקציית `reject`, שלה קוראים אם רוצים שההבטחה תיכשל. יצרת הבטחה נראה כך:

```
let myPromise = new Promise((resolve, reject) => {  
});
```

כאן ממש אפשר לראות את יצרת ה-`Promise` באמצעות פונקציה בנאית שמקבלת כารוגמנט פונקציית חוץ אונומית, שלה יש שני ארגומנטים שהם בעצם קולבקים.

כל מה שנשאר לעשות הוא להכניס את השירות המזוייף ולקראות ל-`resolve` שבתוכו. כאן רוצים למש את ההבטחה עם 4:

```
function readFile() {  
    let myPromise = new Promise((resolve, reject) => {  
        setTimeout(() => {  
            resolve(4);  
        }, 1000);  
    });  
    return myPromise;  
}
```

אפשר לראות שהפונקציה הבנאית מקבלת פונקציית חץ אוניברסלית שיש לה שני ארגומנטים – `resolve` ו-`reject`. מדובר בשני קולבקים שאפשר להפעיל. הראשון הוא בעצם הפעלה במקרה של הצלחה, וזה בדוק מה שעושים בשירות המזוייף.

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

השירות המזוייף ממחכה שנייה אחת (1,000 מילישניות) וואז קורא ל-`resolve` ועביר את הארגומנט 4. הכל אמור בתוך פונקציית `readFile`. הפונקציה מוחזירה הבטחה. אפשר לקרוא לפונקציה זו ולקבל את הבטחה.

מה עושים עם הבטחה? אפשר בעצם לבוא ולומר, "תשמעי, הבטחה, ברגע שתתמלאי, תריצי את הקוד זהה". אין

עושים את זה? לכל הבטחה יש מתודת `then` שמקבלת כารוגמנט רាសון פונקציה שמופעלת כשההבטחה מתמשחת:

```
readFile().then((result) => { console.log(result); })
```

הקוד הזה רץ בכל פעם שההבטחה הנמצאת ב-`readFile` מתמשחת.

בואו נמחיש את זה באמצעות דוגמה נוספת, הפעם אfilו בלי `setTimeout`: יוצרים פונקציה שמחזירה הבטחה.

שמה `readFile`, והיא לא תקבל שם ארגומנט. מה שהיא עשו היא ליצור הבטחה. בתוך הבטחה, במקום לקרוא לשירות זה או אחר, פשוט עושים `resolve` ומ畢ים את הבטחה.

כשקוראים לפונקציה, משתמשים ב-`then` ועבירים כארוגמנט פונקציה שמאזינה להבטחה. ברגע שהיא 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. הבטחה 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;
}

readFile().then(
    (result) => {
        console.log(result);
    }
);
```

ההבטחה שיצרת
פונקציה הראשונה המועברת בארגומנט תיירה
כארה הבטחה תתקיים
ירוץ כאשר הבטחה תתקיים

מה חשוב להבין הוא שיש לשימוש בהבוחות שני חלקים – החלק הראשון הוא יצירת הפונקציה שבה יש את הבטחה וכתיית הבטחה, כולל החלק שבו מבצעים resolve או מקיימים את הבטחה. החלק השני הוא האזנה, באמצעות then, לפונקציה שמחזירה הבטחה.

כמו שאפשר לקיים הבטחה, אפשר גם להפר אותה. למשל אם השירות מדווח שאין כי השרת לא נמצא, כי מסך הנתונים נפל או מכל סיבה שהיא. הפרת הבטחה נעשית באמצעות המתודה 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 לקיים את הבטחה אחרי שתי שניות, מפרים אותה בgasot. שימושו לב שהכל נראה אותו דבר: בניית הפונקציה שעוטפת את הבטחה והשימוש ב-setTimeout. השוני העיקרי

כאן הוא השימוש ב-reject. כך מפרים את ההבטחה. גם פה אפשר לשולח ארגומנט שיתפס בפונקציה שקולטת את ה-reject.

הפונקציה שתופסת את הפרת ההבטחה נכנסת כารוגומנט השני של פונקציית then. הערה: בקוד מורכב מומלץ לעשות reject עם אובייקט שגיאת מסודר, אחרת לא מקבלים לוג שגיאות מסודר וזה עלול לבבלבל.

```
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!  
);
```

הפונקציה הראשונה תרוץ במקרה וההבטחה תקיים
הפונקציה השניה תרוץ במקרה וההבטחה תופר

הפונקציה הראשונה מתמשחת רק כשההבטחה מקוימת. במקרה זה – זה לעולם לא יקרה. הפונקציה השנייה היא החידוש פה: מעבירים אותה כארוגומנט השני של ה-then והוא תפעל אך ורק כאשר ההבטחה תופר. במקרה זה היא מדפיסה את מה שהיא מקבלת בكونסולה בתוספת המילים Bad error. כיוון שבהפרת ההבטחה מחזירים את המילים ERROR! ERROR!, מקבלים בكونסולה את המשפט:

"Bad error: ERROR! ERROR!"

בשימוש בהבטחות צריך תמיד לזכור לסגור את כל המקרים, כדי שלא יקרה שהקוד פשוט יפסיק לרוץ מפני ששכחתם להשתמש ב-`reject` באחד ממשפטי התנאי.

שרשור הבדיקות

אחד הפיצ'רים החזקים ביותר בהבטחות הוא היכולת לשורש אותן – כלומר ה-`then` יכול להזיז promise שיתפס על ידי `then` שמשורשר אליו. ראשית נשאלת השאלה – למה צריך את זה? התשובה פשוטה: פעמים רבות פונים

לשרת, ואת התגובה שמקבלים ממנה שלוחים לשרת אחר או ששמורים במדד הנתונים או כל דבר אחר. שרשור הבתחות או פונקציות אסינכרוניות הוא נפוץ מאוד.

בדוגמה הבאה נראה איך עושים את זה. יש שתי פונקציות שימושות הבטחה. אחת היא `getNumber` שתמיד מקיימת את הבטחה ומחזירה 100, והאחרת היא `writeNumber` שבודקת את המספר. אם המספר הוא 100, היא מקיימת את הבטחה ומחזירה OK. כאן אפשר לראות את התרגול – הפונקציה `writeNumber` תלולה בפונקציית `getNumber` שהחזיר לה את המספר. הן מוממשות כמו כל הבטחה שכבר ראיינו, אבל מה שמיוחד בשרשור הבתחות 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 שימושים את הקראיה ל-`getNumber` כרגע ומשתמשים ב-`then` כדי לתפוס את הקראיה הראשונה. מה שמיוחדפה הוא שלוקחים את התוצאה של קיום הבטחה הראשונה (שתהיה מן הסתם 100) ובמקום להחזיר אותה לפונקציה, מבצעים קראיה של הפונקציה השנייה עם התוצאה (שכמובן היא 100) ומחזירים אותה! זו הבטחה לכל דבר ועניין ואפשר לתפוס אותה עם `then`, שם מדפיסים את התוצאה, מן הסתם היא OK.

ammoish 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 מוחזיר תמיד הבטחה, גם אם לא קוראים לפונקציה שמחזירה את ההבטחה יותר מפעם אחת.

ואיך תופסים בשיטה זו הבטחה שנכשלת? בעזרת מתודה catch שמקבלת לתוכה את מה ששולחים ב-reject, בדיק כמו במתודה השנייה ב-then הרגיל שלמדו עליו 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!
    });
}

```

הדוגמא הו זהה לדוגמה הקודמת למעט שני דברים: נוסף שתופס ומדפיס כל error, ואת ה-`reject` מוחזרים כ-`catch`. הוספה ה-`catch` מאפשרת לתפוס שגיאות בכל רגע נתון כמשמעותם `then`.

קיובץ הבטחות

אם יש כמה הבטחות שאין תלויות זו בזו ורוצים לשולח את כולן יחד ולקבל את התוצאות של כולן בצורה מסודרת, אפשר להשתמש ב-`Promise.all`. מדובר בפונקציה גלובלית של האובייקט `Promise` שמקבלת מערך של הבטחות. אפשר לשרר לה `then` שיטפל במצב שבו כולן עוברות או שאחת מהן לפחות נכשלה, אף על פי שסדר השלילה וההפעלה אינה מובטחת.

כלומר אם יש שלוש הבטחות, אפשר להאזין לקיום של שלושתן בהתאחת. אם כולן עוברות, הפונקציה הראשונה ב-`then` מופעלת. אם אחת נכשלה, הפונקציה השנייה מופעלת, ממש כמו בהזנה לבטחה 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 נצרכו שלוש פונקציות שמחזירות הבטחה פשוטה. יוצרים מערך של שלוש הבטחות שלhn. שימוש לב שבמערך קוראים לפונקציות כדי לקבל את הבטחות שלhn. המערך הוא של הבטחות ולא של פונקציות. את המערך מכניםils ל-all ואפשר להשתמש עבביו ב-then. כיוון שיש כמה הבטחות וכמה תוצאות לקיום הבטחות, הכל נכנס לערך של results. זה השוני היחיד.

תרגילים:

צרו שירות, באמצעות setTimeout, שמחזיר לאחר שתי שניות תגובה עם המילים Two seconds passed. הכניסו אותו להבטחה והכניסו את הבטחה לתוך 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!');
    }, 2000);
  });
  return secondsPromise;
}
waitTwoSeconds().then((result) => { console.log(result) });
```

הסבר:

החלק הראשון הוא יצירת ה-setTimeout, שמקבל שני ארגומנטים. הראשון הוא מה שיש לעשות אחרי פרק הזמן שהוגדר לו והשני הוא פרק הזמן במלישנות. 2,000 מילישניות הן שתי שניות.

החלק השני הוא יצירת פונקציה שמחזירה הבטחה. יוצרים פונקציה בשם waitTwoSeconds. בתוכה יש הבטחה בשם secondsPromise ואלה מחזירים.

החלק האחרון והקשה ביותר הוא ליצוק תוכן בתוך הבטחה. הבטחה נוצרת באמצעות פונקציה בנתית של Promise. היא מקבלת ארגומנט אחד שהוא פונקציית קולבק עם שני ארגומנטים, resolve ו-reject. בתוך הקולבק מכניםים את ה-setTimeout וקובעים שההבטחה תקיים ברגע שתשתי השניות עברו.

הבדיקה פשוטה יותר. קוראים לפונקציה ואז מזינים עם then. היא מקבלת ארגומנט אחד, פונקציה אחת שפועלת ברגע שההבטחה מתקימה. כל מה שצריך לעשות הוא להאזין למה שהזרים ב-result ולהדפיס אותו בקונסולה:

```

function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Two seconds passed!');
    }, 2000);
  });
  return secondsPromise;
}

// מכיון שהפונקציה
// מתקיימת, הנטה
// זו מופעלת
// כאשר ההבטחה
// מוחזיר בקיים
// מה שANI
// מועבר לכך
waitTwoSeconds().then((result) => {console.log(result)});

```

תרגיל:

צרו פונקציה checkIfNumberOK שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזור הבטחה מקוימת עם מהירות הטקסט Number OK. אם המספר קטן מ-10 או שווה לו, תחזור הבטחה מופרת עם מהירות הטקסט Number bad.

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
);

```

הסביר:

הfonקציה checkIfNumberOK היא פונקציה רגילה המקבלת מספר כארוגומנט. בתוך הfonקציה יוצרים הבטחה כאובייקט Promise עם הfonקציה הבנאית. הfonקציה מקבלת ארגומנט שהוא פונקציה אנוונימית מסוג חצץ, ובה יש שני ארגומנטים, reject ו resolve. משתמשים בשנייהם. מפעלים משפט תנאי פשוט ובודקים: אם המספר גדול מ-10, ההבטחה תקווים באמצעות resolve, ואם המספר קטן מ-10 היא תופר באמצעות reject - כמובן עם הטקסטים הרלוונטיים.

בדיקות יהיו פשוטות למדי – הרצה של הfonקציה עם המספר 11 ותפיסה הצלחה או השגיאה באמצעות then. כיוון שה-fonkציית checkIfNumberOK מוחזרת הבטחה, אפשר להשתמש ב-then שמקבל שני ארגומנטים, ארגומנט של הפורה וארוגומנט של הצלחה. מן הסתם, כשביראים 11, ההבטחה מקוימת והfonקציה הראשונה שsavvirim ב-then תופעל. כשביראים מספר קטן יותר מ-10, ההבטחה מופרת והfonקציה השנייה שsavvirim ב-then תופעל.

fonקציית **async**

השיטה השלישייה לתוכנות אסינכרוני לא שונה כל כך מהשיטה של הבטחה. למען האמת מדובר בהרחבה שלה. פונקציית async מאפשרת לטפל בהבטחות ללא ה-then או ליתר דיוק מאפשרת לכתוב קוד אסינכרוני בצורה סינכרונית. אם יש שירות שמחזיר promise, אפשר לבצע אם לעבד עם then כמו בפרק הקודם או בfonkציה אסינכרונית עם המילה השמורה **async**.

לשם הדוגמה השתמש בדוגמה של הקראיה המזוינה שהשתמשנו בה קודם. הקראיה זו לא שונה כלל ממה שלמדנו בפרק הקודם:

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

מדובר בקריאה שמחזירה תשובה מזוינה של 4 אחרי שנייה אחת. שימו לב שהיא מוחזרת promise. אם היינו רוצים להתחבר אליה, היינו צריכים לקרוא לה ואז להשתמש ב-then כדי לתפוס את ההצלחה. אבל אפשר לעשות את זה עם **async** פשוט. ראשית מגדירים פונקציה אסינכרונית, הכוללת הfonkציה זו מ Ritchie קוד שיכול לרוץ בזמן אחר. ההגדרה נעשית באמצעות המילה השמורה **async**. בתוך הfonkציה זו אפשר לקרוא לפונקציות המוחזרות הבטחה כפי שרוצים. פונקציות המוחזרות הבטחה יקבלו await ליד הקראיה שלהן, והסקריפט בתוך ה-**async** יעזור וימתין להן. קוד מחוץ לפונקציה ירוץ כרגיל. כך מרויחים אסינכרוניות אבל עם קוד ברור יותר.

למשל:

```
async function main() {  
    let result = await readFile();  
    console.log('result' + result);  
}  
main();
```

בדוגמה יוצרים פונקציה שモוגדרת כאסינכרונית. שימושו לבי' למילה השמורה `async` לפניהם `function main()`. בתוכה הפונקציה הזו, שיש בה `async`, אפשר להשתמש במילה השמורה `await` שקוראת לפונקציה המחזירה `promise`. שימושו לבי' ש-`readFile` לא שונה מפונקציה אחרת שמחזירה הבטחה, בדיקות כי שלמדנו בפרק הקודם. פונקציות `promise` פשוטណ גודעו לנוהל את ההבטחות, 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 יורץ עד שה `await`
לא יישלם

כפי שב-`then` יש שני ארגומנטים – אחד שמתקיים כאשר ההבטחה מקוימת ואחד שמתקיים כאשר ההבטחה נכשלה. – גם ב-`async` אפשר לתפוס את הכישלון. איך? בעזרת `try-catch` רגיל, שעליו למדנו בפרק על טיפול בשגיאות.

בדוגמה הבאה יוצרים שירות שתמיד נכשל וтопס 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: error
  }
}
main();

```

שירותת `readFile` הוא שירות שתמיד נכשל. הוא מחרכה שנייה ועושה `reject`. הישום שלו הוא בדיקות כמו כל שירות אחר שמחזיר הבטחה ומקיים אותה או לא מקיים אותה. מה שמשמעותו הוא שה-`async` שנקרא `main` זהה לדוגמה הקודמת, למעט ה-`try-catch`. ה-`try` יקרה כאשר הבטחה תקינה (ובדוגמה זו לעולם לא) וה-`catch` יתקיים אם הבטחה לא תקינה (בדוגמה זו תמיד כי עושים `reject`). מה שקרה הוא שמודפסת שגיאה בקונסולה. אפשר כמובן לקבוע כל התנהלות אחרת.

מובן שאין בעיה לקבוע כמה `await` בתוך פונקציית `async` אחת. מה שצורך לזכור, וגם לתרגל, הוא שבסוףו של דבר מדובר ביציפוי של סוכר על גבי ה-`promises`. במקרה להשתמש ב-`then` שעלול להיות קצר מסורבל, משתמשים ב-`async`. **אלו שתי דרכים שונות לעבוד עם שירות זהה שמחזיר הבטחה.**

תרגילים:

בדוק כמו בפרק הקודם, צרו פונקציה `checkIfNumberOK` שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזoor הבטחה מקוימת עם מחרוזת הטקסט `Number is OK`, אם המספר קטן מ-10, תחזoor הבטחה מופרת עם מחרוזת הטקסט `Bad Number`.
התרגיל האמתי הוא למש את הקריאות עם `async`. צרו פונקציית `async` שימושה בשירות, וקראו לה פעמי אחת באופן כזה שהיא תדפיס בקונסולה את מה שהשירות מחזיר אם הוא מצלייח ופעמי אחת באופן כזה שהיא תדפיס 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 הבטחה. אם לא, הוא מפר 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 להכוות לו. אם הבטחה מקיימת, מה שיש בתוך ה-`try` רץ. אם לא, מה שיש בתוך ה-`catch` רץ. עכשו כל מה שנותר הוא לקרוא לפונקציה `checkMyNumber` עם המספר המתאים.

תרגיל:

נתונים שלושה שירותים שונים המוחזרים הבתוחות:

```

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 פונקציית `async` 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!"]

```

הסבר:

ראשית יוצרים פונקציה בשם `callMyPromises`. כדי לסמן שהיא אסינכרונית משתמשים במילה השמורה `async`. ברגע שמשתמשים במילה זו, 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` תتمלأ.

