

תודה רבה על הרכישה של הספר!

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

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מחשב וגם טלפון נייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והתוםן לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאשר שרוב האנשים הוגנים.

העותק הזה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נჩבים פרטי הרוכש באופן שkopf למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העברו לו את הפרטים שלכם באתר ומיחקנו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!

ללמוד גיאו-הסקרייפט בעברית

נן בר-זיך

WixEngineering

Outbrain
Engineering



Really Good

Chegg[®]

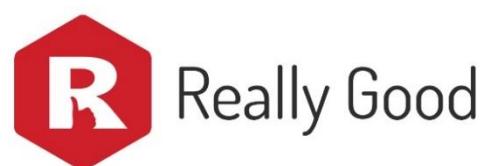


הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

לلمוד ג'אווהסקרייפט בעברית

REN BER-ZIK

מהדורה: 2.2.1



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

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

ਮותר לך לצלט קטעים קצרים מהספר במסגרת הגנת השימוש 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 בתוקן תוכנות שתפתח. אם אתה רוצה להכניס אותן לפרויקט שלך, שלח מייל ונדבר על זה.

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

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



תוכן העניינים

| | | |
|-----------|---|---------------|
| 12 | על הספר | ... 12 |
| 13 | על המחבר | ... 13 |
| 14 | על העורכים הטכניים | ... 14 |
| 14 | דניאל שטרנלייבט | ... 14 |
| 14 | gil pinck | ... 14 |
| 14 | יגאל סטקלוב | ... 14 |
| 15 | תום ביגלאייזן | ... 15 |
| 15 | zechi nemni | ... 15 |
| 16 | על החברות התומכות | ... 16 |
| 16 | Really Good | ... 16 |
| 16 | אואטבריין | ... 16 |
| 17 | Chegg | ... 17 |
| 18 | Wix.com | ... 18 |
| 19 | על ג'אווהסקריפט | ... 19 |
| 22 | אין לומדים | ... 22 |
| 22 | ארגנו לעצמכם סביבת עבודה מסודרת | ... 22 |
| 22 | קראו את הפרקים לפי הסדר | ... 22 |
| 23 | קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות | ... 23 |
| 23 | פתרו את התרגילים לדוגמה בסוף כל פרק | ... 23 |
| 23 | התיעמצו עם אחרים | ... 23 |
| 24 | הגיעו לסדרניות ולMITAPIים | ... 24 |
| 24 | תרגלו, תרגלו, תרגלו | ... 24 |
| 25 | התקנת סביבת העבודה ודרך הלימוד | ... 25 |
| 35 | משתנים | ... 35 |
| 40 | טקסט | ... 40 |
| 46 | מספרים | ... 46 |
| 49 | מציאת השארית | ... 49 |

| | |
|-----------------|---|
| 50..... | אופרטורים מקוצרים |
| 56..... | סוגי מידע פרימיטיביים נוספים |
| 56..... | بولיאני |
| 57..... | משתנה לא מוגדר |
| 57..... | רייך |
| 58..... | Symbol |
| 58..... | מציאת הסוג של המשתנה |
| 62..... | הערות |
| 65..... | בקרת זרימה – משפטי תנאי |
| 68..... | אופרטורים השוואתיים נוספים |
| 72..... | אופרטורים לוגיים |
| 75..... | אופרטור שלילי |
| 77..... | אופרטור תנאי |
| 78..... | אופרטורים המשווים ערכים |
| 90..... | switch case |
| 100..... | קבועים |
| 104..... | בקרת זרימה – פונקציות |
| 109..... | פונקציה עם ארגומנטים |
| 111..... | ארגומנטים עם ערכים דיפולטיבים |
| 113..... | הפונקציה כאובייקט |
| 113..... | Hoisting |
| 115..... | closure |
| 118..... | פונקציה אוניבימית ופונקציית חץ |
| 119..... | פונקציה אוניבימית במשתנה |
| 121..... | פונקציה אוניבימית שمبוזדת מהסקופ הגלובלי |
| 129..... | אובייקטים |
| 135..... | מחיקת מפתח |
| 135..... | הכנסת פונקציה כערך |
| 136..... | קינון ובדיקה מפתח |
| 137..... | אובייקט קבוע |
| 146..... | מערכות |

| | |
|----------|---|
| 151..... | מערכות ומחזרות טקסט |
| 154..... | this-i new |
| 162..... | מבנה טקסט |
| 167..... | ולאות |
| 167..... | ולאות for |
| 178..... | ולאה נוספת |
| 178..... | ולאות while |
| 180..... | ולאות do while |
| 180..... | ולאות forEach |
| 184..... | ולאות for of |
| 189..... | ולאות map |
| 192..... | ולאות filter |
| 196..... | ולאות sort |
| 196..... | ולאות על אובייקטים |
| 197..... | ולאות for in |
| 200..... | Object.keys |
| 209..... | ג'אווהסקריפט בסביבת דפדף |
| 209..... | הסבר כללי על HTML |
| 212..... | מזהים של תגיוט HTML |
| 213..... | גישה אל תגיוט באמצעות ג'אווהסקריפט |
| 213..... | אלמנטים של HTML מתרגמים לאובייקטים של ג'אווהסקריפט |
| 215..... | אירועים עם HTML וג'אווהסקריפט |
| 217..... | הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווהסקריפט |
| 219..... | שינוי עיצוב |
| 220..... | סלקטורים של DOM |
| 225..... | אירועים נוספים |
| 227..... | פעוף של אירועים |
| 231..... | הצמדת אירועי ג'אווהסקריפט לאלמנטים ב-HTML |
| 240..... | דיבאגינג |
| 247..... | אובייקטים גלובליים ואובייקטים מובנים |
| 251..... | parseInt |

| | |
|----------|--------------------------------------|
| 252..... | eval |
| 253..... | Math |
| 254..... | Date |
| 257..... | JSON |
| 259..... | setTimeout |
| 265..... | ביטויים וגלאריים |
| 273..... | טיפול בשגיאות |
| 276..... | finally |
| 280..... | מבנה נתונים מסוג Set-Map |
| 280..... | Map |
| 283..... | Set |
| 288..... | תכנות אסינכרוני – קולבקים |
| 298..... | Promises |
| 304..... | שרשרת הבטחות |
| 307..... | קיבוץ הבטחות |
| 308..... | קיבוץ הבטחות מתקדם |
| 313..... | פונקציית async |
| 320..... | AJAX |
| 322..... | METHODS של HTTP וארגוניים נוספים |
| 327..... | ES6 Classes |
| 332..... | METHODS ותכונות פרטיות |
| 336..... | ומה עכשווין? |
| 338..... | נספח: לימוד טייפסקריפט בעברית |
| 342..... | קביעת סוג הקלט של פונקציה |
| 345..... | הגדרת סוג מידע של משתנה |
| 345..... | הגדרת סוג מידע של פלט של פונקציה |
| 347..... | מערכות |
| 349..... | אובייקט עם סוג מידע פרימיטיבים |
| 349..... | סוג מידע: פונקציה |
| 352..... | במה סוג מידע אפשרי |
| 354..... | Type\ interface |
| 356..... | אסטטינג |
| 357..... | גרירות |
| 364..... | טייפסקריפט עם קלאסים |
| 364..... | METHODS, מוגנות ופומביות |
| 365..... | קריאה בלבד |

| | |
|------------------|---|
| 367..... | קלאס אבסטרקטוי |
| 369..... | סביבת עבודה אמיתית של טייפסקריפט |
| 379..... | סיכון |
| 380..... | תרגילים |
| 383 | נספח: Best Practices |
| 383..... | מה זה Best Practices וולמה בדאו לישם אותם? |
| 385..... | שכל מפתח מקצועני צריך להכיר Best Practices |
| 385..... | בחירה שמות |
| 387..... | ... KISS |
| 388..... | ... DRY |
| 388..... | לא להמציא את הגלגל |
| 389..... | Make it work, make it right, make it fast |
| 390..... | תיעוד |
| 390..... | ניהול גרסאות |
| 391..... | לבקש עזרה |
| 392..... | ביקורת עמיתים – Code Review – |
| 393..... | Tech Design |
| 393..... | Best Practices ואוטומציה |
| 395..... | מה זה ?linting |
| 396..... | ESLint |
| 397..... | רשימה של Best Practices והחוק הרלוונטי של ESLint |
| 397..... | בלי מספרי קסם |
| 398..... | השוואה קפדיות: === |
| 398..... | קוד לא נגיש |
| 399..... | חלוקת לחלקים קטנים |
| 400..... | אורק מינימלי ומקסימלי לשמות משתנים |
| 400..... | בלי eval |
| 401..... | בלי משתנים שלא הוציאו |
| 402 | נספח: בדיקות, יציבות ואיוכות קוד |
| 402..... | קצת רקע |
| 403..... | מבנה בדיקות |
| 404..... | בדיקות ייחודיה |
| 405..... | בדיקות קצה לקצה (End-to-End) |
| 406..... | בדיקות ממשקי משתמש (UI Tests) |
| 407..... | ספריות ופרימורוקים מומלצים |
| 407..... | סיכון |
| 409 | נספח: Velo by Wix |
| 409..... | הקדמה |
| 409..... | מה בונים? |
| 411..... | איך מתחילהים? |
| 414..... | הציג נתונים מסך הנתונים באתר |
| 415..... | Dataset |

| | |
|------------------|---|
| 416..... | חיבור רכיבים למידע מהטבלה |
| 419..... | הוספת משימה חדשה |
| 419..... | Events |
| 421..... | \$w |
| 422..... | wix-data |
| 423..... | – הוספה רשומה לטבלה wixData.insert() |
| 426..... | שינוי סטטוס המשימה |
| 427..... | – wixData.get() – שילוף רשומה מהטבלה |
| 428..... | – עדכון רשומה בטבלה wixData.update() |
| 430..... | מספר המשימות שלא הושלמו |
| 430..... | wixDataQuery |
| 434..... | \$w.onReady() |
| 434..... | סינון המשימות לפי סטטוס המשימה |
| 436..... | wixDataFilter |
| 439..... | ניכוי המשימות שהושלמו |
| 440..... | wix-window |
| 441..... | wixWindow.openLightBox() |
| 442..... | wixWindow.lightbox.close() |
| 443..... | – wixData.bulkRemove() – מחיקת רשומה מהטבלה |
| 446..... | תוספה – יצירה מסד נתונים |
| 449..... | Sandbox Live |
| 450..... | Permissions |
| 451..... | סיכום |
| 452 | נספח: ג'אווהסקריפט מונחה עצמים |
| 452..... | מה זה תכנות מונחה עצמים? |
| 460..... | Prototype based |
| 467 | נספח: שינויים מהמהדורה הקודמת (מהדורה 2.1.0) |

על הספר

הספר "לימוד ג'אווהסקריפט בעברית" מלמד את השפה ג'אווהסקריפט (JavaScript), שפת סקריפט קלה ופושא ללימוד שהפכה לפופולריות מאוד עם השנים. הספר מיועד ללא-מתכנתים שרצו למתכנתות בשפת תכנות ולמתכנתים בג'אווהסקריפט שרצו להעמק את הידע התיאורטי שלהם.

הספר מתחילה בלמידה בסיסי של משתנים ומגיע עד לימוד תכונות אסינכרוני-AJAX. בכל פרק בספר יש הסברים תיאורתיים לצד דוגמאות ורבות הממחישות את העקרונות השונים שהוסברו בו, ובסיומו תמצאו שאלות דוגמה לתרגול עם הסברים מקיפים על הפתרונות. בנוסף על הספר קיים אתר המכיל מאות תרגילים ופתרונות אשר יסייעו לכל הלומדים להציג שליטה בעקרונות השפה ובהתקביר שלה. הספר מיועד להביא אדם שלא מכיר את ג'אווהסקריפט לנקודה שבה הוא מכיר היטב את השפה ואת התקביר שלה וודע איך להשתמש בה כדי לפתור בעיות בסיסיות.