```
results[0] = await myPromiseA();
```

נותר רק לקרוא לשירותים השונים לפי הסדר ולהדפיס את התוצאה. שימוש לב שורירתם הקוד מחוץ ל-`callMyPromises` מתנהלת כרגע, אבל זה לא מאד מעניין.

AJAX

ההגדרה 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/i אינטרנט באמצעות ג'אוּזְקְרִיפְט ופרוטוקול HTTP. זה וושמע מעט מצחיק, כי ג'אוּזְקְרִיפְט היא שפה שהיה בראשת. אבל כשוחשבים על כך לעומק, ג'אוּזְקְרִיפְט לא היה בראשת. ג'אוּזְקְרִיפְט היא שפה שהיה בדף ונטענת מיד כשזהrif נטען. אם התקנתם את סביבת הבדיקה שהסבירתי עלייה באחד הפרקים הראשוניים, אתם יודעים שבכל פעם שמבצעים טעינת דף, קוד הג'אוּזְקְרִיפְט רץ זהה. הקוד רץ עם המידע שמננים לו. הוא יכול לחת את המידע מה-DOM או משתנים שמגדירים לו, אבל מקור המידע אחד – האטר 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 (שעליו למדנו בפרק המוקדש לביטויים כאלה), לחת את הכותרת ולהציג אותה למשתמש. מה שדף או, נכון יותר, המשתמש בדף יכול לעשות, גם ג'אוּזְקְרִיפְט יכולה לעשות. אבל בדרך כלל לא משתמשים ב-AJAX כדי לפנות לאתרם המיעודי לבני אדם, אלא כדי לפנות לכתובות באינטרנט שמחזירות אובייקטי JSON, שקל לעבוד איתם בג'אוּזְקְרִיפְט. אם תפעילו את הדף ותנסו להיכנס לכתובת זו, גם אתם תראו את אובייקט ה-JSON. כאמור, AJAX הוא בדיקת כמו דף שמשגר נתונים, אבל עושים זאת זה באמצעות קוד.

הבה נדגים. יש בראשת שירותים רבים המספקים מידע, למשל אתר המספק נתונים מג אוויר או שערים של מטבחות חז. אלו אתרים המספקים API, זהה ראשית התיבות של Application Programming Interface, המשמעות היא שמי שימצא בהם תועלת הם בעיקר סקריפטים של AJAX. אפשר למצוא רישומי של API כאלה בראש בחיפוש אובייקט public API data sites <https://github.com/toddmotto/public-apis>

אפשר למצוא שם عشرות אתרים המספקים API. כאן נתרgal בשירות שנקרא "בדיקות צ'אק נוריס". השירות אינו דורש הוזהות, 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! פשוט היכנסו ל קישור באמצעות דף רגיל. תוכלם לראות שאתם מקבלים ממש JSON עם מידע.

```
{  
  "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."  
    }
```

毋론 שעובד משתמש רגיל זה קצת בעיתי, אבל לסקריפט שימוש בקשת AJAX זה מעולה. יוצרים בקשת AJAX אל האתר הזה ומציגים את המידע.

נשאלת השאלה "איך?" בג'אוوه-סקריפט המודרנית יותר מקובל להשתמש ב-`fetch`, פונקציה גלובלית שקל להשתמש בה וקיים בסביבת הדפדפן. הפונקציה הזו מחזירה promise שהוא התוצאה. מהתוצאה בוררים את סוג הנתון שרצוים ומהזירם אותו, ואיתו עושים מה שרוצים, ובמקרה של שירות הבדיקה – מדפיסים את הבדיקה.

השימוש ב-`fetch`:

```
fetch('https://api.chucknorris.io/jokes/random')
```

פשוט מאד, נכון? הפונקציה הזו מחזירה promise עם התגובה מהשרת. ב-`promise` למדנו להשתמש בפרקם הקודמים. עכשו צריך רק להחליט מה עושים אם ה-`promise` מתקיים. במקרה זה אנחנו רוצחים את ה-`JSON`, סוג המידע שה-API עובד אותו. אם למשל ה-`API` היה מחזיר תמונה, היה נדרש סוג מידע אחר שלאណון בו כאן:

```
fetch('https://api.chucknorris.io/jokes/random')  
.then((response) => {  
  return response.json();  
})
```

פשוט, נכון? עכשו אפשר לשרשר עוד `then`, שבו מחליטים מה לעשות עם ה-`JSON`. במקרה הזה, מדפיסים 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 בתחום אסינכרוני בכלל ובתכנות מבוסס promises, לא צריכה להיות לכם בעיה בהבנת הסינטקס הזה: `fetch` מחזירה promise עם סוג המידע, ויש לבחור את סוג ולהזיר אותו, אז לעשות בו מה שרוצים.

ethodot Shel HTTP VaRgomentim Nosfim

השימוש שהראיתי עד כה ל-fetch הוא שימוש בסיסי בפורט GET HTTP. פרוטוקול הרשות לא נלמד בספר זה, אך אפשר לציין בקצרה שבפרוטוקול אינטרנט יש כמה סוגים בקשות. הבקשה הנפוצה היא בקשה GET, לקבל נתונים. זו הבקשה שמבצעים כאשר נכנים לאתר אינטרנט למשל. מethodות אחרות הן למשל POST – שבה PUT, (POST, (POST לעדכן נתונים) או DELETE (למחיקת נתונים).

אם רצים לשלוח נתונים (למשל מלאים טופס באינטרנט, הטופס נשלח בבקשת POST, השולחים כשהמתודה מבצעת את הקראיה לשרת.

```
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 שולחים בבקשת POST עם אובייקט אל הכתובת <https://jsonplaceholder.typicode.com/posts> קל לראות שהוא מואוד לביקשת GET, אלא שכן נדרש להעביר ארגומנט שני שմפרט את סוג הבקשה וכמו כן את המידע. באותו אובייקט אפשר גם להגדיר headers למשל. על מנת להבין את סוג הבקשות יש צורך בידע על HTTP שאינו נלמד בספר זה, אבל הפרקтика היא די פשוטה.

כאמור, fetch הוא פשוט מאוד להבנה אם מכירים היטב תכונות אסינכרוני ו-promise.

תרגיל:

נתון ה-API שכתובתו היא <http://jsonplaceholder.typicode.com/posts/1> – הדפיסו את הכתובת המתבקשת מהגישה לכתחובה 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 שמתקובל אובייקט JSON מהסוג הזה:

```
{  
  "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 promise מחזירה fetch שלו תופסים עם .then. ה-.promise מוחזר עם ארגומנט התגובה וצריך תמיד להחזיר את מה שרוצים מהתגובה זו, לעניינו – JSON מחזירים את:

response.json

כיוון שאפשר לשרשר promise, צריך להשתמש בעוד then על מנת להדפיס בו את מה שרוצים, לעניינו – הלקוחות title, שיש להדפסה.

תרגיל:

שלחו בקשה מסוג DELETE והדפסו <http://jsonplaceholder.typicode.com/posts/1> אל URL בקרה של הצלחה.

פתרונות:

```
fetch('https://jsonplaceholder.typicode.com/posts/1',  
{  
  method: 'DELETE'  
})  
.then((response) => {  
  return response.json();  
})  
.then((jsonObject) => {  
  document.write('Delete successful');  
});
```

הסבר:

על מנת לשגר בקשה מסוג `DELETE` נדרש להעביר ארגומנט שני אל `fetch`. הארגומנט הראשון הוא הכתובת. הארגומנט השני הוא אובייקט שיכול להכיל כמה תכונות אבל במקרה זה מכיל רק תכונה אחת: `method`, שהערך שלו הוא `DELETE`. מפה ממשיכים לבדוק כמו ב-`fetch` רגיל. מקבלים את התגובה ומעבירים את ה-`JSON` המתקבל ממנה הלאה. אם מקבלים תגובה מסוג `clash`, סימן שהמתקפה הצליחה.

ES6 Classes

מחלקות, קלאסים, או `classes` באנגלית, הם דרך מצוינה לארגן קוד בג'אווהסקריפט. יש לא מעט שפות תכונות שימושיות במחלקות. באופן עקרוני, בג'אווהסקריפט מחלקות הן פשוט ציפוי של סוכר מעל אובייקט רגיל ופונקציה בנאית, שעליו הוסבר בפרק על האובייקטים. אבל כיוון שמחלקות הן כל כך נפוצות, אפשר להבין לכך, כולל איך הן נראהות כאובייקט רגיל.

מחלקה היא אובייקט שיש לו מפתחות. חלק מהפתחות הם סוג מיידיע פרימיטיבים וחלקם פונקציות (ואז קוראים להן מתודות של המחלקה). הבה נבנה קלאס פשוט כדי להציג 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 בנאית מהפרק על האובייקטים, אתם צודקים. זה בדוק אותו דבר והפונקציונליות היא אותה פונקציונליות, רק שהיכול ארוץ יפה יותר.

ב-`constructor` יש את הפונקציה הבנאית עצמה, זו שרצה כאשר מרים את `new`. בתוך המחלקה מגדירים גם את המתודות.

זה הכלול. בדוגמה אפשר לראות איך יוצרים מהבתנית, שהיא פונקציה בנאית עצם (שימו לב שגם שם המחלקה הוא עם אות גדולה), את האובייקט הפרטי.

מחלקה היא פשוט ציפוי של סוכר או "סוכר סינטקטי" על הפונקציה הבנאית הרגילה שכבר למדנו עליה. אין צורך להיבהל מהamilim השמורות class או constructor מה שלמדנו קודם – זה בדוק מה שלמדנו קודם. המחלוקת המפוררת לעיל בעצם נראית כך מאחורי 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 קלאס יש עוד מתודות מוצלחות, שכאמר עוזרות לסדר את הקוד בצורה טובה ונעימה יותר לעין.

בפרק על האובייקטים דיברתי על get ועל set, וביטה תשmachו לדעת שאפשר להגדיר אותם גם פה. אלא מה? צריך להיזהר מאד מהפניות מעגליות. הינה דוגמה שמתבססת על הדוגמה הקודמת. במקרה זה יש שמביא את התכונה הנדרשת (title) ועוטף 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
```

כאמור, את הקונספט של get ו-set כבר למדנו, וכך מוצגת פשטוט דרך פשוטה יותר ונעימה יותר לעין למש את מה שאתם כבר יודעים. אין פה ממשו חדש. זו הסיבה שמייקמתי את הפרק הזה בסוף הספר. מחלוקת היא לא ודו אלא

פשוט דרך אחרת ונעימה לכתיבת פונקציה בנאית, ואת הפונקציה הבנאית שיזכרת אובייקט עם תכונות ומתחות אתם מכירים.

אם יש כמה פונקציות בנאיות שחוזרות על עצמן, אפשר ליצר פונקציה בנאית שומרישה לפונקציות הבנאיות את כל התכונות שלהן. אם מדובר תכנות מונחה עצמים, אז זה דומה – אבל רק כמעט, כיון שגאווהסקריפט היא לא שפה מונחת עצמים.

הבה נמחיש זאת באמצעות דוגמה מהחיים. יש שני סוגי לקוחות – رجالים ופרימיום. לשני סוגי לקוחות יש תכונות דומות: לשניהם יש שמות וכתוות ומתודה set ו-get, אבל ללקוח פרימיום יש גם תכונת "הטבות". איך ממשיםcosa דבר? אפשר ליצר שתי מחלקות באופן הזה:

```
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` (בתחתית מודגם שימוש בכל אחד מהן). שתיהן המחלקות זותות, 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 ש חוזר על עצמו, וזה בעיה. למה? כי אם יבקשו מכם שינויים במבנה הלקוח (למשל להוסיף תכונה נוספת), הטרכו לשנות את המבנה פעמים. ואם יש יותר משנה סוג ללקוחות, הטרכו לעשות שינויים במקומות נוספים. שכפול קוד הוא משהו שחייב במידת האפשר להימנע ממנו.

בדוק בשביל זה קיים ה-`extend`, שמאפשר ליצור מחלוקת-על ולרשת ממנה את כל התכונות. במקרה הזה – ליצור מחלוקת `client`, `RegularClient`-ו- `PremiumClient` יעתיקו ממנה הכל. במחלוקת `client` 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 הרבה אלגנטי ושאין חזרות. בעצם, `RegularClient` לוקח את `Client` ומضاف לו. ברגע שעושים `extends`, כל המתודות והתכונות של `Client` נכנסות ל- `RegularClient` ו-

אנו. אופן אוטומטי. אפשר להוסיף להן מתודות חדשות.

זה אולי נראה לכם כמו קסם, אבל זה לא. זה בדוק מה שלמדנו בפרק על אובייקטים – רק באוֹרִזָה חֲדֶשָה. אם אתם מתתקשים בתרגילים, כדאי לחזור שוב לפרק על האובייקטים.

תרגילים:

כתבו מחלקה של מכונית המקבלת בפונקציה בנאיות שם, צבע ונפח מנוע. לכל מכונית יש מספר זיהוי ייחודי המורכב מחיבור של הדגם, הצבע והנפח. למשל, אם המכונית היא opel, הצבע הוא white והנפח הוא 1,200, מספר הזיהוי יהיה 1200.opelwhite. צרו למחלקה "מכונית" פונקציה המחזיר 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
```

הסבר:

באמצעות המילה השמורה Class יוצרים מחלקה בשם Car. בפונקציה הבנאית מקבלים את שלושת הארגומנטים הנדרשים ונוצר number. שימושו לב ל-*this* – ברגע שימושים בו, *this.modelNumber* יהיה זמין לכל המחלקה. יוצרים מתודת *get* מסודרת כדי שתחזיר אותו.

למטה משתמשים במחלקה עם הנתונים. אפשר ליצור עכשו איזו מכונית שאתם רוצים. לשם השוואה – הסתכלו על התרגיל הראשון בפרק על *new* ו-*this*. זה אותו תרגיל בדיק והתשובה דומה מאוד לתשובה זו, אלא שכאן השתמשנו בקלאס. כאמור, המילה קלאס והטכנית האלגנטית יותר מבלבולות מתכנתים שמסתכלים על כל העניין של המחלקות כעל וודו, אבל ברגע שבמקרים שהוא ציפוי מעל משחו שכבר היה קיים בשפה – הכל נראה פשוט יותר.

ומה עכשו?

למדנו לא מעט על ג'אוהסקרייפט בספר זהה. התחלנו בעצם ממשתנים בסיסיים ומסוגי מידע פרימיטיביים; המשכנו להאה אל פעולות ואל זרימת קוד באמצעות משפטים תנאי ופונקציות; למדנו גם על מערכיים ואובייקטים ועל LOLAות ושינויי מערכיים ואייך לארגן את המידע; העמכו הלהאה בקוד באמצעות לימוד שמעותי של אובייקטים מוכונים ואיך ג'אוהסקרייפט מתנהלת בסביבת דפדפן; ובפרקם האחרון למדנו איך עובדים בקוד אסינכרוני. זה לא מעט ללמידה, במילויים אם לא רואיתם שפת תכנות קודם לנו.

אם עברתם על כל פרקי הספר כסדרם, יש לכם ידע תיאורטי מבוסס על עקרונות ג'אוהסקרייפט והתחביר שלו. אתם-Amorim לודעת לכתוב קוד פשוט, לנתח קוד מורכב יותר ולפתח בעיות. על אף שהבסיס שלכם יציב וטוב, עדין נדרש תרגול על מנת להזק את הידע שלכם בשפה ולהעшир אותו. אף אחד לא הופך למתכנת בעקבות קריאת ספר, טוב ככל שהיא.

אם אתם רוצים להמשיך ולהרחיב את הידע התיאורטי שלכם, אתם מוזמנים להמשיך לספרי ההדרכה הקיצרים הנוספים שכתבתם: על jQuery, על ריאקט ועל Node.js (ג'אוהסקרייפט מצד השרת). אם אתם לא יכולים לרכוש את הספרים הללו, המדריכים החינמיים שכתבתם ומופיעים באתר שלי: internet-israel.com יעוזו לכם. כדי לעבור על הספרים או על המדריכים עד שתתדרעו את התיאוריה טוב מספיק.

אבל מהתיאוריה צריכים לעבור למעשה ולכתיבת קוד שעושה משהו, וזה הצעד הכי קשה. איך בדיקות עוברים ממצב שבו אני יודע (תיאורתי) לכתוב קוד ג'אוהסקרייפט במצב שבו אני כותב משהו שעובד ושאנשימים משתמשים בו?

הדרך הכי טובה היא לבנות ול כתוב. השאלה היא מה. פה שום ספר או מדריך כבר לא יעזור לכם וזה תלוי בהם ובצרכים שלכם. ג'אוהסקרייפט היא השפה הפופולרית ביותר בעולם ומשתמשים בה בהרבה מאוד מקומות: באינטרנט, באפליקציות דסktop, במובייל, בשורתים ואףלו לצורכי אבטחת מידע. אחרי שלמדו את אחד מהנושאים האלה או את כולם, כדי להתנסות בקוד ממשי.

המקום הטוב ביותר להתחיל הוא באמצעות תרומות קוד לפרויקט קוד פתוח שאפשר למצוא ב-GitHub. עבודה בסוגרת פרויקט קוד פתוח זה יכולה להעשיר ולהעמיק מאוד את הידע שלכם. בחרו פרויקט מבוסס ג'אוהסקרייפט ונסו לתרום לקוד שלו. באמצעות פתרון באג שמתוויד בפרויקט, באמצעות תוספה לדוקומנטציה, לבדיקות האוטומטיות או אףלו באמצעות הוספה תכונה מסוימת. נסו להתקין קומפוננטה ריאקט על אתר של חבר, לבנות תוסף קטן לjQuery עבור האתר של החברה עליו, לתרום לדוקומנטציה של קומפוננטה שאתם אוהבים במיוחד. במיוחד, להגיע להאקטון ולהצטרף לצוות קים או אףלו רק לשבת ולהסתכל על מתכנתים עובדים. אפשר אף רצוי להתנדב בעמותה ולהתנסות בעבודה בפרויקט ממשי שאנשימים השתמשו בו. יש הרבה דרכים – איך דרך תבחרו? זה תלוי בהם.

יש לא מעט מיטאפים ופתרונות של מתכנתים שצדאי להגיע אליהם. בפגשים האלו פוגשים מתכנתים שעובדים בחברות מגוונות, והאויריה נעימה מאוד. אפשר ורצוי לשאול את האנשים האלו איך מתקדמים הלאה בלימוד. יש גם קבוצות פיסבוק בנושא, שהאנשים הנחמדים והמקצועיים בהן יכולים לסייע לגבי המשך הדרך.

באתר זה יש רשימה של מקורות ושל קבוצות שאני משתמש להיות חבר בהן, אני ממליץ לכם להסתכל בה ולבדוק את הקבוצות ואת המקורות השונים: <https://github.com/barzik/web-dev-il-resources>

כך או אחרת, חשוב מאוד להמשיך לכתוב בשפה. כאמור, שפת תכנות אינה שונה מהשפה מדוברת; אם לא תשתמשו בה, תשכחו אותה. אפשר להשתמש בג'אוויסקריפט במגוון עצום של פרויקטים – רק צריך לבחור פרויקט ולהמשיך לתרגם. להשתלב בתחום ההיבט זה לא בשמיים וזה אפשרי. אם אני הצלחתי – כל אחד יכול.

נספח: Best Practices

מחברים: שחר טל, רוני אורבך, דניז רוזג, נופר ברנס, חברת Really Good

מה זה **Best Practices** ולמה כדאי לישם אותם?

הם כלליים, המלצות ומוסכמות שנועדו לסייע לייצור תוכנות איכותיות, בין השאר באמצעות כתיבת קוד בצדקה טובה ונכונה שמזערת טעויות ובלבול.

תוכנות הוא פועלה חברתית. כשהושבים על תוכנות, לרוב מדינינים האקר בודד שישוב בחושך בקפוץון ומוביל ב思绪 בלי לחשב שנייה, כאילו הקוד עף מתוכו. הסיטואציה הזאת היא נדירה, ובינינו, זה די לא נוח לשבת להנאה כלה. לרוב תוכנות נכתבות עם קולגות במקום העבודה או עם חברים או אפילו עם אנשים זרים באינטרנט – והיצירה שלחן דורשת קצת יותר סבלנות ושיקול דעת. מתכנתים אחרים יסתכלו על הקוד לפני שהוא יוכנס באופן סופי לפרויקט (אדרט על **Code Review** בהמשך), ומתכנתים אחרים יצטרכו להבין ולעבוד אליו בעתיד כשירצטו להוסיף, לשנות או לתקן את התוכנה.

לאורך הזמן מתכנתים עוזבים צוותים ואחרים צריכים להתמודד עם קוד שהכותב שלו כבר לא בסביבה, וגם אתם בטח תרצו להשאיר אחריכם קוד שמתכנתים אחרים יוכלו ואיפילו ישמשו לעבוד איתו. אפילו בפרויקט שככלו 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 בעצםכם. לכן חשוב לשמור על קוד מסודר ומאורגן שנכתב בדרך מסוימת. כך מצמצמים באגים וטעויות, מקלים על כולם את קריאת והבנת הקוד ב מהירות וחוסכים רעש רקו וויכוחי סרק על שיטויות – מחליטים פעם אחת על סגנון ומדיניות ונמנעים מ"מלחמות עריכה" אינסופיוות שבהן כל אחד מסדר את הקוד איך שהוא והוא אחורי מבטל את השינויים.

אפשר להסכים על כלליים כלליים בעלי בעלה, אפשר בכתב, ויש כלים שадון בהם בהרבה ממש כאן, שיכולים לעזור לפחות לפחות חלק מהכללים באופן אוטומטי.

במיוחד בשפה גמישה כמו ג'אווהסקריפט, שמאפשרת לעשות המון דברים יוצרתיים ובתוכם המון שיטויות, מומלץ לעקוב אחרי **Best Practices** מקובלים בתעשייה, שאולי לא נכתבו בדם אבל בהחלט עלו ביעז ובධמות. מעבר למניעת שגיאות ולשימוש מושכל באפשרויות שהשפה מציעה, חלק מה-**Best Practices** הם העדפות סטיליסטיות של המתכנתים בפרויקט, שהתקבעו ושהצווות רוצה לשמר – בשני הסוגים יש מקום לשיקול דעת בריא, ולא צריך לקחת הכל כторה מסיני בלי להבין ולבחון מה מתאים לכם.

דוגמה להמליצה שמנועת טעויות היא תמיד להשתמש בהשוואה קפדנית, ככלומר ב-`==` או ב-`!=` במקום ב-`==`. (type coercion).

לא מומלץ:

```
if (numberOfThings == possiblyNotANumber) { /* code */ }
```

מומלץ:

```
if (numberOfThings === possiblyNotANumber) { /* code */ }
```

דוגמה לכל סובייקטיבי יותר, שפושט מבטא העדפה סגונית ועזרה לשמר על אחידות, היא ריווח בין חלקים קוד שונים, למשל ריווח בין שם פונקציה לבין הסוגרים שגדירים את הארגומנטים:

אופציה א':

```
function jump(x) { }
```

אופציה ב':

```
function jump(x) { }
```

אופציה ב' היא המומלצת כי היא מאפשרת להבדיל בחישוף בין הגדרת הפונקציה לקריאות אליה.

סת הכללים והמלצות יכול להיות שונה מאוד בין פרויקטים שונים, ובעבודה צוות יש חשיבות בשם העקבות והשפעות להתגבר על העדפה האישית שלכם ולכבד את מה שכבר נקבע – מומלץ לדבר בפתחות ולבחון ביחד צורך בשינויים.

שכל מפתח מקצועני צריך להכיר Best Practices

בחירת שמות

משתנה או פונקציה עם שם טוב יאפשרו לכם להבין מיד מה הפקיד שלהם. אם השם לא מוצלח תצטרכו לראות איך הגדירו את המשתנה ומה הוא מאחסן בתוכו או לקרוא את הפונקציה ולפענה מה ניסו לעשות בה. פתרון אפשרי הוא להוסיף הערה, אך עדיף פשוט לבחור שם טוב. שם טוב הוא קצר וקובל יותר מהערה, מעבר לכך שモটב לא לפזר הערות ליד כל מקום שבו משתמשים במשתנה או בפונקציה.

איך בוחרים שם טוב? שיטה מקובלת עבור פונקציות היא שיטת ה-verb-noun (פועל-שם עצם).
לדוגמה:

- `getStudents`

- `sortColorsByBrightness`
- `setLunchBreakTimer`

במקרה של `setLunchBreakTimer` לדוגמה, שעה טובה יותר להיות כללי מדי, וכשנתקלים בו צריך להזכיר באיזה טיימר מדובר.

יש גם יוצאי דופן בולטים – שמות משתנים סטנדרטיים שצורך להזכיר ואין צורך להסביר או לפרט אותם. דוגמה לשמות כאלה הם `x` ו-`y` בהקשר של פיקסלים או ציונים על המסך, או `b` ו-`r` בשימוש הקלטاسي שלהם בלאוות. אבל חוץ מהם, כדאי לזכור שכמעט תמיד משתנה של אות או שתיים זה רעיון רע. אין שום יתרון בראשית תיבות מסתוראים על פני מילים מובנות.

שם המשתנה הוא לא הדבר היחיד שעוזר לкриאות הקוד. אפשר לכתוב אותו שם ככמה דרכיהם בעזרת אותיות גדולות וקטנות. כל דרך כזו נקראת "קיס", ומיעודת לייצג סוג המשתנה אחר. אולם השם זהה, אבל על כל אחד מהמשתנים האלה נוסף רובה של משמעות שנרמזו רק מבחירה הקיס:

camelCase
`lunchBreakTimer`
 כל המילים מלבד הראשונה מתחילה באות גדולה; אין רווחים בין המילים.
 קיס זה נמצא בשימוש עבור כמעט כל המשתנים והפונקציות בג'אווהסקריפט.

PascalCase / TitleCase
`LunchBreakTimer`
 כל המילים מתחילות באות גדולה; אין רווחים בין המילים.
 קיס זה נמצא בשימוש עבור קלאסים, וככה מקל את הבחנה בין `instance`-ל-`const`.

SCREAMING_SNAKE_CASE / UPPER_CASE
`LUNCH_BREAK_TIMER`
 כל האותיות גדולות; רווחים בין המילים הופכים לקו תחתון.
 קיס זה נמצא בשימוש בעיקר עבור קבועים, שבדרך כלל יוצבו במשתני `const`.

kebab-case
`lunch-break-timer`
 כל האותיות קטנות; הרווחים בין המילים הופכים לקו מפרד.
 נמצא בשימוש ב-HTML וב-CSS עבור שמות של `id`, קלאסים ואלמנטים.

קיים זה לא נוח לשימוש בג'אווהסקריפט כי קו מפ прид אינותו חוקי בשמות משותפים, אך שקשה עד בלתי אפשרי להשתמש בו ברוב המקרים.

snake_case

lunch_break_timer

לא בשימוש בג'אווהסקריפט בדרך כלל.

KISS

"אמירה שקלעה בול כשנתבעה לפני יותר מחמשים שנה על ידי מהנדס המטוסים קליב ג'ונסון בלקהיד, ועודין מתאימה כהנחיה למתכנתים של 2020 והלאה. כתבו את הקוד בדרך הפשוטה והקצרה ביותר, כל עוד הקריונות נשמרת. למשל, הימנו מסינטקס אזוטרי שלא נמצא בשימוש רחב ומהתחכחות יתר בכלל."

DRY

אין סיבה לכתוב את אותו קוד יותר מפעם אחת. לעיתים זוקקים לאותה פונקציונליות בכמה מקומות אבל מהסיטים להפוך אותה לפונקציה מכיוון שמדובר בקוד פשוט וקצר או בקוד שדורש שינוי קטן בכל שימוש שלו.

אל תהססו. כשעוטפים את הקוד בפונקציה גם נתונים לו שם, שהופך את השימוש בו לкриיא יותר. ככל שימושים יותר קוד שלא לצורך, מגדלים את הסיכון שהמתכנת הבא יפספס חלק מהמקומות בהם יש קוד כפוף, והתזוזקה תהפוך לקשה ורצופת תקלות.

זכרו – הפק מ-DRY הוא WET, ראי תיבות של (-; Write Everything Twice

לא להמציא את הגלגל

למרות שתכנות זה כיף ומעניין, לא בכל דבר שבונים צריך להקים הכלול מאפס. וכמו שכנהרא ברור כיום שתשתמשו בספרייה כלשהי שתעזר לרנדר את המשק, נניה React, ככה חשוב להשתמש גם בספריות ובכלים קטנים ומוקדים יותר כזו הגיוני. מהו זאת בקשות לשרת עם axios, דרך ספריות לפעולות נפוצות על מערכיהם ואובייקטים כמו lodash ועוד לאלפי הרכיבים המוכנים לרוב חלקי המשק שתרצה לבנות – תהיו בטוחים שמיישחו כבר פיתח פתרון טוב כמעט לכל צורך. הבה נזדה באמת – אתם לא הראשונים שcottובים Color Picker או

Dropdown עם השלמה אוטומטית, וגם אלגוריתמים לערכוב רשימות כבר נכתבו בעשורים שקדמו להצטרפות שלכם לתהום.

- יש שלושה מקרים שבהם דזוקא הגיוני להיכנס לעובי הקורה ולכתוב בעצמכם משהו לא אלמנטרי:
1. אם אין פתרון מוכן שעונה על כל הדרישות שלכם (עדין זכרו שליעיתם יש רכיבים קטנים יותר בתוך המכלול שיוכלו לעזור לכם או שתוכלו למדוד מהם, אז תמיד טוב להסתכל עליהם).
 2. אם זה ממש התחום שלכם, מה שבשבילו קמתם בבודך ואתם תרצו לקחת אותו כל כך רחוק שכנראה אף כלי מוכן לא יספק לכם – למשל במסגרת העבודה *Really Good* אנחנו עובדים עם פלטפורמת וידיאו מוביילית, אז כשהתבקשנו לפתח נגן וידיאו חדש יצרנו אותו מאפס על בסיס תגית ה-video של HTML, בלי אף ספרייה חיונית לניגון וידיאו.
 3. אתם רוצים לתרגל או נבחנים על פיתוח עצמאי של משהו זהה. בהצלחה!

בשולוי הדברים, שימו לב לרישיונות בקוד פתוח – רישיונות שונים מאפשרים שימוש בתנאים שונים, וחלקם מגבלים השימוש מסחרי. אם אתם חלק מארגון גדול ולא בטוחים, כדאי לוודא שהשימוש מקובל על הגוף המשפטי בחברה. בכל מקרה תמיד טוב לקרוא ולהכיר את הרישיון של הקוד שימושים 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 בצורה מושלמת בمرة אחת. מתחילה מגרסה בסיסית שפלה עובדת עם הfonקציונליות הבסיסית. אחר כך חוזרים לעובות ולכסות את כל הדרישות, ומשפרים את הקוד כך שהיא קרייה יותר, תמציתית ומובן ככל האפשר. אל תפטו להקדים את המאוחר ולהתוא ב" – אופטימיזציה שומרים לסוף. "Premature Optimization

תיעוד

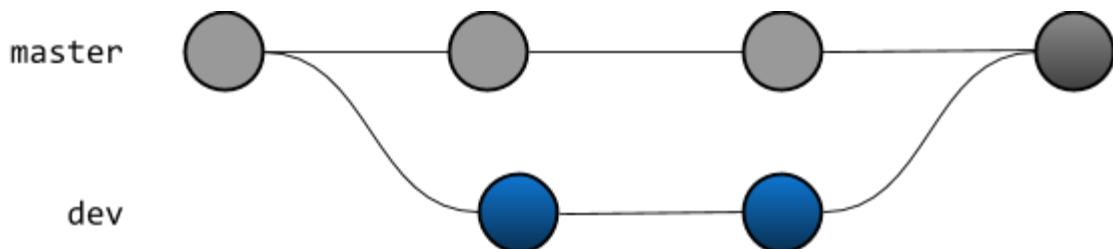
- כבר למדתם מוקדם יותר בספר על העורות בקוד. שני השימושים הכי נפוצים בהערות הם:
4. כינוי זמני של חתיכת קוד בלי לאבד אותה. נוה תוך כדי פיתוח ובדיקות, אבל שימו לב שהערות הן לא שיטה לשמר גרסאות של קוד – מיד אצלול לניהול גרסאות נכון.
 5. תיעוד – להסביר מה מצפים שיקריה או למה פיתחتم משהו בצורה מסוימת. תיעוד טוב מוסף מידע, ולא סתם חזיר על מה שהקוד עושה. אפילו כמה מילوت הסבר עדיפות על כלום. הערה שכתובה טוב יכולה לחסוך למתקנת אחר הרבה כאב ראש. כשאתם ניגשים לכתיבת התיעוד, נסו להשוו מה הדבר hei מרכיב או לא ברור בקוד שלכם ושימו את הדגש שם. מתכוונים אחרים ידועו לכם, ואפילו אתם תודו לעצמכם כתחוירו לעבוד על קוד יישן ומתועד היטב שלכם.

ניהול גרסאות

בגלל שקוד הוא בסופו של דבר קובץ טקסט שנערכים על המחשב שלכם, הפתרון הכי פשוט (ונאיבי) לניהול גרסאות הוא יצירה עותקים של תיקיות בנקודות זמן מסוימות. במצב זה – קל לאבד קוד וקשה לעבור בצוות. המחשב יכול לשבוק חיים או שתצטרכו לשחזר קוד שמחקתם וזה יהיה קשה עד בלתי אפשרי. לא רק זה – אם מישו מהוצאות ירצה לשנות קוד הוא יצטרך לברר אצל מי הקובץ החדש ביותר ולבקש ממנו שישלח לו את הקובץ. באיזו שנה אנחנו?

כדי להתגבר על כל הבעיה הללו, משתמשים בפתרון ניהול גרסאות (Source Control Version Control) או הפתורונות הנפוצים ביותר כיום הם Git ו-SVN. לגיט יש כמה יתרונות משמעותיים ובראשם מהירות, ביוזר שמאפשר יכולת עבודה נוחה ללא חיבור לאינטרנט וקיילה עצומה – כמעט כל עולם הקוד הפתוח היום מתנהל על בסיס Git, לרובם github.

שימוש בניהול גרסאות מאפשר למפתחים רבים לכתוב קוד בכמה ענפים (Branches) שונים במקביל ולשתף אותו ביניהם. אפשר לראות איזה מפתח כתוב מה ומתי, אפשר לשלב כמה ענפים מוכנים אל תוך גרסה אחת שאויה משחררים, והכי חשוב – אפשר לחזור אחריה במידת הצורך.



בקשת עזרה

כשנתקעים במהלך פיתוח – לבקש עזרה זו לא ברושה. אם אחרי Debugging, בידוד הבעיה ובדיקת הלוגים (אם יש כאלה) עדין לא צלחتم את הבעיה, עוברים לגוגל. מנסים לחפש את השגיאה בצורה הכי נקייה שאפשר, בלי שמות קבצים או מספרי שורות מהקוד הספציפי. לרוב הבעיה נפוצה וכמעט תמיד אפשר למצוא פתרון או לפחות כיוון או גישה שיוכלו לעזור.

אם יש מתכנתים נוספים בצוות אפשר לשאול אותם. מתכנתים מנוסים או כאלה שנמצאים בפרויקט כבר תקופה מכירrim את הבעיה הנפוצה והאתגרים שאתה עשוים לחשקל בהם ויכולו להכוין אתכם ולהסוך לכם זמן יקר. שיתוף הבעיה עם חברי הצוות גםגורם לעבור על הקוד פעמי נספה ואפשר למצוא את הפתרון 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, שמנגנון דירוג התשובות בו עוזר לזהות מה מבין התשובות עובד ורלוונטי בעיני הקהילה, בניגוד לפורומים שם הרלוונטיים אך בחיפוש פתרונות בהם נדרש קריאה יותר ביקורתית. אין מה להתבונש, השאלה שתשאלו תהיה כנראה רלוונטית למתכנתים אחרים בעתיד שייתקלו בהאותה בעיה ויחפשו תשובה.

בדרך לפרסום שאלה תצטרכו להכין דוגמה מוקצת וمبודדת של הבעיה (מכונה בדרך כלל **Reduced Test Case** (השאלה שעצם ההשערה בו עשוי להוביל אתכם לפתרון עוד לפני שתפרנסמו משהו. וגם זה תהליך מומלץ שעצם ההשערה בו עשוי להוביל אתכם לפתרון עוד לפני שתפרנסמו משהו.

ביקורת עמיתים – **Code Review**

סקורי הרצה כללים, וזה יכול להיראות מאיים. האם באמת מקפידים על כלום בכלל צוות פיתוח? התשובה היא שאמן לא מקפידים במאה האחים, אבל מפתחים וארגונים מקצועיים יתעקו לעמוד בכמה שיותר מהם, כי מוצרים הולכים ומפתחים וככל שפרויקט מתעצב ונוסף עוד שכבות של קוד כך נהיה קשה לתחזוק אותו ולהבין מי נגד מי. אם מראש היסודות רעוים אף אחד לא מקפיד על Best Practices, מהר מאוד תמצאו את עצמכם מול "קוד ספוגטי" (שמעורבב כמו ספוגטי בצלחת) שקשה מאוד לתחזוק.

אם זכיתם לעבוד בצוות ולא בלבד, Code Review עשוי להיות חלק מהאחריות שלכם, וזהמצוין. העבירו את הקוד שלכם לבדיקה אצל מתכנת אחר והימנוו מהסבירים מיותרים. תנו קצת רקע כללי על ההקשר של הקוד ומה ניסיתם להשיג ואפשרו לו קוד לדבר בעד עצמו. כך, אם חסר תיעוד – זה יבלוט. למי שעבוד בלבד, יש אתרי אינטראקט וקובציות שבהם אפשר להתייען, לשתף קטעי קוד ולבקש ביקורת.

את הערות על הקוד מקבלים ונוטנים בשתי דרכים. הדרך הראשונה היא בעלפה (פניהם אל פניהם או בטלפון). הדרך השנייה היא בכתב, בצווד לשינויים במערכת ניהול הגרסאות. לדוגמה, אם עובדים עם Git, אפשר להשאיר הערות על Pull Request. בקשה זו היא בעצם בקשה דרך מערכת ניהול הגרסאות לאשר את הקוד ולמזג אותו לתוך הענף הראשי של הפרויקט. מתכנתת שבודקת קוד תוכל להשאיר עליו הערות, ולאחר שהמפתח יבצע את השינויים היא תוכל לאשר את הבקשה כדי למזג את שני הענפים.