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

בספר יש נספח מפורט עם תרגילים המלמד על טייפסקריפט, שבנוי על ג'אווהסקריפט ונעשה בה שימוש בסביבות מודרניות רבות.

על המחבר

REN BAR-ZIK הוא מפתח תוכנה משנת 1996 ב מגוון שפות ופלטפורמות ועובד כ מפתח בכיר במרכזי פיתוח של חברות רב-לאומיות, HPE ו עד Verizon, שם הוא מפתח בטכניקות מתקדמות הן הצד הלקוח, והן הצד השירות, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-C\CDI ומכובן על אבטחת מידע. בנוסף על עבודתו כ מפתח במשרה מלאה, REN הוא עיתונאי ב"זה מפרק" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטכנולוגיה ולאבטחת מידע וכותב על אינטרנט ורשות. בשנת 2008 מפעיל REN את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרים וסבירים על תכונות בעברית ומתעדכן לפחות פעמי שבוע. REN נשוי ליעל ואב לארבעה ילדים: עומר, כפיר, דניאל ומיכל. רץ למרחקים ארוכים וחובב טולקון מושבע.

על העורכים הטכניים

דניאל שטרנלייכט

דניאל שטרנלייכט הוא מפתח Frontend מאז גיל 14, המיסיד של המיזמים Common Ninja-*I* and *-That*, There is a bot for that, צרכן כבד של קוד Open Source (ומשתדל גם לתרום בחרזה) ונכון לזמן כתיבת הספר מוביל את גילדת Frontend בחברת אוטבריין. בנוסף על כן, דניאל הקים ומוביל את קבוצת הווטסאפ FEDs Community שמאגדת מפתחי Frontend מחברות מובילות בארץ ובעולם, ובה מעליים זיוניים, מתיעצים ומשתפים לינקים, חדשות ועדכונים מעולם Frontend. בעבר כתב בפייסבוק "עיצוב 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 ומתח ומדקלם את כל סרטי דיסני בעל פה.

גיל פינק

gil fink הוא מומחה לפיתוח מערכות ווב, Google Developer Expert, MVP Microsoft Developer Technologies, sparcXsys מהיסיד של חברת MVP. כיום הוא מייעץ לחברות ולארגוני שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות לייחדים לחברות המעוניינות להתמחות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official Course - MOC), מחבר משותף של הספר "Single Page Application Development Pro" של אוניברסיטת אוניברסיטת מילו (Apress). לפרטים נוספים עלgil פינק <http://www.gilfink.net>.

יגאל סטקלוב

יגאל סטקלוב הוא מפתח Frontend-*I* and Full Stack, מוביל טכנולוגי ומנהל פיתוח מנוסה בעל ותק של יותר מעשור וחצי בתעשייה. כיום משמש מנכ"ל חברת Webiya.

בעבר היה ממוביל בתחום Frontend בחברות Wix ו-Netcraft וஹוביל פרויקטי פיתוח רבים.

יגאל פועל מאוד למען קהילת ה-Frontend בארץ והוא אחד מהיזמים והמארגנים של כנס ה-Frontend הבינלאומי הראשון בישראל, כנס You Gotta Love Frontend (YGLF).

תומ ביגלאין

מפתח Frontend ומצב, נשוי, אב לשתיים וגור בתל אביב.
תומ החל את דרכו בתחום ה-Frontend בתחילת שנות האלפיים מכיוון פחות צפוי –
בפיתוח אפליקציות בגל'אווהסקריפט לממויות של יס סטארט-אפ קטן. אחרי זה נסגר,
עבד בעצמائي ובכמה סטארט-אפים, הפך לאחד המומחים הראשונים בארץ ל-HTML ו-
CSS ולא היה בקיא ממן בబאים של CSS עם זה באקספלורר 6. תרם לפרויקטים שונים
בקוד פתוח, היה שותף בגין פלטפורמת ניהול התוכן דרוול ובנה את אחד הטמליטים
העבריתים הפופולריים באותה תקופה. תומ גם עיצב את האיקונים שעדיין נמצאים
בשימוש עד היום בแกן הוידיאו הפופולרי בקוד פתוח VLC. לפני קצת יותר משבע שנים
נחת ב-Wix, ובחמש השנים האחרונות עומד בראש צוות קטן ומוכשר שמוביל את תחום
ה-Frontend במדיה – וידיאו, תמונות, וקטורים וAnimatzions בפלטפורמת בניית האתרים של
.Wix

צחי נמני

צחי נמני הוא מומחה בפיתוח, סרבר, אוטומציה ודב אופס
מתמחה בטכנולוגיות הבאות: Java, Kotlin, Node.js, C#, Docker, Kubernetes, Git, Terraform, Appium, Cypress
כיום הוא עובד בחברת הייטק בתחום התἷירות. בנוסף, צחי מייעץ ומרצה לחברות וליחידים
בנושאים של CI/CD, Automation Development, Docker, Kubernetes Docker and Kubernetes
ברגע הוא עובד על סדרה של הרצאות בנושא CI/CD ועוז. צחיאמין גדול בקוד פתוח ותרם לפרויקטים שהוא משתמש בהם.

על חברות התומכות

Really Good

היא בוטיק פיתוח Front End שעובדת עם סטארטאפים וחברות טכנולוגיה מאז הקמתה ב-2012 על ידי שחר טל ורוני אורבן. אנחנו ננים לבנות אפליקציות מורכבות עם UX מוקפם במגוון טכנולוגיות לקווות מעניינים שחוויות המשתמש חשובות להם, ושומרים על איזון בריא בין עבודה לחיים.

אנחנו מגיסים מפתחי Front End מנוסים וממש טובים עם תשומת לב לפרטים הקטנים.

<https://reallygood.co.il>

אאוטבריין

אאוטבריין נוסדה בשנת 2006, על ידי זוגbizים ירון גלאי (כו"ם CEO) ואורי להב (CTO) הנוכחי ומנהל כללי- אאוטבריין ישראל). החזון המרכזי של החברה הוא לייצר את Feed התוכן האמין והמשמעותי ביותר. מוצר המלצות המרכז שאאוטבריין פיתחה, מbasס טכנולוגיית machine learning ומודלים שחוצים את הרגלי צrichtת התוכן של הגולש על בסיס התנהגוות חוזרת. באמצעות כן, האלגוריתם לומד את הגולש ושולח לו המלצות תוכן בזמן אמיתי. בין היתר, אאוטבריין פיתחה מוצרים טכנולוגיים נוספים, ה Outbrain Amplify ו Outbrain Smartfeed MSN, CNN, BBC, Sky News The Washington Post, The Guardien, Spiegel online, El Pais החדשנות והידיעות של אתרי התוכן והערוצים המוביילים בעולם לרבות,

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

קישורים:

<https://twitter.com/OutbrainEng>

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

<https://www.youtube.com/channel/UCJLORR2uJglrKm-JIKV-rJA>

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

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

Chegg

סטודנטים בארה"ב, כמו בכל מקום בעולם, רוצים לרכוש השכלה גבוהה כדי למצוא עבודה טובה, מעניינת וcompatibile. רק שלימודים גבוהים עלולים כסף – והרבה. אז איך מושגים עבודה טובה כשלכתהילה אין מספיק כסף למן תואר ראשון או שני ולהמשיך לחיות תוך כדי? כאן בדיק Chegg נכנסת לתמונה.

Chegg היא חברת אמריקאית שחברת השכלה על דגלה לעזור לסטודנטים במהלך לימודיהם ומכוונת להפוך את הצורך בהשכלה לאפשרי ומשתלם לכל אחד. את השם היא בחרה במלוא אותה דילמה, והוא למעשה הלotta של שתי המילים -Chicken-Egg- שמסמלות את השאלה העתיקה ביותר בעולם הפילוסופיה – מה בא קודם: הביצה או התינוקת? הלימודים האקדמיים או הניסיון בשוק העבודה?

החברה שמה את הסטודנט ואת הצרכים שלו בראש, ובהתאם- "Students First" הוא אחד הערכיהם המרכזיים לאורם פועלת החברה. Chegg עוזרת לסטודנטים לא רק להיות מצטיינים במהלך הלימודים, אלא גם אחריהם, בתהליך חיפוש העבודה. היא משפרת את התוצרים של הסטודנט האמריקאי מגוון שירותי: מהוות קנית ספרי לימוד לצורכי שימוש, משמעותית, דרך הצעת שירותי ויישומים לשיפור הזיכרון במידה ל מבחנים ומומחי תוכן שימושיים מרוחק, ועד עזרה במימון הלימודים, מציאת המקום להתמחות לציבור ניסיון ואפילו תכנון של קריירה והשלמת מיזמים עסקיים שיחסורם באקדמיה.

החברה האמריקאית הוקמה ב-2007, היא חברת ציבורית בינלאומית ועובדים בה למעלה מ-1500 אנשיים. המשרד הישראלי, שමוקם ברוחובות, נוסד בתחילת 2011 והתחל

כפרויקט חיצוני עברו Chegg והיום- המשרד הקטן ברחובות נחשב לחוד החנית של כל תחום הפיתוח של החברה.

Wix.com

Wix.com הינה פלטפורמה גlobלית מובילה לפיתוח נוכחות מקצועית באינטרנט, עם מעליה מ-160 מיליון משתמשים רשומים ברחבי העולם כיום. Wix נוסדה על התפיסה כי רשת האינטרנט צריכה להיות נגישה לכל לפתח, לייצר ולשף תוכן. באמצעות שירותים חינמיים, כמו גם שירותי נוספים למניין פרימיום, מאפשרת Wix למילוני עסקים, ארגונים, אנשי מקצוע ואנשים פרטיים ליצור ולנהל נוכחות מקצועית וдинמית בראשת. Wix ADI, האדיטור של Wix, שוק האפליקציות Ascend by Wix ו-Corvid by Wix מאפשרים למשתמשים לבנות ולנהל נוכחות>Dינמית>Dיגיטלית. מטה חברת Wix ממוקם בתל-אביב ולחברה סניפים נוספים בברא שבע, חיפה, ברלין, סן פרנסיסקו, ניו יורק, מיאמי, לוס אנג'לס, סאו פאולו, וילנה, קייב, ניפורו, דבלין וטוקיו.

במחלקה הפיתוח של Wix Engineering, שותפים מפתחים מובילים אשר מעצבים את הארכיטקטורה של השירותים וה מוצר של Wix. הם קובעים את הטון ואת סטנדרט הפיתוח, תוך שילוב של הבוטי הנדסה, ניהול מוצר, DevOps, ניהול וניהול טכני. הם גם מנטררים, ועוזרים למהנדסים פחות מנוסים לשפר את CISORIHAM, כך שגם הם בסופו של דבר יהיו מסוגלים להיות מנהלים טכניים בעצמם. אנו כותבים בסקירה, ג'אווה, ג'אווהסקייפ, ריאקט, פיתון, גו ועוד. נוסף על כן, אנו מאמינים מאוד בהעצמת קהילת המפתחים בארץ ובעולם, ולהעניק בחזרה על ידי הצגה בכנסים, כמו גם תרומה לקוד פתוח. המהנדסים שלנו גם תומכים ומשתתפים בכל מיני אירועים טכנולוגיים ללא מטרות רווח.