כשאתם מקבלים ביקורת, קבלו באהבה הצעות, הערות ושאלות. אל תייחזו את הביקורת כתקיפה אישית נגדכם. זכיתם בעוד זוג עניינים שייעברו על הקוד שלכם, לא משנה למי מהם יש יותר ניסיון. קחו אותה בשתי ידיים, הקשו בהamat ואל תרצו למגננה – אפילו אם עבדתם קשה על משהו ואולי קצת התאהבתם בפתרון מסוים או נקשרתם אליו, או אם הערה שקיבלתם תדרוש שינוי יחסית גדול והרבה עבודה מצדכם. זכרו, אתם לעבור ביחיד וליצור קוד איזוטי וברור, לא להראות מי יותר חכם וצודק.

כשאתם עורכים למשהו אחר – נסו להבין מה קורה בקוד ולהעירך אם תהייו מסוגלים להרחב או לשנות אותו במידת הצורך. בוחנו את הקוד אל מול עקרונות מנהים כמו כל אלו שמצורים בפרק זהה. האם השמות הגיוניות וברוריהם? הפונקציות או הקבצים ארוכים מדי? יש קוד שחזר על עצמו והיה אפשר להפוך ליותר [DRY](#)? הקוד מסודר ומעומד בצורה עקבית וקריאתית? וכן הלאה.

בסוף דבר, ככל יום לומדים משהו חדש. Code Review היא דרך מצוינת ללמידה מהאנשים סביבכם ולהמשיך להפתח בתור מתכנתים. זה הרgel מעולה שיחד אתכם כמתכנתים לזהות קוד פחות טוב ולכתוב קוד יותר טוב. **כולם מרוויחים.**

Tech Design

כשניגשים למאזן פיתוח רציני, סופ' מעשה במחשבה תחילה. כמה שמנסים לקבל באהבה ביקורת, כמעט תמיד מתחבאים לשימוש בסוף התהליך שהוא עדיף להשתמש בספרייה מסויימת שכבר קיימת ולא ידעתם עליה או שאי-אפשר ללכט על פתרון שישים לפתח משיקולים שונים שלא הכרתם. לעיתים אפילו תיתקעו עם הפתרון הלא-מושלם שלכם כי מאוחר מדי לחזור אחורה ולתקון.

כדי להימנע ממצבים כאלה, ראוי לעשות תכנון טכני, שהוא מסמך בפורמט גמיש. יש אזורי התמחות שבהם הגיוני להשתמש בתרשיימי זרימה (למשל תכנון של ארכיטקטורת שרת או מסדי נתונים); כשעובדים גם לצד לקוח וגם לצד שרת מקובל לתאר את ה-API שימושים לבנות לקריאות בינהם – لأن שולחים כל בקשה, אילו פרמטרים מעבירים ואיך בדיקות תוראה תגובה טיפוסית.

בעזרת Tech Design טוב אפשר לוודא שמבנים זה את זה ולהתחליל לפתח את הצדדים במקביל, ולא לגלוות בסוף שכל אחד תכנן מבנה נתונים שונה לחלווטין ולהתוווכח מי צריך להתאים את עצמו לאחדר.

את התכנון זהה תעבירו לביקורת עמיתים, דברו עליו וצאו בדרך רק אחרי שהסכמתם על הפרטים – ככה תמנעו לפחות חלק מההפתעות הגדולות והיקרות בשלבים מאוחרים יותר.

Best Practices ואותומציה

או כתבתם Tech Design וקיבלתם אישור לצאת לדרכ, עבדתם קשה ואוז הגעה ערמה ענקית של הערות ב-Code Review. איך בכלל מתחילה לעبور על זה? ועם כל הכאב וההערכה, איך לא לזקחים את זה קשה כשאלוי אתם מרגיעים 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 הקפdim על מהهو מהכללים המומלצים. זה טבעי, אבל כשמכפילים את הנטייה האנושית לעגל פינות ולפספס פרטים במספר אנשי צוות ובכמה חודשים פיתוח, האיכות מידדרת מהר מאוד.

אחד הדברים שהכי עוזרים בראיכוך המעמך של הביקורת, ובכלל בשמיירה על איכות על ידי איתור ומונעת שגיאות מראש, הוא אוטומציה. יש כלים אוטומטיים מעולים, שאציג בהמשך, שעוזרים לשמר על סדר ולהקפיד על הכללים שתקבעו לעצמכם, בלבד או כצוות.

כך, כשהתגינו ל-Review תדעו שבחלק נכבד מההערות פשוט לא תתקלו, כי כבר קיבלתם פידבק מיידי או תיקון אוטומטי בזמן אמיתי. זה כבר מאחוריכם, וחסכתם גם מהעמיתיים וגם מעצמכם דיוונים על ריווחים, שורות ארוכות מדי ועוד המונן הערות, מהותיות יותר או פחות. יהיה קל, מהיר ונעים יותר להתמקד ולהקשיב לביקורת שכן תגיע כשהיא תהיה מצומצמת יחסית.

בצווות, מומלץ מאוד להחליט פעמי אחת (МОКДם ככל האפשר) על הקווים המנחים שלכם, להיעזר בכלים האוטומטיים שתלמדו עליהם בחלק זהה וליצור מערכת שתפקידה על הקוד שנכנס לפרויקט. למרבה השמחה, מתכנים אחרים כבר בנו מערכות כאלה, ואף נתנו לפועלות האוטומציה של פיקוח על הקוד שם – Linting.

מה זה linting ?

המילה *lint* (לינט) באנגלית פירושה מזוק – הלכלוּך המצתבר בכיס של בגדי הבדורים הקטנים שנוצרים על גבי הסרג' לאחר כביסות. בעת כתיבת ועדרון קוד מתווספים לאט לאט שגיאות או דפוסים לארצוים אחרים שנוגדים את ה-Best Practices שקבעתם – זה המזוק, וכך יש לעבור על הקוד עם כלי שمسיר את המזוק; בהשאלה, קוראים לו linter. לינטרים יודיעים לזהות את השגיאות והדפוסים שהגדרתם, להתריע על הימצאותם ולפעמים אפילו לתקן אותם אוטומטית.

בעולם התוכנה, הלינטר הוא כלי שתפקידו לבדוק את הקוד שכתבתם ולהזהיר משגיאות, באגים או טעויות סגנוניות. הלינטר הראשון פורסם ב-1978, בזק קוד שנכתב בשפת C וונעשה בו שימוש לבדיקת מערכת ההפעלה Unix.

עורכי טקסט מודרניים המיעדים לכתיבת קוד ג'אווהסקריפט מפעילים בודק שגיאות מבנה שיזהיר את המשתמש משגיאות שימנעו מהקוד לרוץ בצורה תקינה בדף. אם כן, למה עדין צריך לינטר? קוד יכול לפעול באופן תקין ועדין לא לעמוד ב-Best Practices ובכללים שקבעם מסיבות שונות, וכך רוצים להריץ על הקוד גם לינטר שיזהיר מחריגות וטעויות מכל הסוגים.

```

function multiply(x){
    ... return x * y;
}
multiply(2,3);

```

'y' is not defined. eslint(no-undef)
Missing semicolon. eslint(semi)

לינטר יכול להזהיר מפני בעיות כמו שימוש במשתנים שלא הוגדרו, קריאות לפונקציות שאין קיימות, הפרת מוסכום בקשר לרווח וסידור של קוד, שימוש בפייצ'רים של השפה שונים להוביל לביעות אבטחה ועוד. הלינטר יסמן בעיות באמצעות סימן בקו מזגゾג אדום לשגיאות וירוק לאזהרות, בדומה לסימון של שגיאות כתיב במעבדי תמלילים. בעבר עבר מעל הקוד המסומן אפשר היה לראות את הסיבה ולעתים גם הצעות לשיפור הקוד.

הכלי הראשון בעולם הג'אוועסקרייפט שעשה זאת היה **JSLint**, שאכן סט מוגדר מאד של כלליים בצורה קשוחה – מי שלא אהב את הדעות החותכות של היוצר שלו, או שיתת הג'אוועסקרייפט דאג'לס קראוקפورد, נאלץ להשתמש בכלוי אחר בשם **JSHint**, שכן אפשר יותר גמישות.

עם השנים וההתפתחות המהירה של ג'אוועסקרייפט, **ESLint** של ניקולס זאקאס חפס תאוצה, בין היתר בזכות מבנה מאד גמיש. ESLint מאפשר להגדיר ברמה פרטנית את הערכים הרצויים לכללים קיימים (כמו למשל אורך שורה מקסימלי או איפה לבדוק מותר או אסור להשאיר שורות ריקות), אבל גם להגדיר כלליים משלכם, ולהשתמש בקלות בכללים אחרים יצרו, כמו שארחיב עוד מעט. נכון לעכשיו, ESLint הוא ה-**Linter** הפופולרי ביותר עבור ג'אוועסקרייפט, ולכןו אטמקד בו.

ESLint

עורכי ג'אוועסקרייפט מודרניים יודעים להציג הערות של ESLint בצורה אוטומטית או בהתקנה פשוטה של נוספת. האזהרות והשגיאות שיוצגו נקבעות על פי קובץ הגדרות, בדרך כלל בשם `eslintrc`, בתיקייה הראשית של הפרויקט. אם אצלכם עדין אין קובץ כזה, קראו על `init` -- -- ESLint וצרו באמצעותו קובץ הגדרות משלכם – מומלץ להתחילה אחד הסטים הקיימים שיוצאו לכם במהלך האתחול.

אם בחרתם סט חוקים שונה لكم להשתמש בו אבל יש בו כמה חוקים שלא נראהים לכם, אל תהססו להיכנס לקובץ ההגדרות ולשנות אותם.

בחלק הבא אעבור על כמה Best Practices, חלקם חדשים וחלקם כבר הזכרתי קודם, לצד הכלל ב-**ESLint** שמשמעותו להם.

רישימה של Best Practices והחוק הרלוונטי של ESLint

בלি מספרי קסם

שם הכלל: no-magic-numbers

"מספרי קסם" הם מספרים שמופיעים בקוד בלי הסבר מפורש, כמו הצבה למשתנה שהשם שלו יכול להאיר עינינו. הצורך בולט בחישובים, וביחד בחישובים שימושים שמשלבים כמה מספרים. ברגע הכתיבה ברור איך מגיעים אליהם ויום אוחודש אחר כך כבר מגדירים בראש ולא מבינים מה הסיפור של המספר הזה.

דוגמה לקוד בעייתי עם מספר קסם:

```
setTimeout(ringBuzzer, 180000);
```

למה 180,000 בפועל? בלי הערה שתספר למה התכוון המשורר, קשה לקלוט במה מדובר במה מזובר. אפשר להוסיף העירה כמו "now three minutes from now", אבל בילדייה לוקח זמן להבין מה אומר המספר זהה ואיך לשנות אותו אם בקשו מכם להאריך את המתנה לחמש דקות. מחשבים מהירים מאוד בחישוב, לא צריך לרוחם עליהם וללעוס עבורם את המספרים מראש. חשוב יותר שהקוד יהיה מובן לכם ולשאר בני האדם שיצטרכו להבין את הקוד, אז מפרקם 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 קפדןית: ===

שם הכלל: eqeqeq

זה הכלל של לנדרים בג'אוהסקרייפט, עוד מ-JSLint המקורי. ג'אוהסקרייפט מאפשרת להשוות בין ערכאים מסווגים שונים. זו תכונה מעניינת שמאפשרת נמיות ויכולת להיות שימושית, אבל היא יכולה עלולה לבלב ולהטעות.

כדי להימנע מטעויות קשותiae לאיתור כתוצאה מההשוואה המתירנית, מקובל ומומלץ להקפיד על שימוש בהשואות הקשוחות יותר. מומלץ להשתמש ב-== לבדיקה שוויון וב-!= לבדיקה אי שוויון. כך משווים בין ערכים בלי להמיר את סוג המשתנים חוץ כדי השוואה.

קוד לא נגיש

שם הכלל: no-unreachable

תוך כדי debugging ו-refactor, קורה שנשארת פקודה שיווצאת מקטע הקוד בשלב מוקדם, כמו `throw`, `return` ו-`return true`. אם מופיע קוד אחריו פקודה כזו, בדרך כלל מדובר בטעות. בדוגמה הבאה, כל מה שאחרי `return true` לעולם לא ירוץ, ו-ESLint תציג את הקטע הלא נגיש:

```
function validateEmail (email) {
  return true;
  if (email.length) {
    let result = true;
    // if (!email...) {}
    // TODO: complete validation
  }
  return result;
}
```

חלוקת לחלקים קטנים

שמות הכללים: max-lines, max-lines-per-function, max-len

קשה לתפוס הרבה לוגיקה ברכף אחד, וכמו שכחיתבת ספר או מאמר מומלץ להשתמש בנדיבות בפסקאות, כתורות ועמודים, כך גם בקוד חשוב לא להתפתות להעmis המון קוד על שורה ארוכה מדי, פונקציה אחת שעשויה הכל או קופץ אחד של אלפי שורות.

החוקים הבאים הם חוקים שאין עליהם מוסכמות, אך הם יכולים לעזור לכם בכך שיישמשו תמרור אזהרה שהקוד שאתם כתובים הפך לארוך ומסורבל. אם תבחרו להשתמש בהם כדי שתתגינו להסכמה עם חברי הצוות שלכם בונגעה גדולים שמקובלים על כולם.

אורך שורה (מספר תוים מקסימלי בשורה)

יש להקפיד על אורך שורה שאינו עבר מספר תוים מסוים. הסיבה לכך היא שרוצים שורה שנכנתה ברוחב המסך ללא צורך לגולץ הצדה בעת קריאת הקוד. קוד קרייא הוא קוד שאפשר לראות את כלו או את רובו בביטחון. בעבר, כאשר המסך היה קטן יותר, היה מקובל להגיד אורך שורה של 80 תוים, אבל בימינו, כשהמסכים הרבה יותר גדולים, נהוגים אורכי שורה של 120 ואף 140 תוים.

אפשר להימנע משורות ארוכות בכמה דרכים. הסיבות העיקריות לשורות ארוכות הן רשיימה ארוכה של תנאים וקוד מקוון. רשיימה ארוכה של תנאים קל לשבור לכמה שורות. קוד מקוון יותר קשה לתקן, אך אם דואגים לשמור על פונקציות קצרות ופשטות מצליחים למנוע זאת ברוב המקרים.

אורך פונקציה (מספר שורות מקסימלי בתוך פונקציה)

אין מוסכמה בנוגע לאורך פונקציה למרות שכולם מסכימים שפונקציות ארוכות זה רע. החוק הזה טוב בעיקר כדי להזכיר לכם שהגוזמתם, لكن אם תשתמשו בו כדאי לבחור מספר שונה לכם. אם איןכם בטוחים איזה מספר לבחור, התחלו עם מספר השורות שנכנסות לכם בסיס ללא גלילה.

עוד דרך לשמר על פונקציות קצרות היא לכתוב פונקציות קצרות שיש להן תפקיד אחד בלבד. רמז לכך שפונקציה עשויה יותר מדי אפשר לקבל כשתנסו לתת לפונקציה שם. אם במהלך הניסיון תגלו בשם הפונקציה את המילה "and", תבינו שניסיתם להכניס יותר מדי לתוך פונקציה אחת. ברגע שזיהיתם מקרה כזה, זה הזמן לחלק את הפונקציה לשני חלקים או יותר.

אורך קובץ (מספר שורות מקסימלי בקובץ)

גם כאן אין מוסכמה מקובלת, למרות שרוב המתכנתים 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 לאבד בו את הידים והרגלים ולכנן כדאי להגביל את עצםם.

אורך מינימלי ומקסימלי לשמות משתנים

שם הכלל: id-length

שמות משתנים קצרים מדי או ארוכים מדי הופכים את הקוד ללא קריא. שמות קצרים מדי מקשים להבין מה מטרת המשתנה ושמות ארוכים מדי מסרבלים את הקריאה. כשבוחרים שם למשתנה כדאי להתחשב בקיים המנהים שדוני בהם תחת הכותרת "[בחירה שמות](#)".

בלי eval

שם הכלל: no-eval

כפי שכבר הסביר מוקדם יותר בספר, פונקציית eval היא פונקציה מסוכנת מבחינה אבטחת מידע. תמיד אפשר וכדי להימנע שימוש ב- eval באמצעות חלופות שונות לפתרון הבעיה.

בלי משתנים שלא הוצהרו

שם הכלל: no-undef

בגלאוסקריפט אפשר להציב ערך לתוך משתנה שלא הוצהר קודם. במקרה כזה המשתנה ייווצר על האובייקט глובלי (window, ב מקרה של ריצה בדף).

```
function greet(name, title) {  
  let firstName = name.split(" ")[0]; //  
  // משתנה שזמין רק בתוך הפונקציה  
  prefix = title || "Mr."; //  
  // הירות! המשתנה מוגדר על האובייקט глובלי  
  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 לבאגים לא צפויים כתוצאה מחוسر תשומת לב ל-*scope* שפועלים בו. מכיוון שהשם לשנתה של הוצהר מתבצעת כמעט תמיד בטעות, הכלל עוזר לאתר מקרים כאלה ולהימנע מהם.

לסיכום, כדי ליצור תוכנות איכותיות שהקוד שלהם ברור ונונה לתחזקה חשוב להקפיד על Best Practices, כמו אלו שסקרתי ברשימה החלקית כאן. הפעילו שיקול דעת בבחירה ובהתאמת הכללים שלפיהם תעבדו, כדי ליצור לעצמכם תהליך עבודה יעיל וקוד ממש טוב לאורך זמן.

נספה: בדיקות, יציבות וaicות קוד

מחבר: דניאל שטרנלייכט, חברת Outbrain

ברכחות! אם הגעתם לנספה זהה, כנראה אתם כבר מבינים איך ג'אווהסקריפט עובד ואתם מוכנים להתחיל לבנות אפליקציות ווב. אבל רגע אחד לפני שתתחלים, רציתי לספר לכם קצת איך לוודא שהקוד שאתם כתבים הוא יציב, איקוטי, ויכול לעמוד ב מבחן הזמן כשמפתחים שאיתם תעבדו יעשו בו שינויים בעתיד.

קצת רקע

באוטבריין, חברת המלצות התוכן הגדולה בעולם שמנגינה המלצות למיליארד משתמשים מדי חודש, עובדים יותר מ-200 מפתחים שכותבים אלפי שורות קוד ומשחררים עשרות גרסאות ביום. האתגר הוא לא קטן, שכן כל שינוי בקוד שמנפה עוזה צריך לבדוק 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 בתהליך העלאת הגרסאות?

התשובה: בדיקות. המון המונם בדיקות. למעשה, לפני כל העלאת גרסה באוטבריין רצות, באופן אוטומטי, לא פחות מאלפי בדיקות!

מבנה בדיקות

כשcottבים בדיקות, אפשר בקלות להגיע למצב שבו יש כל כך הרבה שבות שחייב בשביל זה קיימים היום כל הספריות והפרימיטוקים שעוזרים לסדר את הבדיקות. רוב הספריות מסודרות במתודולוגיה בשם 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 מסווג מסוים. תחת הfonקציה הזאת אפשר לkrוא לעוד **Assertions** פונקציה מאותו סוג כדי לתאר תת-סדרה. הfonקציה **test** היא הבדיקה עצמה, שתכיל "טענות" (או "טענות" באנגלית), וfonקציית **expect** היא הטענה עצמה. אם הטענה נופלת, הבדיקה נכשלה.

כשמדוברים על בדיקות, יש לא מעט סוגי: Unit Test, Integration Test, End-to-End Test, Sanity Test, Regression Test, System Test וכו', וכל בבדיקה המטרות שלה.

בנספה זהה אתמקד בשלושה סוגי בדיקות שונות:

- בדיקות יחידה (Unit Tests)
- בדיקות קצה לקצה (End-to-End) או בקיצור E2E
- בדיקות משתמש (UI Tests)

בדיקות יחידה

המטרה של בדיקות יחידה (או **Unit Tests** באנגלית) היא לבדוק יחידות קטנות של קוד שעומדות בפני עצמן. למעשה, ככל שהיחידות קטנות יותר, כך טוב יותר. בבדיקות יחידה יעבדו מול פונקציות או מול מחלקות, ולרוב יבדקו שקלט מסוים יחזיר פלט מסוים.

קחו לדוגמה את הfonקציה הבאה, שידועת לקבל מספר ולהחזיר את המספר מוכפל עצמו:

```
function multiply(number) {  
    return number * number;  
}
```

כדי לוודא שהfonקציה תקינה ועובדת בדיקת את מה שהיא נועדה לעשות, מרכיבים כמה בדיקות שונות:
1. בטור התחלה, מעבירים לפונקציה מספר ובודקים שהוא מוחזירה את התוצאה הנכונה:

```
expect(multiply(5)).toBe(25);
```

2. בהמשך, מודאים שהפלט המתקיים הוא מהסוג שמצוים לו:

```
expect(typeof multiply(1)).toBe('number');
```

3. אבל מה יקרה אם מעבירים לפונקציה קלט שלא מצפים לו, כמו מחזורת טקסט למשל?

```
multiply('text');
```

כשכתבם את הפונקציה לא חשבתם על האפשרות הזו, וכרגע בעקבות הרצאה הזאת היא תחזיר `NaN`, מה שעלול לגרום לשגיאות בהמשך. התרחיש הזה, אגב, עלול להתקיים בעיקר בשפות דינמיות כמו ג'אווהסקריפט, שבהן משתנים מסווגלים 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 הParmeter שעובר הוא לא מספר, מצפים לשגיאה:

```
expect(multiply('test')).toThrowError('Parameter is not a number');
```

בדרך כלל רוצים לעשות בדיקות ייחודית על יחידות עצמאיות באפליקציה כמו utilities, helpers, services וcdcומה, והבדיקות שMRIIZIM יבדקו שימוש פשוט לצד מקרי קצה – זאת על מנת לכסות את כל האפשרויות ולהגן עליהם מפני טעויות והתנהגוויות שלא צייפיתם להן בשימוש בפונקציה. בדיקות ייחודיה הן דינמיות, וצריך לזכור לשנות אותן אחרי שינויים משמעותיים בקוד.

בדיקות קצה לקצה (End-to-End)

או יש בדיקות ייחודיה שיעודו לróżן על חתיכות קטנות באפליקציה, אבל זה שככל אחד מהחלקים עובד בנפרד לא אומר שהם מסונכרנים ועובדים היטב יחד. נוסף על כך, בדיקות ייחודיה לא בודקות תרחישים שבהם מעורבות מערכות אחרות – לצורך העניין בדיקות ייחודיה שרצות על קוד ג'אווהסקריפט מצד הלוקה לא יודעת להגיד אם ממשκ המשמש עובד כמו שצריך עם הצד השרת.

קחו לדוגמה מפעל לייצור רכב מסווג פורד. יש מחלוקת אחת שאחראית להרכבת המנוע, אחת שאחראית לשלהה ואחת למחשב הרכב. המנוע עובד מצוין, השלהה נראהיה מעולה, המחשב עובד מצוין. ההרכבה של המנוע לשלהה עוברת בשלום, אבל כשباءים לחבר את המחשב, מגלים שהוא מיועד לרכיבים של פורמוללה 1.

בדיקות בשביל זה קיימות בבדיקות E2E. בעזרת בדיקות E2E אפשר לבדוק שהאפליקציה עובדת מקרה אחד לקרה שני. בבדיקות E2E יכולים תרחישים שימושיים ממערכות אחרות, ויודאו שהבדיקה מצילהה להגעה מ对照检查 ב' ללא הפרעות.

לשם ההדגמה אקח אפליקציית חיפוש שכולם משתמשים בה מדי יום: גוגל.

הינה התרחיש שרוצים שהבדיקה תבדוק:

1. לך לאתר google.com.
2. לחץ על אלמנט תיבת החיפוש עם ID בשם "search".
3. הקלד את המילים "אוטוביין" בתיבת החיפוש.
4. לחץ על מקש Enter.
5. בדוק שה-URL google.com/search הנוכחי הוא `q=אוטוביין`.
6. בדוק שקיים query param בשם `q` עם הערך "אוטוביין".
7. בדוק שתיבת החיפוש בעלת ID מסווג "search" מכיל את הערך "אוטוביין".

משמעותו לב: הבדיקה נעשית כתוצאה מפעולות של משתמש אמיתי באתר אמיתי. הרעיון הוא לבדוק אם האפליקציה עובדת ללא התחשבות במבנה שלה או בטכנולוגיה שועומדת מאחוריה, סוג של "קופסה שחורה".

בדיקות ממשקי משתמש (UI Tests)

בדיקות קצרה לנוולות וירודעות לתת מענה לתרחישים שכולים מערכות אחרות, אבל לרוב הן בודקות "Happy flows" – מקרים שבהם הכל עובד כמו שצריך. אבל מה לגבי מקרים קצרים או מקרים שבהם המערכות שהבדיקות תלויות בהן לא עובדות או לא מוחזרות את מה שימוש המשמש מצפה לקבל?

הבה ניקח כדוגמה את התרחיש שבודק חיפוש בגוגל. התרחיש יוצא מנקודת הנחה שהשרטים שאחראים על החזרת תוצאות החיפוש עובדים ומחזירים תשובה מסוימת. אבל מה יקרה אם תהיה בעיה בשרתים והם לא יחוירו תשובה? או לחלופין הם יחוירו תוצאות ריקות? האם האפליקציה שבניתם תדע להתמודד עם התרחישים הנ"ל ותיראה כמו שאתה מצפים שהיא תיראה?

אפשר לכתוב סדרה של בדיקות קצרה שעוברות על תרחישים מהסוג זהה, אבל כאמור בדיקות קצרה הן יחסית כבזות, והמטרה שלהם היא להריץ בדיקות שיוצאות מנקודת הנחה שהכל עובד.

מפתחי אוטוביין מספרים: נתקלנו בבעיה הזאת ורצינו שתהיה לנו דרך להריץ בדיקות על ממשך המשתמש ועל איך הוא מגיב למקרים קצרים ולתרחישים מורכבים, אז הוספנו עוד שכבה של בדיקות שבאה לבסota מקרים מהסוג הזה. אנחנו קוראים לבדיקות האלה "בדיקות ממשקי משתמש" (UI Tests באנגלית). כדי להריץ את בדיקות ה-UI צריכים סביבה סגורה שלא תלווה בשרתים (או לפחות לא בשרתים אמיתיים), אז בנוינו כלי בשם "לאונרדו" שיזען זהות בקשות לשרת, להשתלט עליהם ולדמota תשובה שאנחנו מגדירים מראש.

כך למשל אפשר לראות איך האפליקציה מתמודדת עם תשובות ריקות, עם שגיאות ש망יעות מהשרת ועם מצבים שבהם השרת לא מגיב. כמו כן אפשר לבדוק מצבים שבהם התגובה מהשרת איטה.

בדיקות מהסוג הזה הן מהירות, יציבות ויעילות מאוד שיש להן את היכולת לבדוק כל תרחיש ש্רוצים ללא תלות במערכות אחרות.

הכלי לאונרדו הוא כלי open source זמין לכלם בGITHub תחת הכתובת:

<https://github.com/outbrain/Leonardo>

ספריות ופרימורקים מומלצים

יש לא מעט ספריות ופרימורקים לכתיבת בדיקות בג'אוהסקריפט, הינה כמה שאנו, באוטבריין, משתמשים בהם וממליצים לכם בחום לננות:

- Jest – ספְרִיָּת הבדיקה של פיסבוק. נותנת מענה לרוב סוגי הבדיקות ובעל API אינטואיטיבי לכתיבה הבדיקות ולהתמודדות עם שגיאות. עובדת מעולה עם ספריות מודרניות כמו React, Angular ו-Vue.
- Jasmin – ספְרִיָּת בדיקות ותיקה ומאוד פופולרית. טובה מאד בבדיקות ייחידה ועומדת בפני עצמה ללא תלות במערכת כלשהי.
- Karma – כלי פשוט שיאפשר לכם להרין קוד ג'אוהסקריפט ובמקביל לבצע בדיקות בדפדפני.

סיכום

למזהם מה אומר המושג "בדיקות" בכתיבת קוד, ואיך הוא יעזר לכם לשמור על יציבות ועל קוד נקי. ראיתם דוגמאות לבדיקות ייחידה ובדיקות קצה ובהן לכמה מהן כל כך חשובות. וגם המלצנו לכם על ספריות ופרימורקים שייעזרו לכם לכתוב בדיקות.

עוד משהו קטן ששווה לציין: בקבוצות פיתוח מסוימות לקחו את עניין הבדיקות רחוק יותר ואמצו מתודולוגיה בשם "פיתוח מונחה-בדיקות" או באנגלית **Test-Driven Development** (בקיצור TDD). לפי המתודולוגיה זו, בדיקת ייחידה תיכתב עוד לפני כתיבת הקוד שאותו היא בודקת. קצת על הקצה, אבל אם תחליטו לכתת עם המתודולוגיה זו, אתם יכולים להיות בטוחים שהקוד שתכתבו יעבד בצורה חלקה.

נספח: Corvid by Wix

מחברת: מור גלעד, חברת Wix

הקדמה

Corvid היא פלטפורמת פיתוח אפליקציות שנבנתה על העורך הווייזואלי של Wix. העורך הווייזואלי של Wix נותן למפתח את היכולת לייצר ממשק ויזואלי מפותח מאוד מבלי להשתמש ב-HTML או ב-CSS. הוא מכיל מאות רכיבים חזותיים (elements) שאפשר לעצב אותם באינסוח אפשרויות. בעזרתו אפשר ליצור אתרים מרהיבים.

הפלטפורמה של Corvid מאפשרת:

- לכתוב קוד בג'אויסקריפט עבור הפונקציונליות של האתר
- לשנות פעולות והתנהגות של רכיבים באתר
- לייצר מבני נתונים

מה בונים?

בפרק הנוכחי אלמד אתכם בצעדים קטנים איך לבנות אפליקציה ניהול משימות בעזרת Corvid. אפשר לראות את האפליקציה שנבנה בכתובה [.corvidtodomvc.com](http://corvidtodomvc.com)

- הקוד בספר נכתב במטרה ללמד ולהמחייב בצורה ברורה ו פשוטה ולא דוקא לגרום למערכת לעבוד בצורה אידיאלית ויעילה.



The screenshot shows a user interface for a todo application. At the top, there is a search bar with the placeholder "I want to do...". To the right of the search bar is a button with the number "1". Further to the right is a large blue circular button with a white arrow pointing right. Below the search bar, there is a summary: "1 item left 3". To the right of this summary are three buttons: "All Tasks" (40), "Completed" (0), "Active" (5), and "Clear completed". A horizontal line separates this from the main list area. The list area contains two items: "Go to the gym" (checked) and "Finish writing the book" (unchecked). Each item has a small number "2" next to it. The background of the application is light blue.

תיאור כללי של אפליקציית המשימות:

1. תיבת טקסט (text input) וכפתור ההוספה המאפשרים למשתמש להוסיף משימה חדשה.
 2. רישימת המשימות – עברו כל משימה רואים את הסטוס שלה, אם היא הושלמה או לא.
 3. מספר הרשימות שנותרו 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 יופיע למשתמש חלונית שתשאל אותך אם הוא מעוניין לבצע את הפעולה. אם המשתמש יסכים, כל המשימות אשר הושלמו – יימחקו.
- במהלך הפרק תשתמשו בהרבה פונקציות sh-Corvid מספקת עבור מתכנים וסביר את כולן. אם ברצונכם להרחיב את הידע שלכם וללמוד פונקציות נוספות מ투אות בפרק זה, אתם מוזמנים לבקר באתר המידע של Corvid reference

<https://www.wix.com/corvid/reference/APIsOverview.html>

עבור כל פונקציה שאלמד פה, אוסף הפניה לאזרור שבו היא מוסברת באתר המידע.

איך מתחילה?

הכני עבורהם תבנית מעוצבת מרראש שממנה תוכלו להתחיל =<

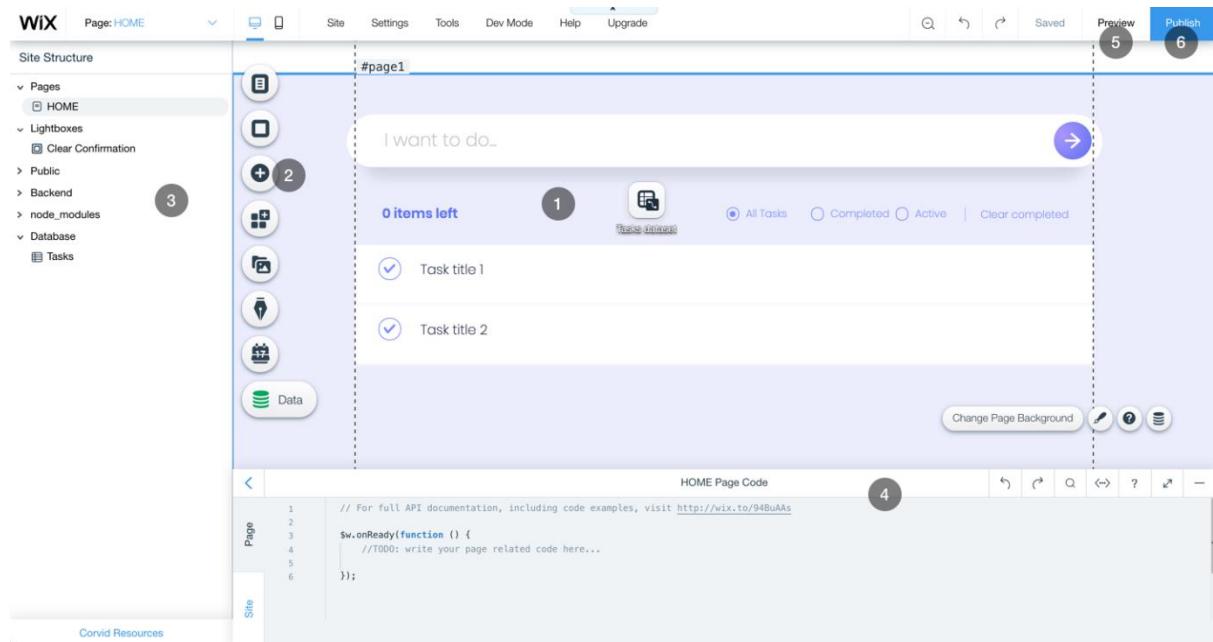
<https://www.wix.com/website-template/view/html/2186>

העורך הויזואלי של Wix עשיר ביכולותיו ויש הרבה מה ללמוד עליו. אבל מכיוון שאתם לומדים כרגע ג'אווהסקריפט, רציתי להסוך לכם את זמן העיצוב והבנייה של הרכיבים באפליקציה על ידי שימוש בתבנית ולהשאיר לכם רק את החלק המהנה – כתיבת הקוד. מובן שאתה יכולם לשנות את העיצוב ולהוסיף רכיבים חדשים.

לאחר שנכנסתם לתבנית, לחזו על [Edit this site](#), הירשמו ל-Wix באמצעות כתובת האימייל שלכם, וcutם אותם נמצאים בעורך הויזואלי של Wix – ברוכים הבאים ☺ כדי להפעיל את Corvid, עברו בעזרת הנקודות מעל Dev Mode, בתפריט שנמצא בחלק העליון של האתר ואו על

[Enable Corvid](#)

אחרי שהפעילם את Corvid, תראו את המסק הבא:



או מה בעצם רואים פה?

1. **עורך ויזואל.** האזר שבו עורכים את הצד הוויזואלי של האתר. אל אוצר זה מוסיפים רכיבים ומשנים את העיצוב שלהם. אפרט לאבי הרכיבים בהמשך.

2. **הכתר +**, להוספה רכיבים חדשים לאתר. תפריט הרכיבים מכיל מאגר של מאות רכיבים שאפשר להוסיף לאתר על ידי גירירה.

3. **מבנה האתר | site structure.** פאנל שמציג את כל העמודים, חלונות (lightbox), קובצי הקוד ומסדי הנתונים שיש באתר.

4. בדוגמה הנוכחי יש כרגע עמוד אחד שנקרא **HOME**, חלונית שנקראת "Clear Confirmation" ומסד נתונים המכיל טבלה שנקראת **Tasks**.

5. **סביבה הפיתוח.** המקום שבו כתבים את הקוד. אפשר לכתוב קוד עbor כל עמוד באפליקציה. עבור כל עמוד חדש באפליקציה, מתחילה עם תבנית קוד מוכנה המכילה **\$w.onReady()** – שדבר עליה בהמשך.

* יכול להיות שסביבת הפיתוח שלכם תהיה מצומצמת בשורה התחתונה. לחזו על כדי להרחב אותה.

6. **כפתור הצג | preview.** מציג את האתר שעבדתם עליו עד עכשיו. לרוב, משתמשים בכפתור זה תוך כדי עבודה על האפליקציה לפני שרווצים לפרסם אותה לעולם.

במהלך הפרק, אעשה עצירות מדי פעם כדי לראות מה עשיתם עד עכשיו. בעצרות האלו אבקש שתתחלצו על **preview**.

7. **כפתור פרסום | publish.** מפרסם את האתר שנבנו עד עכשיו וחוושפ אותו לכל העולם.

התבנית שעלייה אתם בונים את האפליקציה מגיעה מוכנה עם מסד נתונים, שבו טבלה Tasks שמכילה משימות. תלמדו איך להוסיף לטבלה זו משימות חדשות, לשנות את סטוס המשימה, למחוק משימות ועוד כל מיני פעולות שיעזרות לניהל מידע.

אתם יכולים לראות את המשימות שנמצאות בטבלה על ידי להיצה על Tasks ב-site structure.

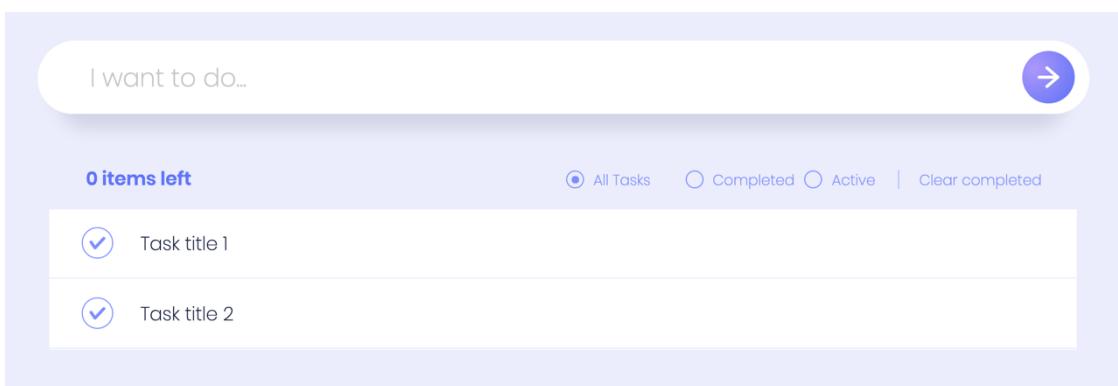
כל שורה מייצגת משימה וכל משימה יש שני מאפיינים:

- Title – תיאור המשימה

- Completed – משתנה בוליעני המתאר אם המשימה הושלמה או לא

אם אתם רוצים ללמוד איך ליצור את הטבלה בעצמכם, קפצו לסוף הפרק, לנספה שבו הוספה לכם פירוט והרבה על בניית טבלאות.

לפני שמתחילה לכתוב קוד, לוחצים על preview ומתבוננים בנקודת הפתיחה:



כפי שאפשר לראות, כל הרכיבים מוכנים ומעוצבים מראש אבל אף כפטור לא מgive להיצחה ומספר המשימות שנותרו אינו נכון.

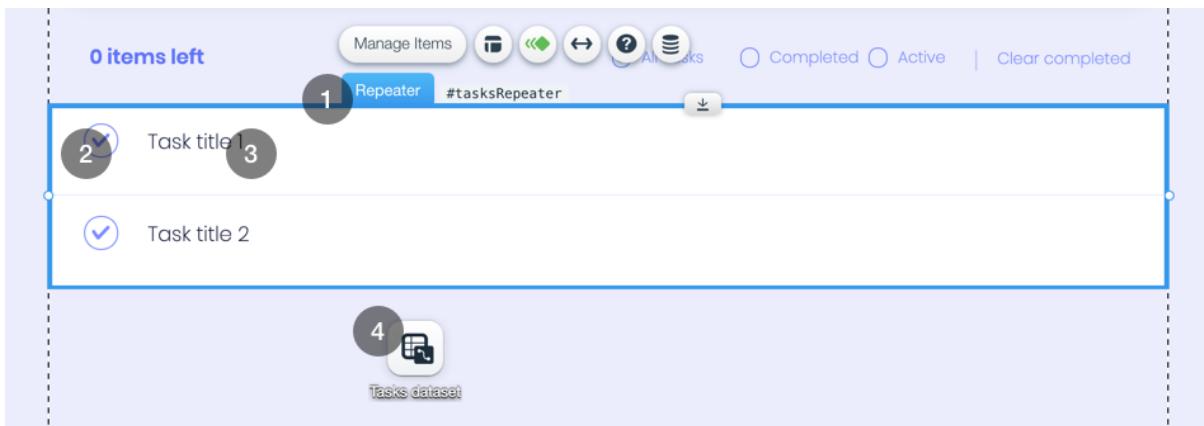
עכשו לוחצים על **כדי המשיך לעבוד על האפליקציה.**

[Back to Editor](#)

הציגות נתונים ממסד הנתונים באתר

בחלק זה אלמד איך אפשר להוסיף נתונים מהטבלה על ידי "חיבור" רכיבים מהאפליקציה לטלחת-h-Tasks שלנו. כך חוסכים כתיבת קוד טריוואלית. מובן שניתן לעשות את זה גם באמצעות קוד – אבל אל דאגה, כתבו הרבה שורות קוד ממש בקרוב ☺

מתחלים בהכרת הרכיבים שייקחו חלק בהציג המשימות שבטלחת-h-Tasks של האפליקציה.



.1 – רכיב שיצרת בתוכו כמה עותקים של רכיב אחר. כל עותק מכיל עיצוב זהה אך מידע משתנה. מספר העותקים שייצרת הוא משתנה ונקבע לפי מספר הפריטים שעליו להציג. במקרה הזה, מספר המשימות שהוא יציג.

[https://www.wix.com/corvid/reference/\\$w.Repeater.html](https://www.wix.com/corvid/reference/$w.Repeater.html)

.2 | תיבת סימון – רכיב המאפשר הכנסת ערך בוילאי (true | false) Checkbox

[https://www.wix.com/corvid/reference/\\$w.Checkbox.html](https://www.wix.com/corvid/reference/$w.Checkbox.html)

.3 | רכיב טקסטואלי – רכיב שמציג טקסט. Text

[https://www.wix.com/corvid/reference/\\$w.Text.html](https://www.wix.com/corvid/reference/$w.Text.html)

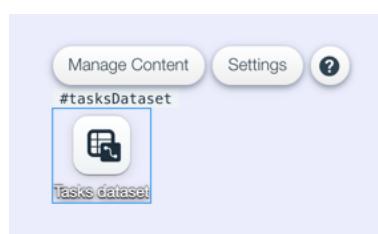
.4 – הרכיב שמקשר בין המידע בטבלת Tasks לבין מרכיבים אחרים בעמוד. Dataset

<https://www.wix.com/corvid/reference/wix-dataset.html>

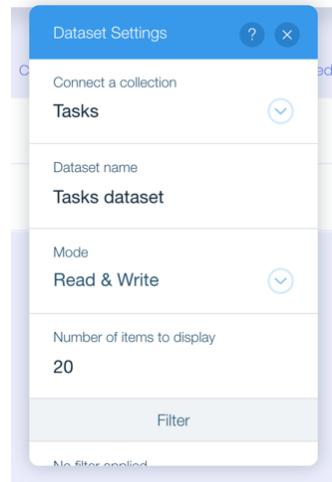
Dataset

ה-component Dataset מחבר רכיבים בעמוד למידע מטבלאות נתוניות. 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 באמצעות כפתור `settings`:



או נראה שהואdataset מחובר לטלטלה Tasks וنمצא במצב של קרייה וכתיבה (read & write).



הוא צריך להיות מוגדר במצב של קרייה וכתיבה, מכיוון שתרצה אפשרות להציג את המשימות אבל גם לשנות את סטטוס ה-*completed* שלו.

* יש עוד שני מצבים ל-*dataset*:

– מצב שמאפשר רק קרייה של נתונים מהטלטלה Read Only
– מצב שמאפשר רק הוספה של נתונים חדשים לטלטלה Write Only

נוסף על כך, Number of items to display מהוווה את כמות הרשומות המקוריות שהdataset ישולוף מהטלטלה.
אם אתם רוצים לתמוך ביותר מ-20 משימות, אתם יכולים לשנות את זה בהגדורה.

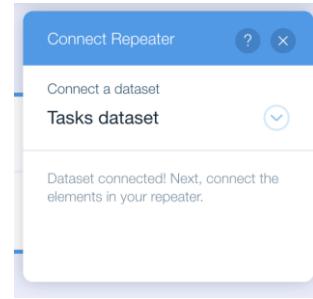
חיבור רכיבים למידע מהטלטלה

כעת, משתמשים ב-*dataset* על מנת להציג את המשימות בכל שורה ב-*repeater*.

על מנת לעשות זאת זה לוחצים על האיקון של מסד הנתונים, שMOVED כאשר לוחצים עם העכבר על ה-*repeater*.

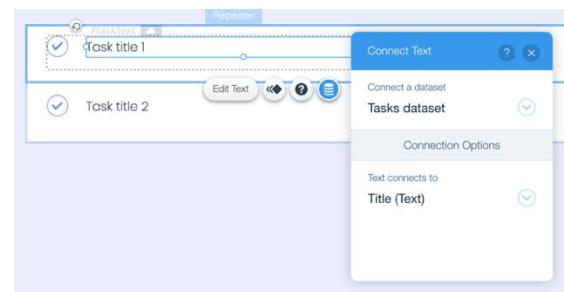


כעת נפתח פאנל החיבור (connection panel) של ה-*repeater*. מכיוון שאתם רוצים לחבר את ה-*dataset* ל-*repeater*, בחרו אותו ב-*connect a dataset*.



התוצאה של הפעולה זו היא שמספר הקופסאות ב-repeater ישוכפלו כמספר הרשומות שה-dataset שולף מטבלת המשימות. הצעד הבא הוא להציג כל משימה בשני הרכיבים שיש בתחום ה-repeater.

מתחילה ברכיב הטקסט. פותחים את פאנל החיבור של רכיב הטקסט (זה דורש לחייב ראשונית על ה-repeater) ומחברים את הטקסט של הרכיב לשדה Title מטבלת ה-Tasks.



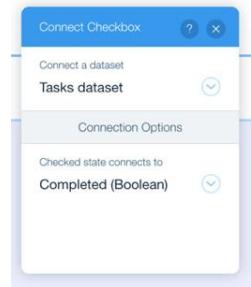
- אולי שמתם לב שה-dataset שיצרתם כבר מחובר. זה קורה באופן אוטומטי מכיוון שהחברתם את ה-dataset ל-repeater.
- ברשימה השדות שנפתחת יש שדות נוספים שאפשר להתחבר אליהם. אלו שדות שנוצרו באופן אוטומטי עבור כל רשומה בטבלה:

ID -

createdDate -

updatedDate -

לאחר מכן, פותחים את פאנל החיבור של ה-checkbox (גם זה נדרש לחייב ראשונית על ה-repeater), לאחר מכן לחייב על ה-group שמכיל את ה-checkbox, ולבסוף על ה-checkbox (checkbox checked) ומחרבים את הערך לשדה completed מהטבלה.

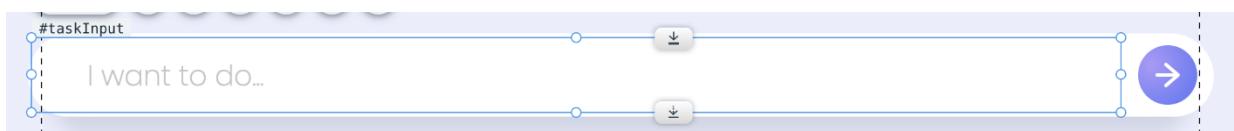


אחרי שהחברתם את הרכיבים שלכם לטבלה, הגיע הזמן לראות את התוצאות!
לחצו על preview והסתכלו על התוצאות. התוצאות הרצויות הן:

רואים את המשימות? יש!
עכשו הגיע הזמן להתחיל לכתוב קוד – חגיגת!

הוספה חדשה למשימה

הרכיבים שיעזרו לכם להוסיף משימה חדשה:



1. תיבת טקסט | text input – מאפשרת להכניס את תיאור המשימה.
[https://www.wix.com/corvid/reference/\\$w.TextInput.html](https://www.wix.com/corvid/reference/$w.TextInput.html)
2. כפתור | Button – בלחיצה עליו תתווסף המשימה מתיבת הטקסט לטבלה Tasks
[https://www.wix.com/corvid/reference/\\$w.Button.html](https://www.wix.com/corvid/reference/$w.Button.html)

אם רוצים לאפשר למשתמשים להוסיף משימה חדשה, יש ללמוד איך בלחיצה על הכפתור  מוסיפים משימה חדשה לתוך מסד הנתונים ומציגים אותה ב-repeater.

Events

[https://www.wix.com/corvid/reference/\\$w.Event.html](https://www.wix.com/corvid/reference/$w.Event.html)

בדומה לארועים בג'אווה סקריפט, גם הרכיבים של Corvid מאפשרים של אירועים (events). אלמד אתכם שני אירועים על שני רכיבים שונים:

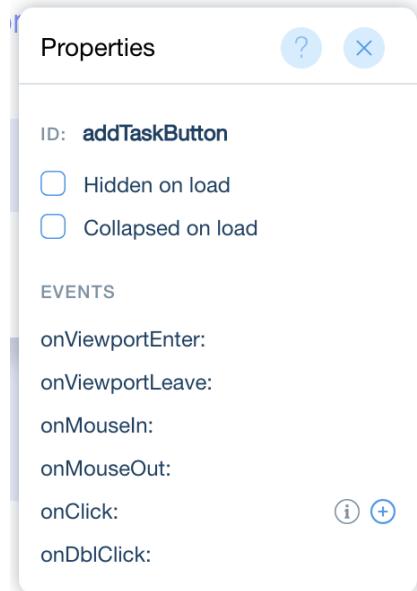
- button.onClick – event שנראה כאשר המשתמש לוחץ על כפתור.

[https://www.wix.com/corvid/reference/\\$w.Button.html#onClick](https://www.wix.com/corvid/reference/$w.Button.html#onClick)

- textInput.onKeyPress – event שנראה כאשר המשתמש לוחץ על כפתור במקלדת בזמן שהוא נמצא בתוך רכיב ה-text input.

[https://www.wix.com/corvid/reference/\\$w.TextInput.html#onKeyPress](https://www.wix.com/corvid/reference/$w.TextInput.html#onKeyPress)

מתחלים בהוספה event לכפתור  . עושים זאת על ידי פתיחת פאנל ה-properties של הכפתור, בклיק ימני על הכפתור ולהיצה על View Properties.



פאנל ה-properties מחולק לשני חלקים:
בחלקו העליון אפשר לראות את ה-ID של הרכיב. בעזרת ה-ID אפשר לבצע פעולה עם הרכיב דרך הקוד. בתבנית שהכנתי לכם, נתתי מראש לכל רכיב ID שם משמעותי, אבל אתם יכולים לשנות את זה. בנוסף לכך, מתחת ל-ID מופיעות הגדרות הנראות של הרכיב בזמן העליה של העמוד.

בחלקו התיכון של הפאנל נמצאת רשימה של כל ה-events הנתמכים עבור הרכיב:

onViewportEnter	-
onViewportLeave	-
onMouseIn	-
onMouseOut	-
onClick	-
onDblClick	-

* לכל רכיב יש events שונים בהתאם לפונקציונליות שהוא מספק. ניתן לראות הסבר מדויק על כל ה-events של הכפתור בlienck:

[https://www.wix.com/corvid/reference/\\$w.Button.html](https://www.wix.com/corvid/reference/$w.Button.html)

במקרה הזה רוצים להוסיף משימה כאשר המשתמש לוחץ על הכפתור. לכן, עוברים עם העכבר על הצד הימני של event ה-**onClick** event ולוחצים על כפתור ה-. לחיצה על הכפתור תכניס באופן אוטומטי שם event 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 event שלכם. הנה נראה שזה אכן נקרה בכל לחיצה. מוסיפים כתיבה ל-console:

```
export function addTaskButton_click(event) {
  console.log('button clicked');
}
```

שיםו לב שהפונקציה שלכם מקבלת ארגומנט שנקרא event. הארגומנט זה הוא אובייקט המכיל מידע על ה-event שנקרה. האובייקט זה משתנה בהתאם לסוג ה-event.

* מעתה כתבו ותעתיקו הרבה שורות קוד. אם אתם רוצים שהקוד יעבור הזזה באופן אוטומטי, לוחזו על

הכפתור , שנמצא מצד הימני של החלק העליון בסביבת הפיתוח.

עוברים ל-w-**preview**, לוחצים על הכפתור ובתחתית הדף ייפתח ה-**Corvid** של console ויזיג את מה שכתבתם בכל פעם שתלחצו. עובד לכם? מעולה! ממשיכים להלאה ושמוררים את המשימה.

\$w

[https://www.wix.com/corvid/reference//\\$w.html](https://www.wix.com/corvid/reference//$w.html)

על מנת לשמר את המשימה שהמשתמש כתב, יש "להוציא" את הטקסט שנכתב מתוך רכיב ה-text input. את זה עושים בעזרת שימוש בפונקציה \$w.

\$w היא פונקציה שזמין בקורס 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?
זכרים את ה-ID שיש לכל רכיב בפאנל ה-properties? אז על מנת לקבל את המשתנה של הרכיב משמשים ב-ID שלו. ואו, כדי לבחור אותו, קוראים לו \$w עם ארגומנט מסווג טקסט, המכיל את ה-ID של הרכיב ומתחיל ב-#.

לדוגמא, אם תפתחו את פאנל ה-properties של תיבת הטקסט שלכם, תראו שה-ID שלו הוא taskInput וואז תבחרו אותה כך:

```
$w('#taskInput').value
```

אחרי שיש תיבת טקסט, רוצים לדעת מה הטקסט שהמשתמש הבניס לתוכה. לכל רכיב יש מספר רב של נתונים שהוא יכול לספק. במקרה זה 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 המשתנה ה-value של תיבת הטקסט:

[https://www.wix.com/corvid/reference/\\$w.TextInput.html#value](https://www.wix.com/corvid/reference/$w.TextInput.html#value)

עובדרים לו-\$w, כתובים את המשימה בשורת המשימה,لوحצים על הכפתור והייד! רואים את הטקסט שכתבתם.



הצעד הבא => לשמר את המשימה כמשימה חדשה בטבלת Tasks.

wix-data

<https://www.wix.com/corvid/reference/wix-data.html>

wix-data הוא מודול המאפשר עבודה מול המידע שיש לכם בטבלאות. בעורתו, אפשר להוסיף רשומות לטבלה, למחוק רשומות וגם לעורק רשומות קיימות.

במהלך הפרק אלמד בכל פעם שימוש ופונקציונליות חדשה של wix-data.

על מנת להשתמש במודול, יש ליבא אותו מהאזור העליון של קוד הקוד:

```
import wixData from 'wix-data';
```

הוספה רשותה לטבלה – wixData.insert()

<https://www.wix.com/corvid/reference/wix-data.html#insert>

מתחלים עם היכולת להכניס משימה חדשה לטבלה. זה אפשרי באמצעות הפונקציה האシンכרונית insert() לפונקציה זו מעבירים שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשותה החדשה – במקרה זה Tasks.
2. המשימה שאויה רוצים להוסיף. המשימה תהיה במבנה של אובייקט, שבו המשתנים יהיו זהים למאפיינים של המשימה בטבלה והערכים שלהם.

או על מנת להוסיף משימה חדשה לטבלה, מייצרים את אובייקט המשימה שיכיל:

- title – טקסט המתאר את המשימה, שנלקח מהתיבת הטקסט

- completed – משתנה בוליאני המייצג את סטוס המשימה ויואתחל ב-false

קוראים ל-wixData.insert() עם האובייקט שיצרתם. נוסף על כן, משלימים את החוויה בכך שימושיפים 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);
  }
}
```

* שימושו לב שמיון שהפונקציה wixData.insert() היא אסינכרונית, מוסיפים event.async ל-

עכשו קופצים שוב ל-preview, מוסיפים משימה חדשה, לוחצים על Enter, חוזרים לeditor ועובדים לעורך התוכן של טבלת Tasks כדי לראות אם המשימה החדשה הותוספה.