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

על ג'אוהסקריפט

ג'אוהסקריפט החלה את צעדיה הראשוניים ב-1993 כשפה שפועלת בסביבת דפדפן ונוועדה להענישר דפי HTML. בג'אוהסקריפט יכולנו ליצור אנימציות ופידבק למשתמשים – כל דבר שהוא קשור לאתר אינטרנט דינמיים, למשל דברים שנראים היום מאוד טריויאליים ופשוטים כמו לחיצה על כפתור שעושה פעולה כלשהו בדף. אף על פי שההתחלת שלה הייתה צנועה, ברנדון אייך, ממציא השפה, יצר אותה מלבת חילה כשהשפה גמישה מאוד. הnymיות הזהן, וגם חוסר ההבנה של רבים מהמתכנתים שהשתמשו בה בזוגע לעקרונות הבסיסיים שלה, גרמו ללא מעט מתכנתים בשפות אחרות לזלול בה. גם השם שלה לא סייע לתדמיות. השם ג'אוהסקריפט נקבע מסיבות שיווקיות בלבד – JAVA היא שפת תכנות פופולרית, ואנשי נטסקיים חשבו שכך יסייעו לתדמייתה. 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 נטסקיים העבירה את השליטה בסטנדרטים של השפה אל ארגון ECMA, ארגון אירופי (היום בינלאומי) המתחמה בתקינה. המהלך הוביל לשחרור הספסיפיקציה של השפה. שידוע בשם ECMAScript, ג'אוהסקריפט "התישראל" לפי התקינה של ECMAScript. משנת 1997, שנת שחרור ECMAScript, ג'אוהסקריפט, כפי שהיא מושמת בדפדפניים שונים, עוקבת אחר התקינה של ECMAScript, שהיא בעצם "תוכנית המתאר", ג'אוהסקריפט עצמה היא היישום. לכל גרסה יש מספר משלחה בaczmod למיללים ES (ראשי תיבות של ECMAScript).

מיקרוסופט התנגדה בתוקנה ליישום השפה וייסמה שפה משלה בשם Jscript בדפדפן אינטרנט אקספלורר, שהייתה בנוייה בדומה לג'אוהסקריפט. למרות היריבות הגדולה בין אנשי מיקרוסופט לאנשי ECMA, שנוצרה כתוצאה מפיתוח שתי שפות שנשענות על שני תקנים מתחרים, העקרונות של ג'אוהסקריפט שולבו גם בגרסה של מיקרוסופט. הפופולריות של השפה עלה כאשר מקромדי (יצירת פלאש) שיתפה פעולה עם ארגון ECMA ושילבה את עקרונות השפה בשפת Actionscript, ששימשה את תוכנת פלאש שהייתה פופולרית מאוד אז.

בשנת 2008 נפגשו אנשי מיקרוסופט ו-ECMA באוסלו והחלו בשיחות שלום. בינו לבין לשיחות שלום אחריות שהתקיימו באוסלו, שיחות השלום האלו הסתיימו בהצלחה. תקן ES5, הגרסה הרביעית של ג'אווהסקריפט, שוחרר ויושם בכל הדפדפנים שהיו קיימים אז. מאז, התפתחות השפה והתפוצה שלה הווא צורמתית. דפדפן כרום, שMRI, ג'אווהסקריפט באופן יוצא דופן, נכנס אל השוק בסערה ואפשר למפתחי ג'אווהסקריפט לכתוב סקריפטים שפועלים על מנווע 87 העוצמתי של כרום ולהריז ג'אווהסקריפט במהירות מסחררת. השימוש ב-AJAX תקשורת אסינכרונית עם השרת – נכנס לפולה, החליף שיטות מיישנות כגון Long polling ואפשר לאתרים לספק חוות שימושיות מדיהימות למשתמשים. בשנים האחרונות, פרימומורקים וספריות ג'אווהסקריפט אפשרו פונקציונליות מורכבת מאוד וספריות אחרות אפשרו כתיבה של ג'אווהסקריפט גם לטלפונים ניידים ואףilon בклות. הראשונות שבספריות האלו נקראו MooTools ו-jQuery, והן אפשרו לכל מתכנת לכתוב אפליקציות בקלות. הספריות האחרונות נקראות ריאקט, אנגולר ו-אנדゥ והן מאפשרות לבנות תוכנות מורכבות מאוד על גבי הדפדפן (צד הלוקה). ג'אווהסקריפט לא נותרה מוגבלת רק לצד הלוקה, ככלומר לדפדפנים ולמכשירי כזה אחרים; השימוש של ג'אווהסקריפט לצד השרת, הידוע בכינויו `node.js`, הפך לפופולרי גם בשרתים. ג'אווהסקריפט מרחיצה כיום אפליקציות מורכבות גם לצד השרת, במיוחד אפליקציות לצרכים לביצוע קריאות ולשרות מילוני משתמשים. כיום אפשר למצוא ג'אווהסקריפט בכל מקום: באתר אינטרנט, באפליקציות של טלפונים ניידים, באפליקציות המיעודות למחשבים רגילים וכמוון בשרתים. הביקוש למתכנתים ג'אווהסקריפט נמצא בשיאו ואין זה פלא – אפשר לעשות בשפה זו המון דברים יישומיים כמעט מ於是. יש כל כך הרבה ספריות וכל עוז, עד שכמעט בכל שבוע יוצא ספרייה שימושית חדשה. בעזרתו ידע מועט אפשר לעשות הרבה מאוד. מה שחשוב הוא ידע בסיסי בשפה.

בשנים האחרונות, תקן ES מתעדכן בכל שנה ומתווסףים אליו תוכנות ושימושים חדשים. ספר זה מעודכן לגרסה الأخيرة של ECMA Script. חשוב לציין שגם תקן מתעדכן, אין פירוש הדבר שהעדכן החדש מופיע מיד בדפדפנים שMRIים ג'אווהסקריפט או בשרתים שMRIים ג'אווהסקריפט, אלא לווח זמן עד שהעדכנים החדשניים ביוטר עושים את דרכם אל הדפדפנים/שרתים שколоנו משתמשים בהם. אם שמעתם מפתחי אינטרנט "מקטרים" על דפדפנים ישים – זו בדיקת הסיבה.

זה המנייע לכתיבת הספר. הבן שלי, ביום מתכנת בזכות עצמו, ניסה למדוד ג'אווהסקריפט מ於是 ולא הצליח למצוא למצוין ספרים בעברית. החומר שיש לכם בעברית בנוגע

לג'אוوهסקרייפט הוא דל ומיושן. חלק מחוברות העזר הנמצאות בบทי הספר מתיחסות לתקנים שהפסיקו להיות בשימוש בשנת 2007! התיחסות אמיתית לתקנים החדשים ביוטר של השפה, שיצאו בשנת 2017, אין בנמצא בעברית. התחלתי לנכון הסברים עבור בני, ומפה לשם הבנתי שאני חייב לחת את זה הלאה.

ג'אוوهסקרייפט היא שפה שקל ללימוד. בגיןוד לשפות אחרות, לא נדרש בה סביבת שרת מורכבת או כלי פיתוח שעולים נספ. לא נדרש ידע מקיף במדעי המחשב. כל ש;brיך הוא לפתח Notepad במחשב, לפתח דף-דף ולהתחליל ללמידה ולפתחה. צרייך גם הדרכה נconaה ומשמעות עצמית. אני מ庫ווה שבספר הזה תמצאו לפחות הדרכה נconaה. המשמעות העצמית – עלייכם. אני מאמין שככל שתתקרדו בספר תראו את פירות הלימודים, הניצוץ בעיניהם יתחזק ולא תצטרכו עוד משמעות עצמית – אתם פשוט תאהבו בג'אווהסקרייפט בכלל ובפיתוח לוב בפרט. אל תדלגו על הפרק שבו אני מסביר איך ללמידה; זהו הפרק החשוב ביותר בספר.

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

– רן בר-זיק

איך לומדים

לא למדתי מדעי המחשב באוניברסיטה או במכילה. למען האמת, עד גיל מאוחר מאוד לא למדתי תכנות בעזרת מדריך. רוב מה שאני יודע למדתי ללא הרכה, וממנה סיבות: הראשונה היא שכאשר עשיתי את צעדי הראשונים בתכנות, בשנת 1996, החומר שהוא זמין באינטרנט היה דל מאוד. על חומר בעברית לא היה מה לדבר, והחומר באנגלית סייק בדרך כלל רק את הדוקומנטציה. אני לא ממש מתגאה בכך; למידה בלבד ללא הרכה היא כמעט מידה מלאה לא יעילה. הרכה ממשעה לא רק מרצה מנוסה, אלא גם אתר אינטרנט שבו יש הסבר מקיים, פורום או קבוצה בראשות חברת זו או אחרת (לא רק פיסבוק), שיש בהם אנשים מנוסים שיכולים לסייע או להפנות לחומר עוזר. היא גם מקום שבו אפשר לתמגלו ולהתנסות. למרבה המזל, בימים אלו קיימים שלל חומרים, עוזרים וסייעים. גם הספר הזה הוא הרכה. לא תידרשו לצלול לתוך הדוקומנטציה של ECMAScript על מנת להבין את השפה, אבל גם בעזרת הספר תמצאו דרך טובה ללמידה.

כיוון שהוא הזמן למדתי ללא הרכה, גיבשתי כמה עקרונות למידה שהכנסתי בספר זהה. 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 שהוא בסגנון דומה, טפלו בכך. ודאו שגם יכולים להיכנס לאתר הלימודי شاملו את הספר ושהסיסמה שלכם תקינה ופעלת.

קראו את הפרקים לפי הסדר

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

קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות

קראו את הפרק פעמיים אחד קרייה שוטפת. לאחר מכן קראו אותו שוב. הפעם קחו את כל דוגמאות הקוד, העתיקו אותן לסייעת העבודה שלכם ושחקו בהן! נסו לשנות את הערכיהם, לגרום לשגיאות, לכתוב קוד דומה.

אני מאמין שקוד לומדים דרך הידים ולא רק דרך הראש. חשוב להבין את התיאוריה ואת הרעיון מה אחורי מה שמנסים לעשות, אך לא מימוש, הידע הזה יתנוון וייעלם, בדיק נמו בשפט דיבור. אם לא תשתמשו ותרגלו, גם הלימוד התיאורטי המעמיק ביותר לא יהיה שווה הרבה. הקרייה הראשונה נועדה ללימוד התיאוריה, הקרייה השנייה נועדה לתרגול. לימוד שפה הוא לא מרוץ! 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 מתיאשו מהר ואל תרצו אל הפתרון אלא שברו 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'אווהסקרייפט ולדיבון 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, בעברית.

אל תהססו לקרוא, למשל דיון על שיטות השפה, להשתתף בדיונים או לסייע למי שהוא שheidע שלו דל משלכם. אל תשחחו להבין את רוח הדברים בקובזה ולא להציג או להעיק. רוב המשתתפים בקובזה הם אנשים עובדים שלאו דווקא זמינים להודעות מיידיות.

הגיעו לسدנאות ולミיטאפים

לא מעט חברות מארגנות בחינם סדנאות, מיטאפים ומפגשים שבהם מתכנתים מציגים את ג'אוהסקרייפט ומרצים עליה. כדאי מאוד להגיע למפגשים האלו, לא רק על מנת לשמע את התכנים ולהשתתף בסדנאות אלא גם להכיר אנשים אחרים בתעשייה. עם חלكم אתם תתכתבו בקבוצות הפיסבוק לג'אוהסקרייפט. קהילת מפתחי הג'אוהסקרייפט בארץ היא קהילה מאוד שיתופית וקרובה, וربים בה מכירים זה את זה.