התווסף? מעולה!

אבל אני מניה ששפתם לב שחררים כמה דברים כדי שהחוויה תהיה שלמה:

1. שורת המשימה לא הטרוקנה אחרי הוספת המשימה.
2. המשימה לא התווסף לרשימה המשימות שלכם.

ציינתי קודם שבעוזרת \$w אפשר לקבל גם לשנות מידע על רכיב באתר. אז הפעם, במקום לקבל את הערך של תיבת הטקסט שלכם, יש לאפס 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 האсинכרונית .refresh()

```
await $w('#dataset1').refresh();
```

למידע נוסף על פונקציית ה-refresh של dataset

<https://www.wix.com/corvid/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,עובדים ל-preview ומכניסים משימה חדשה. איזה כיף! זה עובד!

תרגילים:

בצעו שמיירה 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.

השובה:

לאחר פתיחה של פאנל ה-properties וואז יצירה של event.onKeyPress יתווסף event נוסף לסביבת הפיתוח. אבל הפעם יהיה צורך לוודא שהכפתור שעליו המשתמש לחץ הוא כפתור ה-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 ייראה 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-event taskInput_KeyPress ההפוך לאсинכריוני ונוסף לו .async

עוברים שוב ל-preview ומכניסים משימות חדשות באמצעות לחיצה על Enter וגם בעזרת לחיצה על כפתור ההוספה.

שינוי סטטוס המשימה

או אפשר לראות את המשימות הקיימות ולהוסיף משימות חדשות. אבל המשתמש עדיין 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 בהוספה event של onChange על ה-checkbox פאנל repeater, כמו שמלמדתם קודם:

```
export function completedCheckbox_change(event) {  
    //Add your code for this event here:  
}
```

למידע נוסף על onChange event של כפתור הבחירה:

[https://www.wix.com/corvid/reference/\\$w.Checkbox.html#onChange](https://www.wix.com/corvid/reference/$w.Checkbox.html#onChange)

* מכיוון שה-checkbox נמצא בתוך ה-repeater event שנמדד על checkbox אחד יופעל אוטומטית על כל אחד מהמ-checkboxות שיש בכל איבר ב-repeater. כלומר אין צורך לכתוב event לכל checkbox אם יש לנו repeater.

כדי לשנות את הערך הבוליאני של מאפיין ה-completed של המשימה לערך החדש שהמשתמש סימן ב-checkbox, קודם כל צריך להציג את הערך החדש, ועושים זאת על ידי שימוש במשתנה ה-target, שנמצא 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/corvid/reference/\\$w.Event.html#target](https://www.wix.com/corvid/reference/$w.Event.html#target)

לאחר מכן נרצה לעדכן את הערך החדש בטבלה.

wixData.get() – שיליפת רשומה מהטבלה

<https://www.wix.com/corvid/reference/wix-data.html#get>

על מנת לייצר את המשימה שאותה רוצים לערוך, יש לשולף את הרשומה הזו מהטבלה כדי שכל הנתונים הנוכחים שלה יהיו זמינים.

למודול wixData יש פונקציה אסינכרונית הנקראת ()`get`, שמאפשרת שליפה של משימה מהטבלה. לפונקציה זו מעבירים שני ארגומנטים:

1. שם הטבלה שמננה רוצים לשולף את המשימה – `Tasks` –
2. המזהה הייחודי של המשימה – `ID`.

לכל משימה שמכניסים לטבלה מתווסף באופן אוטומטי מזהה שנקרו `_id`. המזהה הזה מאפשר לזהות באופן ייחודי את המשימה.

או איך בעצם מישגים את המזהה הייחודי זהה מתחם המשימה ב-`repeater` שעליו לחץ המשמש? בעזרה ארוגמנט `event`. כדי שכתบทי קודם, ארוגמנט `event` משתנה בהתאם ל-`event` שנקרא. אבל יש חשיבות למיקום הרכיב שבו מתרחש ה-`event` ובמיוחד אם הוא בתוך ה-`repeater`. אם ה-`event` הופעל והרכיב שהוא הופעל נמצא בתחום ה-`repeater` אנו מקבל באובייקט ה-`event` משתנה נוסף, שאינו `context` נוסף, אלא ב-`event` רגיל, שנקרו `context` ומכיל את ה-`ID` של הרשומה שצרכים מהטבלה:

`event.context.itemId`

למידע נוסף על משתנה ה-`context` של `:event`:

[https://www.wix.com/corvid/reference/\\$w.Event.html#context](https://www.wix.com/corvid/reference/$w.Event.html#context)

ובעת, כשייש 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, הפתהם את ה-`event` שלכם לאסינכרוני כיוון ש-`wixData.get()` היא אסינכרונית.

עדכון רשומה בטבלה – wixData.update()

<https://www.wix.com/corvid/reference/wix-data.html#update>

יש לכם רשומה, כפי שהיא שמורה בטבלה. עכשו, יש לשנות את ערך ה-`completed` שלו לערך החדש ולעדכן אותו בטבלה.

למודול `wixData` יש פונקציה `update()` שמאפשרת עדכון של רשומה מסוימת. הפונקציה זו מקבלת שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה – `.Tasks`
2. המשימה המעודכנת שרוצים לשמר.

מייצרים משימה מעודכנת ואז קוראים לה-`wixData.update()` עם העדכון:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;

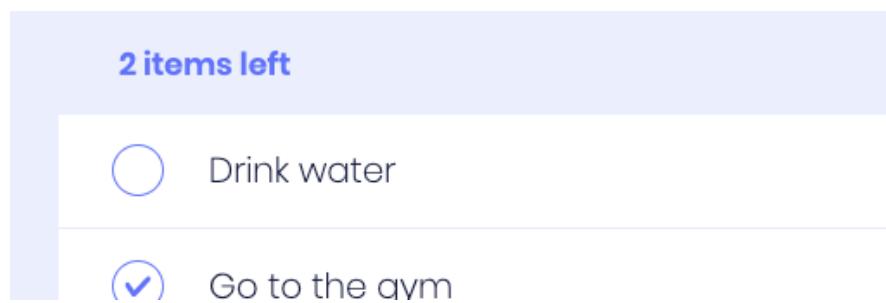
  const item = await wixData.get('Tasks', itemId);
  const updatedItem = Object.assign({}, item, { completed: completed });

  await wixData.update('Tasks', updatedItem);
}
```

הבה נבדוק אם זה עובד ב-`preview`. מסננים כמה משימות וחזררים לעורך התוכן לבדוק אם זה באמתעובד.

נדיר! זה אכן עובד! הפונקציונליות הבסיסית של האפליקציה עובדת.

מספר המשימות שלא הושלמו



היכולת הבאה שורצים להוסיף לאפליקציה היא תצוגה של מספר המשימות שלא הושלמו (שהערך הבוליאני שלhn במאפיין ה-completed הוא false) ברכיב הטקסט שמופיע מעל רשימת המשימות.

מתחילתים בبنיה פונקציה אסינכרונית חדשה שתפקידה יהיה לעדכן את רכיב הטקסט במידע הרצוי.
שםה: updateActiveTaskCount

```
async function updateActiveTaskCount() {  
} async function updateActiveTaskCount() {  
}
```

הצעד הבא הוא לספר 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/corvid/reference/wix-data.WixDataQuery.html>

wixDataQuery היא שאלתה המאפשרת לבצע בקשה למידע מטבלת המשימות. הבקשה יכולה להכיל שילוב של פקודות שיגדרו את התוצאות שלhn מצפים.

יצירת שאלתה נעשית בשלושה שלבים:

1. אתחול השאלתה בהינתן שם הטבלה שממנה רוצים לקבל את המידע.
2. בניית הפקודות שגדירות את השאלתה.
3. הפעלת השאלתה.

אתחול השאלתה – wixData.query .1

<https://www.wix.com/corvid/reference/wix-data.html#query>

את השאלתה מאתחלים בעזרת הפונקציה `wixData.query()`, שמקבלת ארגומנט אחד:
1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה.

מתחילתים לבנות את השאלתה בעזרת:

wixData.query('Tasks')

2. בניית השאלה

לאחר מכון בונים את הפקודות של השאלה באמצעות מגוון פונקציות. במקרה זה יש לקבל רק את המשימות שלא הושלמו. כלומר, כל המשימות שערך ה-completed שלן הוא .false

- לכן, משתמשים בפונקציה eq (מהמילה שווה – equal), שמקבלת שני ארגומנטים:
1. שם המאפיין שבאמצעותו רוצים לסנן את התוצאות – במקרה זה .completed
 2. הערך שרוצים שהוא לתוצאות – במקרה זה .false

למידע נוסף על פונקציית ה-eq:

<https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#eq>

wixData.query('Tasks')

.eq('completed', false)

* יש פונקציות רבות ומגוונות שמאפשרות בניית שאלה ואפשר לקרוא על ולן בlienk:

<https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html>

3. הפעלת השאלה

לביצוע השאלה אפשר להשתמש בשתי פונקציות אסינכרוניות שונות:

1. find() – הזרת כל האיברים שעוניים על השאלה.

<https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#find>

2. count() – הזרת מספר האיברים שעוניים על השאלה.

<https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#count>

מכיוון שצורך לדעת רק את מספר המשימות שלא הושלמו, משתמשים בפונקציה ()count:

```
const activeTaskCount = await wixData.query('Tasks').eq('completed', false).count();
```

עכשו, כשהוושג מספר המשימות שלא הושלמו, כותבים אותו ברכיב הטקסט.

מתחלים בייצור הטקסט שרצוים לראות:

```
const activeTaskText = activeTaskCount === 1 ?  
  '1 item left' :  
  `${activeTaskCount} items left`;
```

עכשו, מציבים את הטקסט החדש בערך ה-`text` של רכיב הטקסט שה-ID שלו הוא `#activeTaskText`:

למידע נוסף על משתנה ה-`text` השיך לרכיב הטקסט:

[https://www.wix.com/corvid/reference/\\$w.Text.html#text](https://www.wix.com/corvid/reference/$w.Text.html#text)

זכרים איך עושים את זה?

```
$w('#activeTasksCount').text = activeTaskText;
```

הינה כל הפונקציה:

```
async function updateActiveTaskCount() {  
  const activeTaskCount = await wixData.query('Tasks')  
    .eq('completed', false)  
    .count();  
  const activeTaskText = activeTaskCount === 1 ?  
    '1 item left' :  
    `${activeTaskCount} items left`;  
  $w('#activeTasksCount').text = activeTaskText;  
}
```

יצרתם פונקציה שמעדכנת את מספר המשימות 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 קיימת – בסוף ה-`checkbox_change` event –

```
export async function completedCheckbox_change(event) {  
    // completed update code  
  
    await updateActiveTaskCount();  
}
```

3. כאשר העמוד סים להיטען בעזרת `$w.onReady`

\$w.onReady()

[https://www.wix.com/corvid/reference/\\$w.html#onReady](https://www.wix.com/corvid/reference/$w.html#onReady)

זהה 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` `onReady` כך:

```
$w.onReady(function () {  
    updateActiveTaskCount();  
});
```

וברים ל-`preview`.

- מודדים שהמספר הראשוני שמופיע נכון
- מוסיפים משימה חדשה ומודדים מספר המשימות שלא בוצעו עולה ב-1
- מסמנים משימה שלא בוצעה ומודדים מספר המשימות שלא בוצעו יורד ב-1
- מורידים סימון למשימה שבוצעה ומודדים מספר המשימות שלא בוצעו עולה ב-1

עובד? שיגעון! הלאה!

סינון המשימות לפי סטטוס המשימה

עכשו מסננים את המשימות שהמשתמש רואה לפי הסינון שהוא בcptori הרדיו (radio button) שנמצאים מעל רשימת המשימות:



למידע נוסף על רכיב כפתיי הרדיו:

[https://www.wix.com/corvid/reference/\\$w.RadioButtonGroup.html](https://www.wix.com/corvid/reference/$w.RadioButtonGroup.html)

להלן הנתנות הרצואה עבור כל אחד מהכפרורים:

- All Tasks

- רק המשימות שהושלמו (שהמשתנה completed שליהם שווה ל-true) יופיעו

- המשימות שעדיין לא הושלמו (שהמשתנה completed שליהם שווה ל-false) יופיעו

מכיוון שהסינון יבוצע לאחר שהמשתמש ילחץ על אחד הcpfutors, ה-event המתאים הוא onChange – מוסיפים אותו כמו שמלודתם קודם: radioButtonGroup.onChange

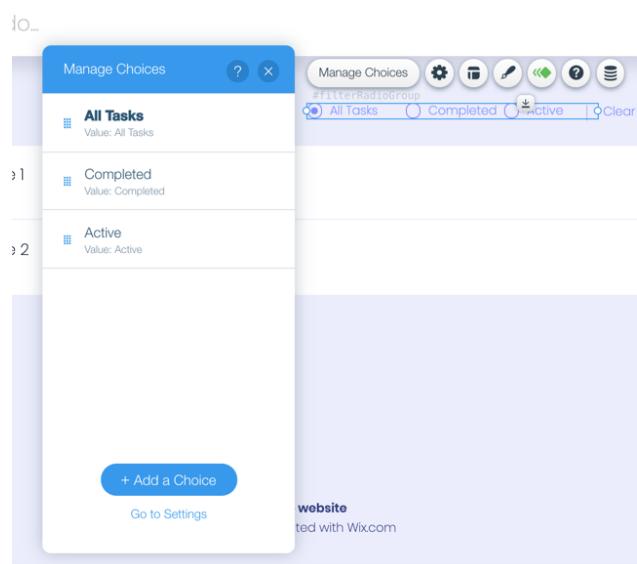
```
export function filterRadioGroup_change(event) {  
    //Add your code for this event here:  
}
```

למידע נוסף על onChange event של כפתיי הרדיו:

[https://www.wix.com/corvid/reference/\\$w.RadioButtonGroup.html#onChange](https://www.wix.com/corvid/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 הערכים בלחיצה על .Manage Choices



על מנת לקבל את הערך שנבחר על ידי המשתמש, קוראים לערך ה-value שלו חלק מכל כפטור רדיו, וכשמו כן הוא – הנתון (או הערך) שהוא רצים לקבל מהמשתמש.

```
$w('#filterRadioGroup').value
```

למידע נוסף על משתנה ה-value של כפתורי הרדיו:

[https://www.wix.com/corvid/reference/\\$w.RadioButtonGroup.html#value](https://www.wix.com/corvid/reference/$w.RadioButtonGroup.html#value)

עכשו מבצעים סינון של המידע שה-dataset מביא אל העמוד בהתאם לבחירה של המשתמש.

סינון מיידע מה-dataset נעשה בשלישה שלבים:
1. אתחול המנסן ()
.wixData.filter()

2. בניית הפקודות שמנדרות את הסינון.
3. הפעלת הסינון על ידי העברת של אובייקט הסינון שיצרתם ל-.dataset

wixData.filter() .1

<https://www.wix.com/corvid/reference/wix-data.html#filter>

מייצר אובייקט סינון, שמועבר ל-.dataset, ובכך מאפשר לסנן את המידע שה-dataset מביא לעמוד:

```
let filter = wixData.filter()
```

2. בניית הסינון

לאחר מכך בונים את הפקודות של ה필טר על ידי שימוש באותו פונקציית שמנדרות שאילתת.
במקרה זהה צריכים ליצור שני סוגי של פילטר:

כאשר המשתמש לוחץ על Completed, רוצים לסנן את כל המשימות שלא הושלמו ולהציג רק את המשימות שהושלמו. בעצם רוצים להشير את כל המשימות שערך ה-completed שלhn שווה ל-true.
לצורך זה משתמשים ב-eq (בדומה לשאליתה 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 להشير את כל המשימות שערך 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();
```

dataset.setFilter() – הפעלה הסינון על ה- dataset.setFilter()

<https://www.wix.com/corvid/reference/wix-dataset.Dataset.html#setFilter>

dataset.setFilter() היא פונקציה אסינכרונית שמבצעת את הסינון על רכיב ה- dataset. היא מקבלת ארגומנט אחד – אובייקט הфиילטר שיוצרת עד עכשיו. לאחר שקוראים לפונקציה זו, ה- dataset יבצע קריאה מוחדשת למשימות מהטבלה, אבל ללא משימות שאין עונות על קритריון הסינון.

```
await $w('#tasksDataset').setFilter(filter);
```

עכשו מחברים הכל יחד ל- event onChange של ה- filter:

```
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 לפונקציית הסינון רק כאשר המשתמש לוחץ על אחד מכפתורי הסינון. יש לסנן את המשימות בעוד מצבים בקורס. לכן מתחילהם להעביר את כל קוד הסינון לפונקציה ייעודית בשם :filterTasks

```
async function filterTasks() {
  const filterValue = $w('#filterRadioGroup').value;
  // create filter code
  await $w('#tasksDataset').setFilter(filter);
}
```

עכשו, קוראים לה מכל מקומות השונים שדורשים אף הם סינון:

1. כאשר המשתמש לוחץ על אחד הכפתורים מה-radio group buttons (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:

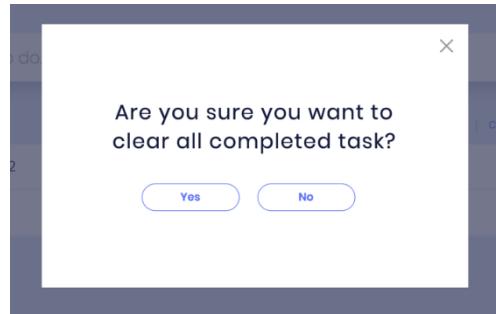
```
async function addNewTask() {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    // add new task
    await updateActiveTaskCount();
    await filterTasks();
  }
}
```

ובבירם ל-preview, "מENCHIM" עם הסינון שזה עתה נכתב, משנים סטטוס משימות כאשר נמצאים על סינון מסוים ומודדים שהן נשאות או נעמלות ולבסוף, מוסיפים משימה חדשה כאשר נמצאים במצב סינון של משימות שלא הושלמו ומודדים שהמשימה לא מופיעה ברשימה.

שמחה ושבון! יש סינון! ☺

ביקוי המשימות שהושלמו

עכשו מממשים את כפתור ה-Clear completed, 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 עבורכם וקראתי לו Clear Confirmation. תמצאו אותו ב- Site Structure . מתחת ל-lightboxes .

מוסיפים event של לחיצה על כפתור ה- Clear completed : (properties כמו של מודול (דרך 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/corvid/reference/wix-window.html>

wix-window הוא מודול שמאפשר עבודה מול חלון הדף שנעליו עובדים .

על מנת להשתמש במודול, יש ליבא אותו מהאזור העליון של קוד קובץ הקוד (בדומה ליבוא של wix-data):

```
import wixWindow from 'wix-window';
```

wixWindow.openLightBox()

<https://www.wix.com/corvid/reference/wix-window.html#openLightbox>

המודול wixWindow תומך בפתיחת של חלון חדש על ידי קריאה לפונקציה האסינכרונית openLightBox , שמקבלת שני ארגומנטים :

1. שם החלון שאותו רוצים לפתוחה. במקרה זה – ."Clear Confirmation"
2. האובייקט שורוצים להעביר לחלון. במקרה זה אין צורך בכך.

ערך ההזורה של ה-promise יהיה האובייקט שיווהר על ידי החלון (במידה שקיים).

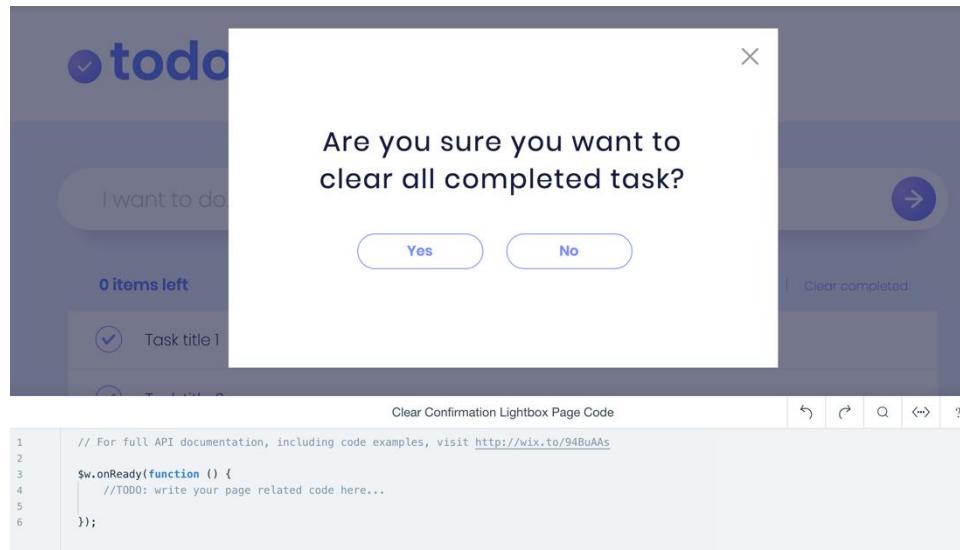
לכן, על מנת לפתח את החלון כותבים את הקוד:

```
export async function clearCompletedButton_click(event) {
  const shouldClearCompleted = await wixWindow.openLightbox('Clear Confirmation')
}
```

ועוברים לו-preview כדי לוודא שהחלון אכן נפתח.

כעת רוצים לדעת אם המשתמש לחץ על Yes או על No, כדי לדעת איך לפעול. ככלומר, רוצים שההתשובה שותחוור מה-promise כשהחלון יסגר תהיה ערך בוליאני, שאם הוא חיובי – יימחקו המשימות שהושלמו ואם הוא שלילי – לא יקרה דבר.

עוברים לחלון ה-.event כדי להחזיר תשובה ל-.clearConfirmation.



גם פה, כמו בעמוד ה-HOME שלכם, אפשר לכתוב קוד בעורך הקוד שבתחתית המסך.

ועכשיו, מוסיפים event להחיצה על כפתור ה-Yes:

```
export function approveBtn_click(event) {
  //Add your code for this event here:
}
```

ומוסףים event להחיצה על כפטור ה-No:

```
export function rejectBtn_click(event) {  
    //Add your code for this event here:  
}
```

כעת רוצים לסגור את החלון בלחיצה על כל אחד מהכפתורים. מתחילה ביבוא המודול wix-window

לחלק העליון של הקוד:

```
import wixWindow from 'wix-window';
```

wixWindow.lightbox.close()

<https://www.wix.com/corvid/reference/wix-window.lightbox.html#close>

wixWindow.lightbox.close() היא פונקציה שסגרת את החלון שבו נמצאים ומחזירה את המשמש לעמוד שפתח את החלון. הפונקציה זו מקבלת ארגומנט אחד: ערך שינון כתשובה של ה-promise שהזר .wixWindow.openLightbox()-.

לכן, בלחיצה על Yes מחזרים true ובלחיצה על No מחזירים :false

```
export function approveBtn_click(event) {  
    wixWindow.lightbox.close(true);  
}  
export function rejectBtn_click(event) {  
    wixWindow.lightbox.close(false);  
}
```

אש! חוזרים לעמוד ה-HOME כדי לבצע את המהיקה מיידת שהמשמש ילחץ על Yes.

מיידת שהמשמש בחר למחוק את המשימות שהושלמו, מבצעים את המהיקה בשני שלבים:

1. שאלתה שתחזיר את כל המשימות שערך ה-completed שלהן הוא true.

2. מהיקה של כל המשימות שהתקבלו מהשאליטה.

למ长时间 מוקדם יותר איך לבצע שאליתה דומה, אבל בשאליתה הקודמת רציתם לקבל את מספר המשימות ולכן השתמשם ב-.count(). הפעם תשימושו ב-.find().

```
const queryResult = await wixData.query('Tasks')  
.eq('completed', true)  
.find();  
const completedTasks = queryResult.items;
```

למידע נוסף על פונקציית ה-`find`:

<https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#find>

כשקוראים ל-`find()`, חזר או ביקט שמכיל כל מיני דברים. אחד מהם הוא ה-`items` שמחזיק מערך של כל הערכים שהזרו – במקרה זהה, מערך של כל המשימות שהזרו מהשאילתת.

כעת עוברים על כל אחת מהמשימות הללו ומוחקים אותן באמצעות:

מחיקת רשומה מהטבלה – `wixData.remove()`

<https://www.wix.com/corvid/reference/wix-data.html#remove>

מחיקת רשומה מהטבלה נעשית על ידי שימוש בפונקציה האסינכרונית `remove()`, שמקבלת שני ארגומנטים:

1. שם הטבלה שממנה רוצים למחוק את הרשימה – במקרה זה `Tasks`.
2. ה-`id` של הרשימה שאותה רוצים למחוק.

זכרים את מזהה ה-`_id` שנכנס באופן אוטומטי לכל שימוש בטבלה? מעולה!
או עכשו תשתמשו בו.

מכיוון ש-`remove()` היא אסינכרונית, וצריך לעבור על מערך של משימות שהזרו מהשאילתת, משתמשים ב-`Promise.all()` כדי לבצע את המחיקה:

```
await Promise.all(completedTasks.map(task => wixData.remove('Tasks', task._id)));
```

ולקינוח, קוראים שוב לפונקציה `filterTasks()`, כדי להסיר מהציגה את המשימות שנמחקו.

ולבסוף 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;

    await Promise.all(completedTasks.map(task =>
      wixData.remove('Tasks', task._id)));
    await filterTasks();
  }
}
```

ועכשו ל-preview. צריך לוודא שיש כמה שימושות שהושלמו וכמה משימות שלא הושלמו ולבצע שתי בדיקות שונות:

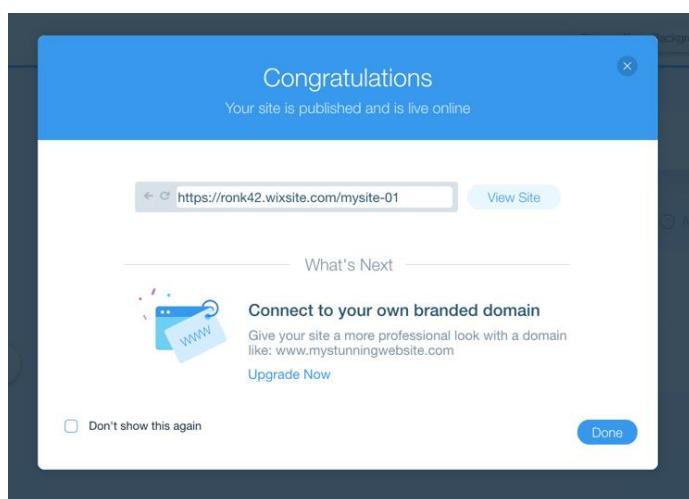
1. לחיצה על Clear completed ולאחר מכן לחיצה על No => יש לצפות לכך שאין דבר לא יקרה.

2. לחיצה על Clear completed ולאחר מכן לחיצה על Yes => יש לצפות שככל המשימות שהושלמו יימחקו.

משם כמעט סיימת!

זכרים את כפתור ה-publish מתחילת הפרק?
עכשו הזמן להוציא עליון!

אחרי לחיצה על כפתור ה-publish תהיה האפליקציה שלכם זמינה באתר לכל דבר ולכל העולם!
הכתובת של האפליקציה שלכם תוצג בחלון שייפתח לאחר שתלחצו על publish:



* אחרי שפרסמתם את האפליקציה, המשימות שהוספთ בזמן שעבדתם על האפליקציה לא יופיעו ב-preview, מכיוון שתתמודד עם מסדי נתונים שונים עבור פיתוח ועבור האפליקציה האמיתית.

נספח – יצירה מסד נתונים

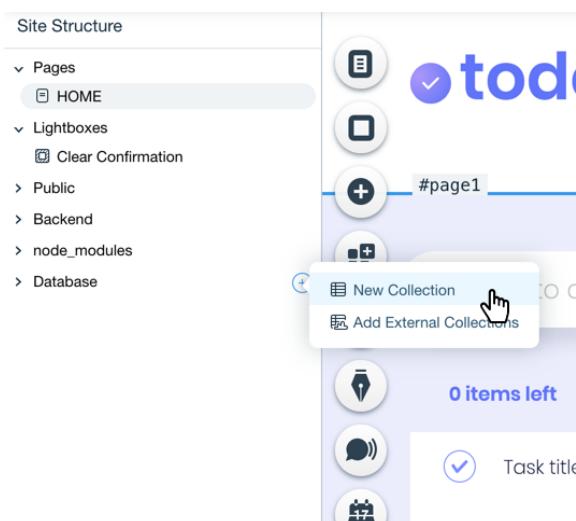
מסד נתונים הוא אמצעי המשמש לאחסון מסודר של נתונים. במקרה של Corvid, הנתונים הללו מאוחסנים במודול של טבלאות (collections) ובכל טבלה העמודות מייצגות שדות (fields) והשורות מייצגות רשומות שונות.

נוהג לשיקד כל טבלה לישות מסוימת באפליקציה, ואז כל שורה מייצגת מקרה ספציפי של הישות. בנוסף על כך, כל עמודה בטבלה היא בעצם שדה שמייצג מאפיין מסוים של הישות. לכל שדה יש סוג כגון טקסט, מספר, בולייאני, תמונה וכדומה.

הבה נתכנן את מסד הנתונים של האפליקציה.

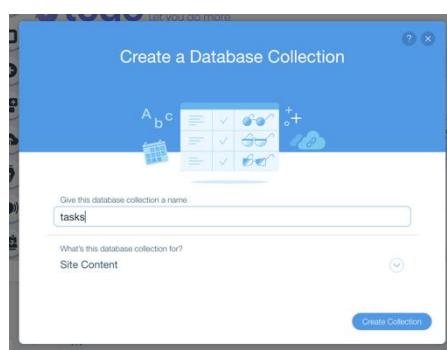
באפליקציה שלכם יש ישות שנ刻画ת משימה (task), ולכן יש ליצור טבלה שנ刻画ת Tasks. כל שורה בטבלה תציג משימה בודדת. על מנת ליצור טבלה חדשה עבורם עם העבר על כפתור +, שיוופיע בעבר על צידו הימני של ה-`database` site structure בפאנל ה-`site structure`.

לאחר מכן לוחצים על ה-`New Collection`:



Create Collection

הבחירה תקפיים מסך שבו נתונים לטבלה את השם tasks ולאחר לחיצה על תיווצר טבלה חדשה.

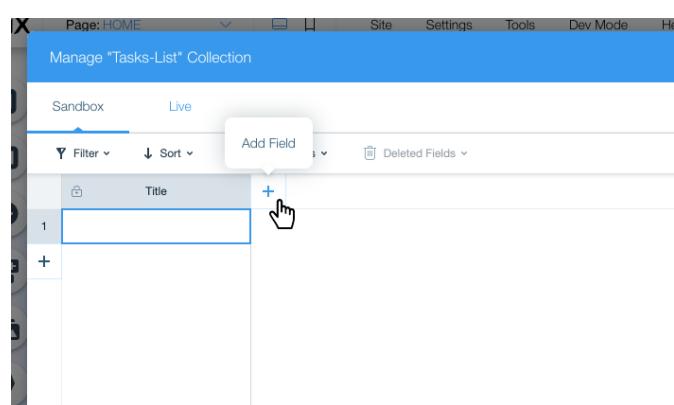


יצירת הטבלה הוסיפה טבלה חדשה ב-site structure מתחת ל-database, ובנוסף העבירה אתכם ישר לאזור שנקרא "עורך התוכן" – content manager – שבו מאפשר להוסיף שדות ולערוך את התוכן של הטבלה.

הגיע הזמן לייצר את השדות. כל משימה מכילה את תיאור המשימה בטקסט וערך בוליани שמאגדיר את סטוס המשימה – אם היא הושלמה או לא. לכן, יש צורך בשני שדות:

- שדה Title מסוג טקסט (text) – שיופיע אוטומטי ברגע שמייצרים טבלה חדשה וכן לא צריכים להוסיף אותו ידנית.
- שדה Completed מסוג בוליאן (boolean).

על מנת להוסיף שדה יש ללחוץ על כפתור הפלוס ואו יפתח פאנל ההגדרות של השדה.

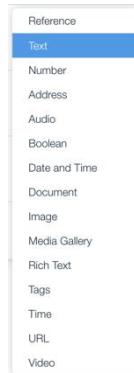


בפאנל ההגדרות יש שלושה קלטים שצרכי למלא כאשר מייצרים שדה חדש:

Add Field	<input type="button" value="X"/>
Field Name	Completed
Field Key	completed
Field Type	Boolean
<input type="button" value="Add"/>	

.Task – שם השדה שרוצים. במקרה זה – Field Name

- **Field Key** – זהו מפתח השדה שנוצר באופן אוטומטי על ידי השם שניתן לשדה. הערך של השדה ישמש אתכם כאשר תרצו לפנות לשדה זהה דרך קוד. לרוב אין צורך לגעת בערך שהמערכת נותנת באופן אוטומטי.
- **Field Type** – סוג השדה שרוצים ליצור. אפשר לבחור ממבחן גדול של סוגי.



כעת קוראים לשדה הנוסף **Completed** ומגדירים אותו מסוג **Boolean**.

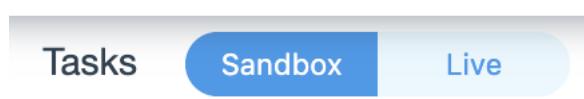
לבסוף להזים על **Add** והשדה יתוסף.

כעת יש טבלה בעלת שתי עמודות. אפשר להוסיף רשומות באופן ידני, לשנות אותן ואפילו למחוק אותן מתוך עורך התוכן.

למדתם את רוב המידע שייהי לרלוונטי לכם עבור מסדי נתונים ב-**Corvid**.uccishו תלמדו על עבודה עם מסד הנתונים לאחר שמספרסמים את האתר.

Sandbox | Live

אם תשימו לב, בחלק העליון של עורך התוכן יש שני כפתורים:



- בלחיצה על כפתור **Sandbox** יוצגו הרשומות שיופיעו בזמן עריכה האפליקציה. כלומר, הרשומות שרואים כאשר להזים על **preview**.
- בלחיצה על כפתור **Live** יוצגו הרשומות שרואים באפליקציה האמיתית אחרי שלוחצים על **publish**.

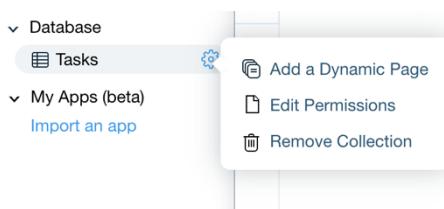
Permissions

לכל טבלה יש הרשאות שגדירות מי יכול לראות את הנתונים שבטבלה, מי יכול ליצור אותם, לעורוך אותם ולמחוק אותם.

הרשאות הראשוניות שמקבלים כשיצרים טבלה חדשה הן מאוד שמרניות. למשל, הן לא ייתנו יותר מדי יכולות למשתמשים חיצוניים. לכן, הפעולות SMBצעים על הטבלה באפליקציה האמיתית (לא ב-preview) יחוירו שגיאה:

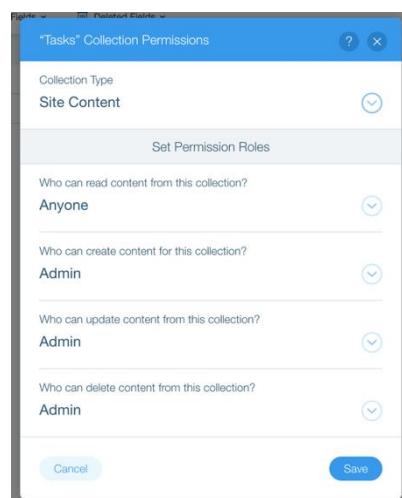
wixData.get	-
wixData.insert	-
wixData.update	-
wixData.delete	-

כדי למנוע את זה, מוטב לשנות את הרשאות. עושים זאת באמצעות לחיצה על **Edit Permissions**:



ואז, יפתח חלון שבו תוכלו לעורך את הגדרות של הרשאות. עבור אפליקציית המשימות שלכם, תרצו שההרשאות ייראו כך:

* בתבנית שיצרתי עבורכם כבר הגדרתי את הרשאות של הטבלה בצורה הזו.



סיכום

סיימתם ל כתוב את האפליקציה שלכם. מה למדתם?

- חיבור מידע מטבלאות הנתונים לאפליקציה
- ייצירת אינטראקטיות עם רכיבים של העורך הווייזואלי דרך הקוד
- צפיה, הוספה, עריכה ומחיקה של רשומות בטבלאות בעזרת wix-data
- ייצרת שאלות וסינונים על הרשומות
- פתיחה חלונית וסגירתה באמצעות קוד
- מי שקרה את הנספה – איך ליצור טבלאות חדשות

עכשו הגיע הזמן שלכם ליצור **אפליקציות חדשות**, ללמידה על עוד יכולות Sh-Corvid מספקת לכם ולהתנסות בהן.

תודה רבה על הרכישה של הספר!

עבדתי מאד קשה על הספר זהה: שעות רבות של כתיבה, הגאה, תיקונים ומעבר על תוצרי העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. ככלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקורא גם מקינדל, 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 להעתיקו סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

danielast4@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - ככלומר בתוך דפי הספר נחאים פרטי הרוכש באופן שקויף למשתמש. כדי למנוע מהעתיקה של הספר לאלו שלא רצשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומייחקו את העתק שנמצא ברשותכם.

תודה וקריאה נעימה!