תרגול, תרגלו, תרגלו

הספר לא שווה הרבה הינה בלי תרגול מكيف ושימוש בו. אל תעברו לפרק הבא לפני שתסתינו את כל התרגולים שקשורים לפרק שקראתם. זה לא קריטי, אלא סופר-קריטי. באתר hebdevbook.com, המלווה ספר זה, ישנו אתר תרגולים המועד לקוראי הספר.

העזר ב-GPT וIALIZEDIM דומים

מומלץ להעזר בבינה מלאכותית בכל הקשור ללימודים. הבינה המלאכותית הנפוצה כיום היא GPT Chat וניתן להעזר בשירותה באמצעות כניסה אל האתר <https://chat.openai.com> ויצירת חשבון חדשני. ניתן להזין אל הצעת שאלות ובקשות שונות הקשורות לקוד. למשל: Explain about this error ואז את השגיאה המתבלת. ניתן גם לפנות לבינה המלאכותית בבקשת דוגמאות שונות והסבירים על נושאים בספר. למרות שרמת העברית של הבינה המלאכותית משתפרת, מומלץ לשאול את השאלות באנגלית.

אני ממליץ להמנע מהפייטוי להשתמש בבינה מלאכותית לבצע תרגולים עבורכם או לתוכנת עבורכם. מעבר לכך שתהlixir הלימוד לא יהיה אפקטיבי, בסופו של יום הבינה המלאכותית לא יכולה להחליף מתכנתים והכרות עם הקוד נדרש על מנת לפתח עליה או להנחות אותה באופן אופטימלי. אם לא תכירו היטב קוד, לא תוכלו לעבוד איתה ביעילות תוך יצירת קוד אינטואיטיבי מספיק לשימושים בעולם האמיתי.

התקנת סביבת העבודה ודרך הלימוד

ג'אוּהַסְקְּרִיפְט יכולת לזרוץ בסביבת שרת או דרך הדפדפן. איך זה בדוק עובד? ג'אוּהַסְקְּרִיפְט נמצאת בקובץ טקסט. כן, בדוק כמו זה שאפשר ליצור באמצעות כתוב הטקסט (Notepad) שיש בכל מערכת חלונות. בדפדפן מותקן כלי שלוקח את קובץ הטקסט זהה וMRIIZ אותו ואת הפקודות שנמצאות בתוכו. אם הדפדפן היה אדם, הוא היה פותח את קובץ הטקסט וקורא את מה שיש בתוכו, למשל: "לך ימינה ופתח את הדלת", ועשה בבדיקה מה שכותב. הפעולה זו נקראת בלשון הפופולרית "רינדור", מלשון render בלאז. הדפדפן לוקח את קובץ הג'אוּהַסְקְּרִיפְט וMRIIZ אותו. קובץ הג'אוּהַסְקְּרִיפְט יכול לעשות כל מיני דברים ולהציג או לא להציג אותם.

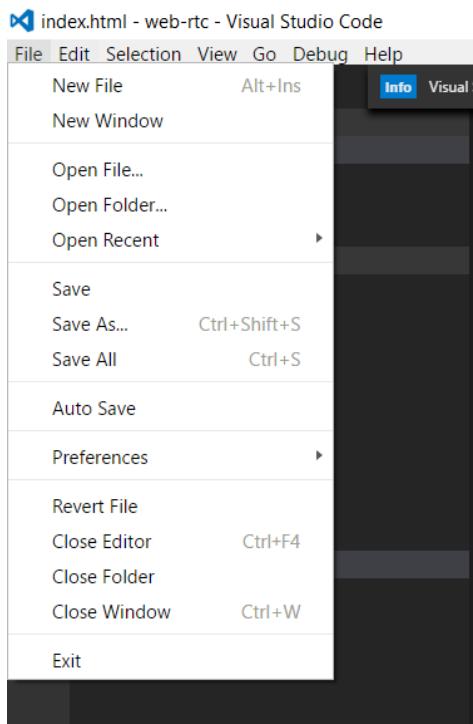
איך הדפדפן טוען את קובץ הג'אוּהַסְקְּרִיפְט? יש כמה דרכים לעשות זאת, אבל כרגע ארצה ללמד אתכם איך לכתוב קובץ ג'אוּהַסְקְּרִיפְט ולראות אותו פועל על מנת להבין את כללי השפה ולכתוב משהו באופן ראשוןינו ביותר. החלק החשוב והקשה ביותר הוא יצירת סביבת העבודה, ככלומר סיבה ממוחשבת שבה אפשר להקליד ג'אוּהַסְקְּרִיפְט ולראות אותה עובדת. סיבה זו היא חשובה מאוד כאשר לומדים, כיון שלימוד של שפת תכנות נעשה ראשית כל "דרך הידים" וחשוב מאד לא רק לקרוא אלא גם לתרגל. וכייל תרגל צוין סיבה שמאפשרת להקליד פקודות שפה, לשמרם ולראות את הפלט. אני שבעודגייש: התקנת הסביבה היא החלק הקשה ביותר בתחום לימוד שפה חדשה וגם החשוב ביותר. לפיכך כדאי להיאוזר בסבלנות, לקחת נשימה ארוכה ולזכור שדווקא עכשו מתמודדים עם החלק הקשה ביותר.

הבה נתחיל בעורך טקסט טוב. אמנם אפשר להשתמש בנוטפץ, אבל הוא לא מציע צביעת קוד לצורך עזרה בקריאות ולא יוצרת הזחות בקלות. יש כמה עורכי טקסט המותאמים במיוחד לכתיבת ג'אוּהַסְקְּרִיפְט, שאצין כמה מהם. בחרו בעורך טקסט אחד! רובם זהים במידה ומכילים יכולת עריכה בסיסית של HTML, CSS וג'אוּהַסְקְּרִיפְט.

שימוש לב: מתכוונים מנוסים לא משתמשים בעורך הטקסט פשוט אלא בעורך טקסט משוככל יותר שנקרא IDE או Integrated Development Environment – סביבת פיתוח משולבת. הסיבה זו מאפשרת השלמת קוד, מצינית שגיאות בכתיבה ובוגם יכולה להרייז את הקוד עצמו.

IDE מעולה הוא Visual Studio Code. הוא חינמי, מבוסס קוד פתוח, כתוב בג'אווהסקריפט (כן, כן), נתמך על ידי מיקרוסופט וניתן להורדה פה: <https://code.visualstudio.com/>.

אחרי ההתקנה תוכל לפתח את התוכנה, לבחור ב-File ואז לפתח את התקינה שבחרתם ולפתח או ליצור בה קבצים. אני ממליץ לכם בחום להוריד את עורך הקוד זהה: הוא חינמי לחלווטין, איןנו מכבייד על המחשב וכן נפוץ בתעשייה היום.



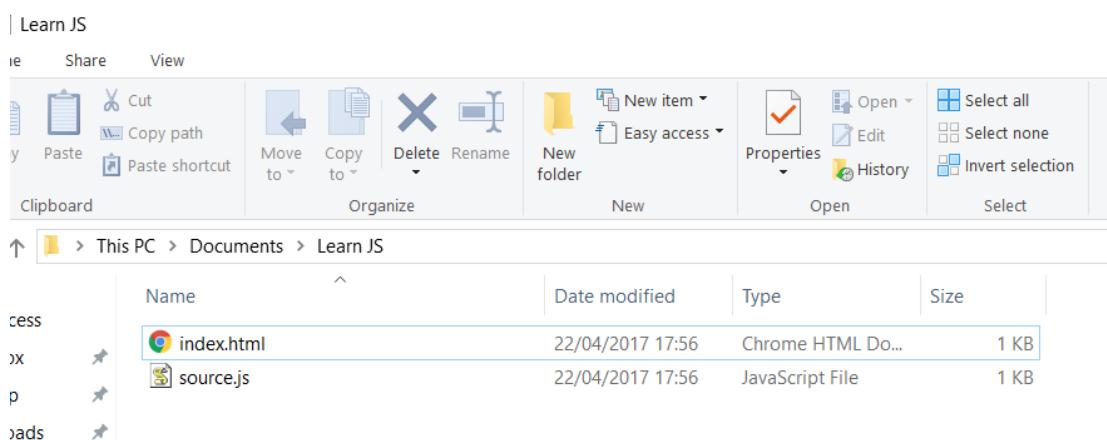
IDE מוצלח אחר הוא Atom. גם הוא חינמי ובסיסי קוד פתוח וגם הוא... כתוב בג'אווהסקריפט. הוא נתמך על ידי גיטהאב וניתן להורדה פה: <https://atom.io/>. הוא דומה ל-Visual Studio Code ואחריו ההורדה וההתקנה שלו אפשר להפעיל אותו בקלות. עורך טקסט נוסף שנחשב לאמין וטוב הוא תוכנה חינמית בקוד פתוח שנקראת <https://notepad-plus-.plus.org/download>. הוא ניתן להורדה בקישור הבא: <https://notepad-plus-.plus.org/download>.

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

הסיבה הטובה ביותר ללמידה היא יצירת קובץ HTML שטוען קובץ ג'אווהסקריפט. קובץ HTML הוא קובץ שהדף יודע לפרש ולהציג, והוא יכול לקרוא לקובץ ג'אווהסקריפט באופן צהה שהדף ירנדר אותו. כאמור, רינדר הוא הרצת הפקודות שנכתבות בקוד

ג'אוوهסקריפט והציגן על המסך או במקום אחר. רינדור, מילשון render, הוא מונח מתחום מדעי המחשב ופירושו הוא "הרצאה". כאשרпи כתוב "הקוד מרונדר" הפירוש הוא שהקוד רץ ומציג את התוצאות. יש ליצור במחשב תיוקיה – זה יכול להיות ב"המסמכים שלי" או על שולחן העבודה – ובתוכה ליצור קובץ בשם source.js וקובץ בשם index.html.

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



ודאו שמערכת החלונות או המქ שלחכם תומכת בתצוגת שם הקובץ המלא, כולל הסיומת .index.html.txt. אחרת, הקובץ index.html יהיה בעצם (Extension) של הקובץ.

בקובץ ה-HTML כתבו את הקוד הבא:

```
<!doctype html>
<html>

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

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

</html>
```

בתוך קובץ `source.js` כתבו את הטקסט הבא:

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

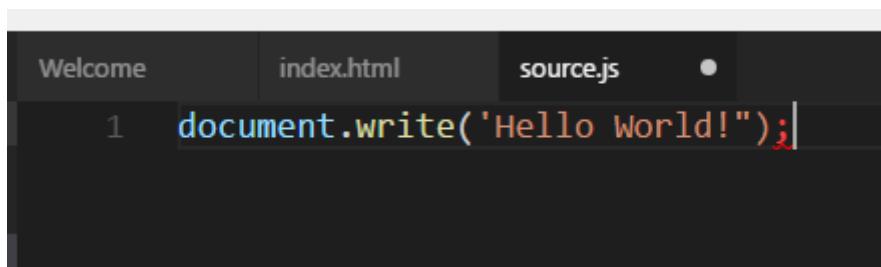
אחרי ששמרתם את תוכן שני הקבצים, פתחו את הקובץ `index.html` בדפדפן כרום או בפיירפוקס (לא באdeg'). אם הכל תקין, תראו שכותוב על המסך "Hello World!" – כתבתם את הג'אווהסקריפט הראשון שלכם!

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

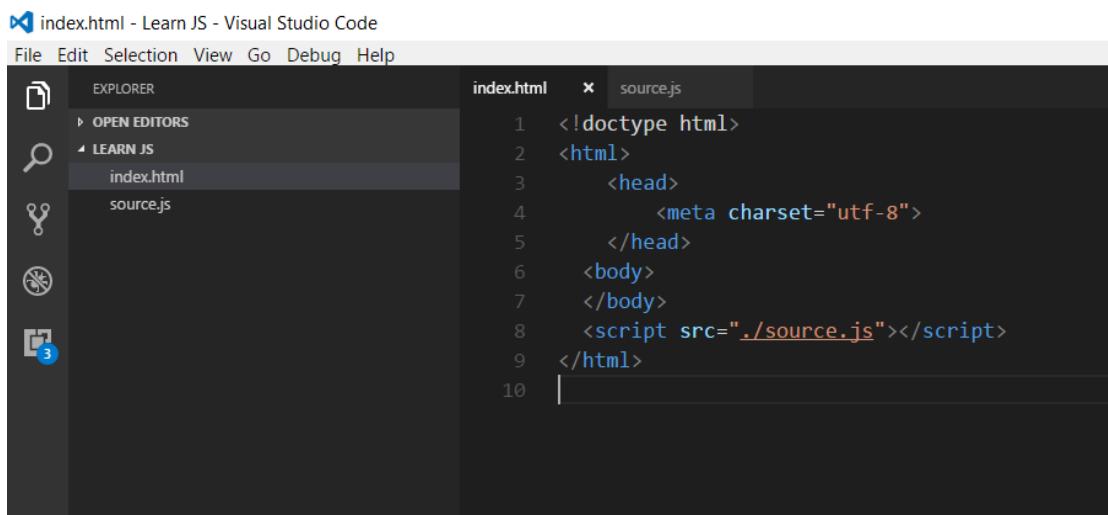
בדקו שאכן קראתם לקבצים `source.js` ו-`index.html`. הבדיקה צריכה להתבצע באמצעות הכפתור הימני של העכבר בחלונות, כדי למןעו מצב שבו מערכת הפעלה הוסיפה תוספות לשמות ו-`index.html` נשמר בשם `index.html.txt`.

בדקו שאתם פותחים את הקבצים בדפדפן כרום או בפיירפוקס. בדקו שאין תוספים מיוחדים לדפדיניים שעולמים לחסום את הרצת הקובץ על ידי הרצה של מצב פרטיות.

בדקו שהקלידתם את הטקסט כשורה בקובץ `source.js` ללא רווחים או תווים מיוחדים. נסו להשתמש ב-`Atom` או ב-`Visual Studio Code`. שימו לב שאין התראות על שגיאות הקלדה. כאן למשל מובאת דוגמה של שגיאת הקלדה שביצעת. אם קיבלתם התראה כזו, בדקו שוב שלא טעתם בגרש ושלא הכנסתם רווחים מיותרים כמו בדוגמה זו שבה יש שימוש שגוי בגרש יחיד ואז בגרשיים. עדיף להשתמש תמיד בגרש יחיד:



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



```

index.html - Learn JS - Visual Studio Code
File Edit Selection View Go Debug Help
EXPLORER
OPEN EDITORS
LEARN JS
index.html
source.js
index.html x source.js
1 <!doctype html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5   </head>
6   <body>
7   </body>
8   <script src=". /source.js"></script>
9 </html>
10

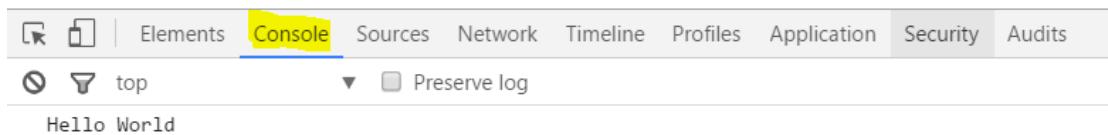
```

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

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

כיוון שאתם כבר מפתחי ג'אווהסקריפט מנוסעים, כדאי שתתמודו להשתמש בקונסולה של הדף. מדובר במכשיר מיוחד שמאפשר "לדבג" את ג'אווהסקריפט. הפעול "לדבג" כוונתו להסתכל על צפונות הרינדור ולראות ממש את הפלט של השפה או את תוצאות הרצה שלה. ככלומר מה שכתבנו. נשמע מסובך? יש להבין איך הדף מנסה להריץ את הקוד שלו (כאמור מה שנקרא "רינדור", מלשון הריצה, באנגלית) ולקבל מידע נוסף במקרה שהוא נכשל, ככלומר "זורק" שגיאה צבועה באדום בקונסולה ואף מפנה לשורה הביעיתית.

על מנת לראות את הקונסולה, פתחו את כל הمبرחים של הדף. בכרום ובפיירפוקס לחזו + Ctrl + Shift + F אם יש לכם חלונות או + Cmd + Option + F אם יש לכם מק. אפשר לעשות את זה גם דרך התפריט העליון בשני הדפים. לאחר פתיחת כל המברחים, לחציהם על לשונית **Console**.

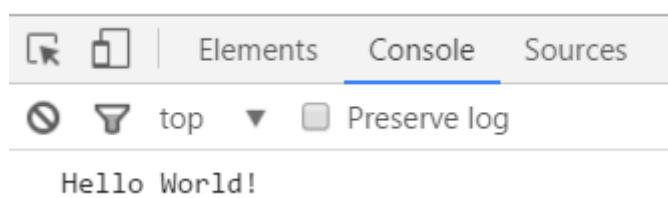


הבה נבדוק מה אפשר לעשות בעזרת הקונסולה. היכנסו אל `source.js` והחליפו את הטקסט ל:

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

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

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



חשוב: אל תלגגו על השלב הזה. בכל שלבי הלימוד כדאי להשתמש בקונסולה, שהיא הרבה יותר נוחה להציג. אם משהו 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` או `Cmd + R` את הדף?

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

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

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

תרגיל:

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

פתרון:

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

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

שמרו את הקובץ, פתחו את הקובץ index.html, שטוען את הקובץ source.js, ורעננו את הדף. לחזו על נספחים Ctrl + Shift + F5, וראו את התוצאות.

תרגיל:

גרמו לקובץ ה-HTML לטען קובץ ג'אוوهסקריפט בשם targil.js ושמדפיס בקונסולה: I .am new file

פתרון:

צרו קובץ targil.js באוטה תקינה של הקובץ index.html. יש לוודא שהוא שמו האמתי של הקובץ ושםערכות הפעלה לא מźמינה לקובץ עם סימנת txt. את זה עושים על ידי בדיקה בהגדרות התצוגה של מערכת הפעלה. פתחו את קובץ ה-HTML באמצעות עורך טקסט (מומלץ להשתמש ב-Visual Studio Code) ושנו את שם הקובץ הג'אווהסקריפט ל-.targil.js

```
<!doctype html>
<html>

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

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

</html>
```

בקובץ targil.js כתבו את השורה זו:

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

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

פרק 1

משתנים



משתנים

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

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

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

לאחר מכן מכניסים ערך למשתנה. הפעולה זו נקראת "צבת ערך" או "השמה", והוא נקראת כך מכיוון שהערך מוצב בתוך המשתנה. במקרה זה מדובר בטקסט הכלול את המילים `Hello World`.

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

אפשר לקצר ולהציב את הערך מיד בהגדרת המשתנה:

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

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

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

אם תציצו בקונסולה, תראו שמודפס המשפט "Hello World". מודיע? כיוון שהוא מה שיש בתחום המשתנה. מן הסתם, משתנה ניתן לשינוי. נסו את הקוד הזה:

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

מה לפי דעתכם יוצג בקונסולה? יוצג "I am a new version" או "Hello World"? כיוון שהערך הקודם "נדرس" על ידי הערך החדש. הכנסתם (כלומר הצבתם) ערך חדש לתוך המשתנה, ועכשו מה שיש בתוכו השתנה. כשמציגים את המשתנה רואים את הערך החדש.
יש הבדל מהותי בין הגדרת משתנה לבין הכנסת ערך לתוכו. הגדרת משתנה היא כמו בנייה ארון או קופסה ואפשר בכל פעם להכניס לתוכו ערך אחר.

שימוש לב: אחרי שימוש מסוים 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 מחיבבים. אפשר להשתמש בכל אות שהיא ובסימנים _ או \$ בתחילת השם, ובכל האותיות והמספרים ובסימנים _ או \$ בהמשך השם.

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

שימוש לב: אפשר להשתמש בעברית בהגדרת המשתנים, אבל מומלץ שלא לעשות את זה גם כיון שהקוד שלכם יהיה פחות קריא וגם כיון שהוא עשוי להוביל להתנהגות מזו娘ת ולצרות.

אתן לכם טיפ: בעולם התכונות אל תחפשו צורות כי יש מספיק מהן גם כן.

שימוש לב 2: בגרסאות קודמות של ג'אווהסקריפט השתמשו במילה השמורה `var` להגדרת משתנה. במקרים החדשים כבר לא מקובל להשתמש ב-`var` כיון שלו שימוש בו יש השלכות שנדונן בהן בהמשך הספר. הוא נשאר איתנו בעיקר בשל יכולתו לאחור עברו קוד שנכתב בשנים עברו.

תרגיל:

צרו משתנה בשם המרכיב לפחות שתי מילים, הדפיסו אותו בקונסולה וודאו שבהדפסה 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 `=`. שימוש לב שהtekst מוקף בגרשיים. פקודה `console.log` שלמדנו בפרק הקוד משמש להדפסת הטקסט - ובמקרה זה המשתנה שהוא מקבלת.

תרגיל:

צרו משתנה בשם myVar והכניסו לתוכו את טקסט "me not know JavaScript". דרכו
console.log ידפיסו אותו באמצעות log JavaScript" know I ו- "הזה בטקסט

פתרון:

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

הסבר:

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

תרגיל:

צרו משתנה בשם myVar והכניסו לתוכו את הטקסט "I am myVar". צרו משתנה נוסף בשם myVar2 והכניסו לתוכו את הטקסט "I am myVar 2".
הדפיסו את שניהם באמצעות log.console (בונוס: הדפיסו את שניהם ב קונסולה באמצעות פקודת console.log אחת)

פתרון:

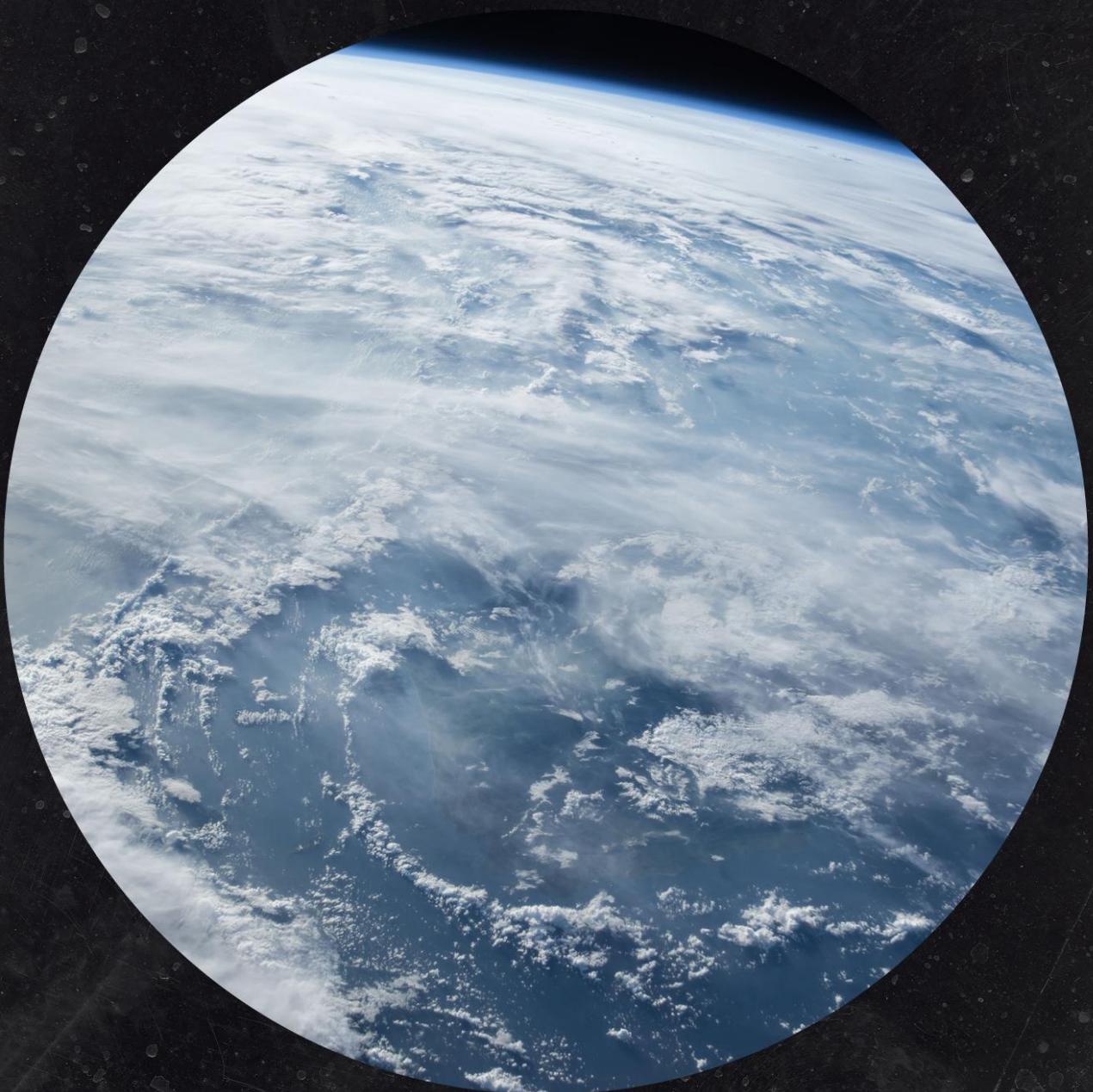
```
let myVar = 'I am myVar';
let myVar2 = 'I am myVar 2';
console.log(myVar);
console.log(myVar2);
```

הסבר:

אין מניעה להגדר כמה משתנים באותו סקריפט. הגדרתי את המשתנה באמצעות המילה השמורה let, שם המשתנה (הכולל מספר בסוף) והדפסה שלו ל-log.console. הסבר לתרגיל הבonus: נסו לחפש באינטרנט איך להדפיס ל-console שני משתנים. הרבה פעמים יהיו דברים שלא תבינו בתרגול. במקרים קראוא שוב ושוב את ההסבר, נסו לחפש תשובה בראשת. כך עושים מתכנתים מקצועיים: הם מחפשים תשיבות בראשת.

פרק 2

טראנס



טקסט

טקסט (באנגלית `text string`) הוא סוג מידע חשוב מאוד ובסיסי מאוד במערכות מידע. משתמשים בטקסט בכל מקום ובכל מערכת, ומדובר במקרה חשוב מאוד בלמידת קוד – יותר ממתמטיקה ופעולות חשבוניות. בעבר היה קשה לעבוד עם טקסט שאינו אנגלית. אם אתם מבוגרים מספיק אתכם ודאי זוכרים כל מיני אותיות מעוותות שהופיעו באתרים או בתוכנות. היום קל לעבוד במגוון שפות בתוכנות, אתרים וcmbובן בג'אוوهסקריפט, בזכות תקן חדש שנקרא **"יוניקוד"**. יוניקוד מאפשר כתוב בעברית, ביינית, ברוסית ובכל מערכת כתוב אחרת בקלות ובלוי חשש להופעת גיבריש והוא מותםע במחשב שלכם באופן אוטומטי.

בפרק הקודם למדנו איך ליצור משתנים ולהכניס לתוכם טקסט. בפרק זהה תרchieבו את המידע שלכם שלהם בנושא הטקסט. כאמור, אפשר להכניס טקסט למשתנים בג'אוوهסקריפט באופן הבא:

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

הtekst מוקף בגרשי בודד, אבל אפשר להשתמש גם בגרשיים כפולים:

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

הבחירה ביניהם היא בדרך כלל עניין של טעם. רוב המתכנתים נהגים נcone להיום להשתמש בגרש בודד להגדרת טקסט או בגרש מסוג ` (backtick), שעליו נלמד מאוחר יותר. ג'אווהסקריפט מאפשרת לחבר בין מחרוזות טקסט ידי בקלות בעזרת הסימן + (כמו בפעולות חיבור):

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

מה שודפס על הקונסולה הוא **foobar**. אף על פי שהסימן ה+ מוכך לחבר מספרים, בג'אווהסקריפט אפשר כאמור להשתמש בו לחיבור מחרוזות.

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

שימוש לב: השימוש ב-`foo` וב-`bar` הוא מאד נפוץ בדוגמאות בשפות תכנות ונדיר למצוא מדריך לשפת תכנות שלא משתמש במילים האלה ובמילה `baz` למשתנים (או לחלקים אחרים בקוד שעוד תלמדו עליהם). לדוגמה האלו יש שם מפחיד: משתנים מטה-סינטקטיים – אבל לא צריך לפקח משמות מסווגים.

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

לדוגמה:

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

יחזיר שגיאה בקונסולה. כיון שככל מחuzeות טקסט חייבת להיות מוקפת בגרשיים, המנווע של ג'אוوهסקריפט חושב שככל מה שמגיע אחרי הגרש הוא משתנה ולא יודע מה לעשות איתו. בדוגמה הבאה, החלק המודגם הוא מחuzeות הטקסט, והחלק שאינו מודגם הוא מה שהמנוע של ג'אוوهסקריפט לא יודע מה לעשות אליו:

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

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

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

ההדפסה תציג את הטקסט כמו שצרכיך, עם הגרש. הפעולה זו נקראת **escaping** ובה לוקחים טקסט שהוא שובר את שפת התוכנות והופכים אותו לבטוח. לחולופין, אפשר ליצור אליו תווים שלא ניתן לכתוב במקלדת, למשל ירידת שורה. איך כתבים ירידת שורה? **בעזרת \n.** העתיקו את הדוגמה זו:

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

ותראו מה קורה. בקונסולה יופיעו שלוש שורות.

ויתכן שתיתקלו ב-escaping במקומות נוספים. אם רוצים לכתוב רק א, צריך לעשות לו escaping ולכתוב אא.

כעת, כשאתם מרגישים בנוח בכל מה שקשרו לטקסט או, נcone יותר, למחרוזת טקסט בג'אווהסקריפט, הבה נסבך מעט את העניינים. בג'אווהסקריפט מחרוזת טקסט נחשבת מסווג מידע. באנגלית זה נקרא **Data Type**. יש כמה סוגים מיידע, כמו מספריים למשל, אבל סוג המידע של מחרוזת טקסט נקרא **string** והוא עומד ברשות עצמו. כאשר יוצרים משתנה ומכניסים לתוכו טקסט, בעצם אומרם שהמשתנה הוא מסווג מחרוזת טקסט. ברגע המשתנה מסווג מקבל לעצמו מידע והופך למשתנה מסווג מסוים, הוא מקבל גם סט של יכולות מיוחדות, ממש כמו קלאר קנט שהופך לוסףמן. מהשניה שנכנסה למשתנה טקסט, אפשר לעשות בו כמה פעולות שאפשרות אך ורק לסוג המידע של הטקסט.

כך למשל אפשר לבצע על המשתנה המכיל טקסט את הפעולה שתගרום לכל האותיות שלו להיות ראיות (כלומר קפיטלים). לדוגמה:

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

יוצרים משתנה עם טקסט. על המשתנה זהה עושים פעולה שאפשרית אך ורק עם משתנה מסווג טקסט. שם הפעולה הוא **toUpperCase()**. התוצאות של הפעולה הזה מושמות למשתנה נוסף בשם **myVarUpper** ומודפסות. אם תעתייקו את הטקסט הזה ותריצו בקונסולה, תראו שההתוצאה היא 'HELLO WORLD'.

השימוש בהפעלת פעולות על משתנה עם **Data Type** מסוים אינו בלבד רק למחרוזות, ובהמוך הספור תראו שאפשר לבצע פעולות גם על Data Types אחרים. בין היתר צריך להבין שיש סט רחב מאוד של פעולות שאפשר להפעיל על מחרוזות, **toUpperCase**, ו- **toLowerCase** רק אופציה אחת. על מנת להפעיל פעולות שונות משתמשים ב-. (נקודה) על המשתנה. אם משתמשים ב-IDE (עורק טקסט) כמו VS Code, כאשר תרשמו ". " אחרי המשתנה, עורק הקוד יראה לכם את שילל הפעולות האפשרות על המשתנה זה.

שימוש לב שהפעולה **toUpperCase** לא משנה את המשתנה עצמה! שינוי המשתנה עצמו נקרא "מו太子ה", וזה לא מה שעושים פה. כדי לקבל את הערך שהשתנה צריך להגיד משתנה נוסף ולהכניס את הערך למשתנה. מבלבל? הנה נציגים זאת שוב בפעולת נוספת. הפעם צריכים המשתנה והכניסו לתוכו טקסט. המשתנה הראשון, מהרגע שיש בתוכו טקסט, יוכל לבצע פעולה של טקסט ולהחזיר את התוצאה אל המשתנה אחר שאותו תדפיסו:

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

יש פעולות נוספות שאפשר לעשות על מחרוזות טקסט, אבל כרגע התמקדו בשתי הפעולות האלו גם בתרגול.

תרגיל:

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

פתרונות:

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

הסבר:

যօצרים ומצביעים ערכיהם במשתנים בשם myVar1 ו-myVar2. יש להקפיד שמחרוזות הטקסט יהיו מוקפות בגרשיים בודדים. מגדריהם המשתנה השלישי בשם answer ושמותיו בתוכו את הסכום של myVar1 ו-myVar2. ההדפסה נעשית כרגיל. שימוש לב שרווח גם נחשב לאות. אם לא מכניסים רווח באחת מחרוזות הטקסט בתוך המשתנים, לא יהיה רווח.

תרגיל:

צרו משתנה שיכיל את מחרוזת הטקסט:
."The student's and the teacher's motivations were in conflict"

פתרונות:

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

הסבר:

שים לב לשימוש ב-**escaping!** מה זה escaping? פשוט שימוש בסימן \ על מנת להודיע על מנוע של ג'אוּהַסְקְּרִיפְט שהגרש הוא חלק ממחרוזת הטקסט. ההשמה וההדפסה של המשתנה נעשות כרגיל.

תרגיל:

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

פתרונות:

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

הסבר:

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

פרק 3

מספרים



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, כיוון שהוא מספר. מספרים טהורים יכולים להכנס ללא גרשים. להזכירם, מחוץ לטקסט, שעליה למדנו קודם, מוקפת בגרש או בגרשיים. מספרים לא מוקפים בגרש או בגרשיים.

אם תדפיסו את המספר כמו בדוגמה, תראו שירודפס 12. כדי העין שביניכם ישימו לב שההדפסה בקורסולה נראית מעט שונה מאשר במספר ולא בחרוזת טקסט. ההבדל נוצר בגלל השוני בין סוג המידע של מחרוזת טקסט לסוג המידע של מספר.

אם מחברים בין מספרים אזי החיבור ייעשה בבדיקה כפי שהייתה מצפיהם:

```
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 לחברו, אפשר לעשות פעולה חישור בעזרת הסימן "-". למשל:

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

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

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

כאן התוצאה תהיה `-22`.-10- ו עוד `12`.

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

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

התוצאה כאן תהיה `16`. 2 כפול `8`.

אפשר גם לבצע חילוק בעזרת הסימן `/`.

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

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

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

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

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

בדומה לפעולות המוחדרות של מחרוזת טקסט שראיתם שאפשר להפעיל על משתנים מסווג מחרוזת טקסט, גם למשתנים מסווג מספר יש פעולות מיוחדות. למשל, פועלה המגבילה את המספר לאחר הנקודה העשרונית. אם תחלקו 10 ב-3, תקבלו 3.3333333 עד שלא ידע. אפשר להיעף את הנקודות העשרוניות. איך? באמצעות פועלה מיוחדת בשם :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 זהה.

בשורה הראשונה יוצרים משתנה בשם foo ומכניסים לתוכו את המספר 10.

בשורה השנייה יוצרים משתנה בשם bar ומכניסים לתוכו את המספר 3.

בשורה השלישית יוצרים משתנה בשם answer. מה הוא מכיל? foo חלקי 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 ומכניסים את התוצאה למשתנה finalAnswer. הפונקציה round מעגלת את המספר. למה? ל-3. אם תדפiso את finalAnswer תקבלו 3.

הינה טבלה קצרה שמסכמת את כל הפעולות הבסיסיות שאפשר לעשות:

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

מציאת השארית

אם אתם זוכרים מתקופת בית הספר היסודי, יש דבר זהה שנקרא שארית. כמה זה שיש חלקו ארבע? אחת ושארית שתיים. כמובן, החלק מהמספר השלם שהוא קטן מהחלק. תשע חלקו שווה זה אחת עם שארית אחת. תאמינו או לא, בתכנות יש עדנה לדברים שלומדים בכיתה ד'. אפשר לחלק ולמצוא את השארית בעזרת סימן מיוחד – %. הסימן הזה נקרא על ידי המתכנתים "מודולוס" (באנגלית modulus).

```
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 המספרי 4. שואלים מה השארית של 6 חלקו 4. התשובה כאן היא המספר 2.

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

אפשר להציג את תוכנות האופרטורים ישירות בתוך המשתנה. למשל, אם רוצים להציג בתוך משתנה את התוצאה של 4 כפול 2, אין צורך ליצור משתנה שכיל 4, משתנה שכיל 2 ומשתנה שכיל את המכפלה של שניהם, כמו שצוין:

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

במקרה, אפשר לנתח מושהו זה:

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

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

אופרטורים מקוצרים

נניח שיש משתנה שרצים להוסיף לערך 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

מןו 1. ניזון שהוזרים על התוצאה שלוש פעמים, המשתנה שווה 7-
הבעיה היא שזה מכוער... יש גם אופרטור קיצור שיכול להוסיף או להוציא 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
מןו 3. התשובה היא 7.

הינה טבלה חלקיות של האופרטורים המקבזרים הנפוצים ביותר:

| משמעות | אופרטור מקוצר |
|------------------------------|-----------------------|
| <code>foo = foo + 1;</code> | <code>foo++</code> |
| <code>foo = foo - 1;</code> | <code>foo--</code> |
| <code>foo = foo + 10;</code> | <code>foo+=10;</code> |
| <code>foo = foo - 10;</code> | <code>foo-=10;</code> |

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

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

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

זה נראה שונה לעין בלתי מיומנת, אבל ג'אווהסקריפט היא שפה שישği המשתנים בה הם **implicit** – כלומר המנוע של השפה מנשה לנחש אותם ולא ממהר לזרוק שגיאה כמו בשפות אחרות. כאמור, משתמש לא מיזמן עלול להתבלבל, ולמתקנתים בשפות אחרות זה נראה כאוטי, אבל ברגע שמתרגלים – מדובר בתוכנה נואה מאוד.

תרגיל:

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

פתרון:

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

הסבר:

יוצרים שלושה משתנים ושמיהם בתוכם מספרים באופן קצר כפי שלמדונו. שלושת המספרים מחוברים והתוצאה נכנסת למשנה `answer`, שבתורו מודפס.

תרגיל:

צרו משתנה, הציבו בתוכו את המספר 12 והדפיסו אותו. באותו משתנה הציבו מחרוזת 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 מחרוזת טקסט משתמשים בגרשיים. ההדפסה נעשית באותה אופן, אך כאמור, ברוב הקונסולות תראו הבדל בין שתי ההדפסות שנועד לסמן שהדפסה אחת היא מספר והאחרת היא טקסט.

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

תרגיל:

צרו משתנה והציבו בתוכו את המספר 100. הדפiso 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 מתייחס בעצמי על זה שהכנסתי תחכום במתמטיקה
לתרגיל כאן.

תרגיל:

צרו משתנה שיכיל את השאריות של 10 חלקו 3.

פתרון:

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

הסבר:

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

תרגיל:

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

פתרונות:

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

או

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

הסבר:

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

פרק 4

סוגי מדע פרימיטיביים נוספים



סוגי מידע פרימיטיביים נוספים

בפרקים הקודמים למדנו על סוג מחורזת טקסט ומספר. סוג מידע אלו נקראים "סוגי מידע פרימיטיביים". כשאומרים שסוג מידע הוא פרימיטיבי לא מתחכמת בכך שהוא לא אוכל בסכין ובמצלג אלא שהוא ייחידת מידע בסיסית ולא מתוחכמת שבאה עם השפה. חוץ מספר וממחזרות טקסט יש עוד כמה סוגים מידע כאלה.

بولיאני

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

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

שים לב שה-`true`, בדיק כמו מספר, אינו מוקף בגרשיים. אם הוא היה מוקף בגרשיים הוא היה מידע מסוג מחורזת טקסט. פוטולה מיוחדת שיש לסוג מידע בוליאני היא המרת מחזרות טקסט באמצעות הפעולה `:toString()`:

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

מה קורה פה? קודם כל יוצרים משתנה ומכניסים לתוכו את הערך `true`. הערך הוא ערך בוליאני. ה-`true` מגיע ללא גרשימים כיון שהוא מילה שמורה. אם תדפיסו אותו, תראו שהקונסולה מראה אותו כ-`true`. מהרגע שהמשתנה זהה קיבל ערך בוליאני, הוא מסוג מידע בוליאני, ועל מידע בוליאני אפשר להשתמש בפעולות מיוחדות. אחת מהן היא `(toString()`, שגורמת לערך הבוליאני להפוך למחזרת טקסט. השורה השלישית מכניסה את הערך של המשתנה הבוליאני אל המשתנה `bar` ואז מדפיסים אותו. כדי העין יבחינו שיש הבדל בקונסולה בין ההדפסה הראשונה לשניה. הסיבה היא שבהדפסה הראשונה המשתנה הוא בוליאני ובהדפסה השנייה הוא מחזרות טקסט. בהמשך הספר אשוב למשתנים בוליאניים ואדון במצבים שבהם כדאי להשתמש בהם.

משתנה לא מוגדר

"**לא מוגדר**" (`undefined`) הוא הערך הראשוני של כל משתנה עוזר לפני ההגדירה 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`.

ריק

סוג מידע פרימיטיבי נוסף הוא "**ריק**" (`null`). ככלומר, המשתנה לא מכיל דבר. שימוש לב שלא מדובר ב-0 ולא במחרוזת ריקה, אלא בכללם. ריק. ההגדירה נעשית באמצעות המילה השמורה `null`:

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

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

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

שימוש לב: ההבדל בין `undefined` ל-`null` לא נראה כרגע שימושתי אך הוא ממשמעוני מאוד. **undefined** משמעותו משתנה שהוגדר אך לא אוחTEL בערך כלשהו. `null` משמעותו משתנה שהוגדר ואוחTEL כ-`null`. כדאי לזכור שמדובר בשני סוגי מידע שונים לחלווטין. המידע הזה יהיה חשוב בהמשך הדרכך. כדאי לציין שיש כאן שוחשבים שמדובר במקרה בעיצוב השפה ושה-`null` לא אמור להתקיים.

Symbol

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

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

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

שימוש לב: `Symbol` הוא סוג מידע שנכנס רק בגרסאות האחרונות של ג'אווהסקריפט ושימושו כאשר רוצים ליצור מזהה ייחודי. חשוב להכיר אותו.

מציאת הסוג של המשתנה

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

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

בקוד שלעיל נוצר משתנה בשם `foo` והוצבה בו מחרוזת הטקסט "Hello World". מהרגע הזה, `typeof 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 ועדייף להשתמש בו אך ורק לסוגי מידע פרימיטיביים.

תרגיל:

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

פתרון:

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

הסבר:

המשנה הבוליאני `foo` נוצר עם הערך `false` ללא גרשימים. אם היוitem 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 הזה נכנסת למשנה אחר בשם `bar`, שהוא כמובן כבר מחרוזת טקסט לכל דבר ועניין. בהדפסה השנייה הוא מודפס כמחרוזת טקסט.

תרגיל:

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

פתרון:

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

הסבר:

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

תרגיל:

צרו משתנה מסוג `undefined` והמיירו 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 בנטח באמצעות המילה השמורה `null` הוא הופך למוגדר אך הוא חסר ערך. שימוש לבמהילה `null` היא מילה שומרה שאינה מוקפת בגרשיים. אם היא הייתה מוקפת בגרשיים היינו מקבלים מחזוזת טקסט שערכה הוא `null`.

תרגיל:

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

פתרון:

```
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` ובזדוקים אותו באמצעות `typeof` שערכו נכנס ל-`bar`. אפשר לראות שבהתחלת `foo` הוא `undefined`. כמשמעותם לתוכו מספר, סוג המידע בו הוא `number` וכמשמעותם לתוכו מחרוזת טקסט, שונות מהמספר בכך שיש סיבוב המספר גרשיהם, הוא סוג `string`. כמשמעותם ערך בוליאני, שהוא המילה השמורה `true`, הסוג הוא `boolean`, וכמשמעותם `Symbol` הוא הופך לסוג `symbol`. לבסוף מכניסים את הערך `null`. ה-`typeof` שלו הוא `object` מסיבות היסטוריות שלא נכנס אליו פה.

הערות

ברוב המקרים רוצים לכתוב הערות בקוד, טקסט שיסביר כל מיני דברים על הקוד בלי שירונדר. למה צריך את זה? לעיתים 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 התוצאה תהיה undefined, כי ההגדירה של a נמצאת בהערה. מקובל בהערות לשמר על רוח בין שני תווים // להערה עצמה, אך אין חובה לעשות זאת.

לעתים רוצים לכתוב הערה בכמה שורות. לפיכך משתמשים בתווים /* */ ו-/*/. בתחילת הגוף ההערה כתבים */ ובסוף /*:

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

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

בהמשך אשתמש בהערות על מנת להציג את הערכים השונים. כך למשל אם ארצה 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 10. באמצעות האופרטור המקיים ++ כבר למדנו עליו, נוסף 1 למשתנה זהה. על מנת לציין את הערך החדש נוספה הערה.

תרגיל:

בדומה לתרגיל הקודם, צרו משתנה ובאמצעות **typeof** שנו את סוג המידע שלוSSH
פעמים לשישה סוגי מידע שונים. כתבו בהערות את סוג המידע

פתרונות:

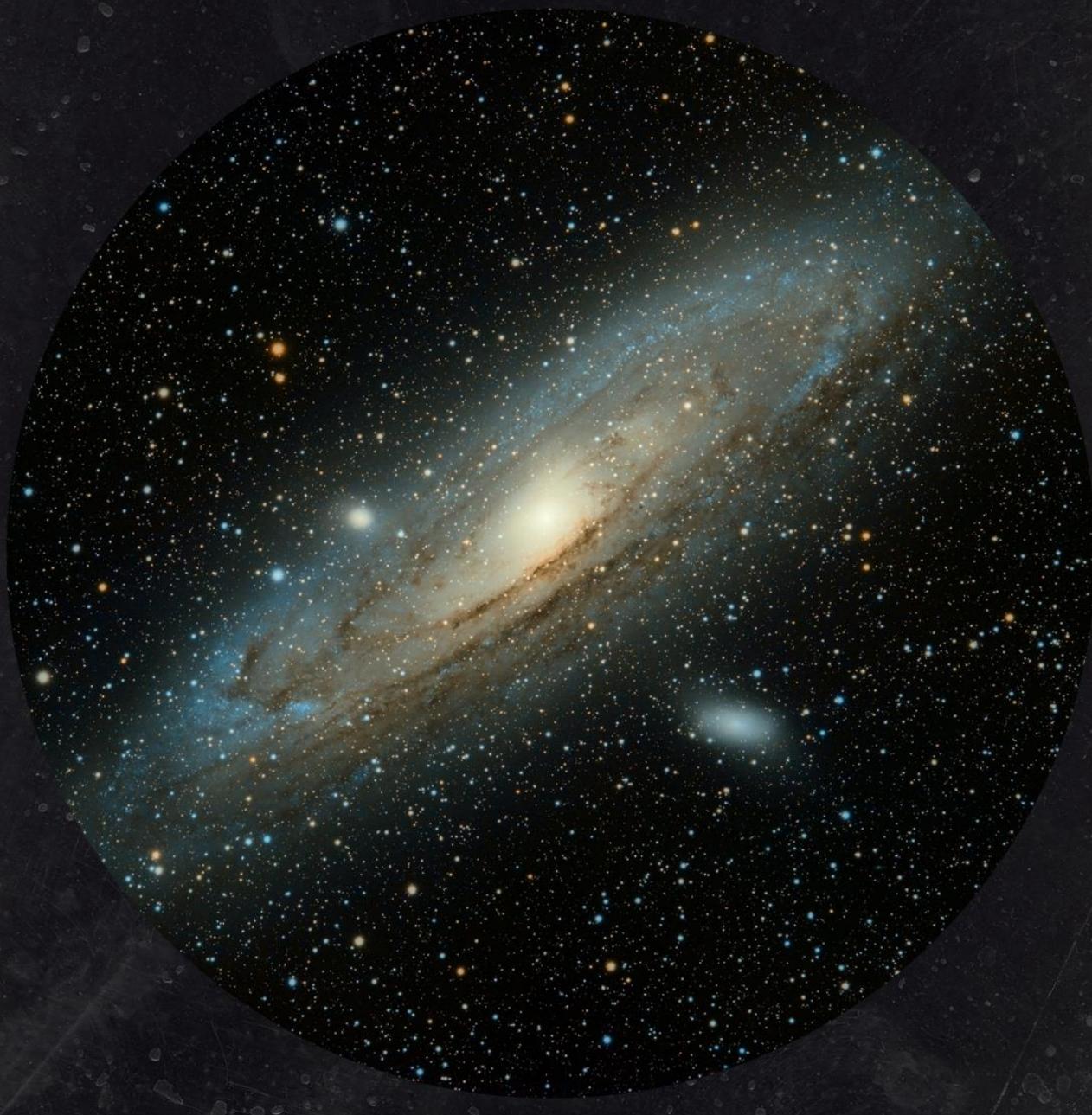
```
let foo;
let bar = typeof foo; // undefined.
foo = 1;
bar = typeof foo; // number.
foo = '1';
bar = typeof foo; // string
foo = true;
bar = typeof foo; // boolean
foo = Symbol();
bar = typeof foo; // Symbol
foo = null;
bar = typeof foo; // object
```

הסבר:

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

פרק 5

בקרת זרימה - מושפי תנאי



בקרת זרימה – משפטים תנאי

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

למשל, הבה נניח המשתנה `foo` מסוים. אם הוא שווה ל-5, יודפס בקונסולה "This is five". אם הוא שווה לאין עושם את זה?

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

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

שימוש לב: `==` נקרא **אופרטור**. כן, בדיק כמו האופרטורים של החיבור, החיסור, הכפל ודומיהם, שלמדנו בפרק המספרים. האופרטור הזה הוא אופרטור של השוואה, ובניגוד לאופרטור נומירי (של מספר) הוא מחזיר ערך בוליאני. כשהלמדנו עליו היה ערטילאי מאד. עכשו הוא קצת יותר הגיוני, אבל רק קצת. בקרוב הוא ייחקר לעומק. בדוגמה שלhallן, אם מזכיר במספר מסוים איזי 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 שכתוב בסוגרים המסורלים מיד אחרי ה-`if`. אם לא, מתרחש מה שכתוב בסוגרים המסורלים אחרי ה-`else`.

כמו שאפשר להשוות מספרים, אפשר להשוות גם מחזוזות 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 למחוזת הטקסט `.five`.

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

אפשר ליצור משפטי 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 `if-else`:
שילוב של `if-else`:

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

כרגע, משפט התנאי if בודק אם foo שווה ל-5. אם כן, קטע הקוד שבתווך הסוגרים המסמלים הראשוניים יופעל. אבל כאן יש גם if עם עוד תנאי, שבודק אם foo שווה ל-6. בסופו של דבר יש תנאי שתופס את הכל. אפשר לשלב כמה if 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 מרובים בקוד זהה, שבודקים את המספר שוב ושוב, בכל פעם נגד תנאי אחר. אם המספר הוא לא 1, 2, 3 או 4, יופעל ה-else האחרון. חשוב לציין שם תוסיפו סתם else באמצע, לעולם לא תגיבו ל-if else שמוספיו אחרים.

עוד שהוא חשוב לציין הוא שצריך להיזהר מהשימוש בתוך משפט תנאי. ג'אווהסקריפט היא סלচנית מדי, ואם תכתבו `u==x` במקום `u=x` הקוד שלכם יעבד ולא יזרוק שגיאה – זו הסיבה שבօperationים השוואתיים צריך לשימושם לב היטב מה כותבים ו לעולם לא לכתוב `=` אחד בלבד.

אופרטורים השוואתיים נוספים

כמו `==` שבודק השווות טהורה, יש אופרטורים השוואתיים נוספים, למשל `!=` (סימן הקראיה משמאל 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 זהה הוא שווה ל-4, וקטע הקוד בתוך הסוגרים המסולסים יפעל. אושר גדול! הנה עוד דוגמה:

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

כאן בודקים מחוץ לטקסט האם `foo` שונה מ-`one`? במקרה זהה כן, ולפיכך הקטע שבתוں הסוגרים המסולסים יופעל.

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

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

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

מה לפי דעתכם יודפס בקונסולה אם תרצו את הקוד הבא?

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

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

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

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

אפשר גם להשתמש באופרטור נוסף שמשמעותו גדול או שווה:

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

כאן אפשר להשתמש באופרטור `=<` שמשמעותו גדול מ-. או שווה ל-. במקרה זהו 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. הינה דוגמה:

```
let foo = -10;
```

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

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

הבה נבחן את כל האופרטורים השוואתיים שהכרתם:

| תפקיד | שם האופרטור השוואתי |
|--------------------|---------------------|
| שווין | == |
| אי-שווין | != |
| גדול מ- | > |
| קטן מ- | < |
| גדול מ- או שווה ל- | >= |
| קטן מ- או שווה ל- | <= |

הסבר מאחורי הקלעים בנוגע לאופרטורים השוואתיים:

האופרטורים השוואתיים בעצם מתרגםים את הביטויים שמעבירים להם ערך בוליאני, קלומראמת או שקר. למשל:

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

זאת אומרת שכמו שהאופרטור הנוומי מוסיף או משנה מספר, האופרטור השוואתי מחזיר תמיד true או false, וזאת בלי קשר למשפט תנאי, אפילו אם סתם זורקים מספרים.

למשל:

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

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

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

אפשר לראות בקונסולה שהיא true. משפט התנאי לא מתעניין באופרטורים, במשתנים או בקשוש אחר. בסופו של דבר, משפט התנאי מתעניין אם מעבירים לו true או false. כך:

למשל:

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

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

אופרטורים לוגיים

האופרטורים הלוגיים מרחיבים את האופרטורים ומאפשרים ליצור תנאים מורכבים יותר. למשל, אם 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-|||. נק' לדוגמה:

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

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

הבה נציגים זאת שוב במחוץ לטקסט. נניח שתרצה לבדוק אם משתנה מסוים שווה למחוץ הטקסט "Sunday" או "sunday". איך עושים את זה? ממש בפשטות. 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. לדוגמה:

```
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-Sunday או ל-1 או ל-.true.

מה קורה מאחרוי הקלעים? האופרטור הלוגי יחזיר true אם אחד התנאים מתממש, תוק שהוא מתעלם מכל השאר. נזכיר, if עובד רק עם true או false וזה מה שהאופרטור הלוגי מחזיר לו.

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

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

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

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

אפשר לשלב בין התנאים באמצעות סוגרים, ממש כמו במקרים אחרות מתמטיות של מודדים בבית הספר. כך למשל כותבים משפט שבו בודקים אם foo או bar שוים ל-1 וגם baz שווה ל-1:

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

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

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

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

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

אם גם `baz` יהיה שווה ל-1, התנאי יתממש.
אפשר להתרעם כמה שروצים 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` אינם 1.
בדרך כלל, אם יש תנאים מורכבים כאלו, סימן שימושו לא בסדר בקוד. קוד אמור להיות קרייא, ואף על פי שמשפט תנאי שייש בו מורכבות כזו הוא אפשרי, לא בטוח שמדובר בנוהג נכון.

אופרטור שלילי

אופרטור לוגי נוסף ו שימושי מאוד הוא **אופרטור לוגי שלילי**. האופרטור הזה קצר קשה להבנה, אך יש לו חשיבות רבה ושווה להשיקע זמן בהבנתו. בגדול, האופרטור הלוגי הזה לוקח כל ערך בוליани והופך אותו. למשל:

```
!true
```

יהיה בעצם `false`. אפשר להשתמש בו במשפט תנאי כדי לומר הפוך. 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 כשהרצים להפוך תנאים, ורוב המתכנים משתמשים באופרטור הזה על מנת להפוך תנאים מורכבים. כך למשל אם יש תנאי שבו אופרטור השוואתי שבודק אם משתנה גדול מ-10, אפשר להפוך אותו כדי לבדוק אם המשתנה קטן מ-10 באמצעות האופרטור `!`. הינה דוגמה:

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

אם נשים שפות שמאלצות שימוש בתנאי שלילי, אבל הפיכת תנאים נחשבת מנוג פסול מאוד ולא הייתה ממליצה להשתמש בה. כדאי לדעת את זה על מנת להבין את פעילות האופרטור `!` שבסופו של דבר ממיר `false`-`true`, אבל מומלץ מאוד לכתוב את התנאים כמו שצריך. למה כן משתמשים באופרטור הזה? הוא שימושי מאוד לבדיקה אם משתנה מסוים הוא "ריק" ולמנוע `else` מיותר. כשתצברו ניסיון בתכנות, תגלו שהמונ פעים צריך לבדוק אם המשתנה מסוים הוא "ריק", וב"ריק" אני מתכוון ל:

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

כלומר מחרוזת ריקה, 0, או אפלו משתנה שלא הוצב בו ערך. כל הערכים האלו ניתנים לבחינה בבת אחת באמצעות 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 האלו 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 |
| && | גם |
| ! | אופרטור שלילי |

אופרטור תנאי

אופרטור תנאי, "if מקוצר" או **ternary operator** הוא האופרטור המשמעותי האחרון שתלמודו. בוגדול, הוא מאפשר להכניס ערכים משתנים לפי תנאים מסוימים. כך למשל אם רוצים שהמשנה `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;
```

ועכשיו ההסבירו: משפט תנאי מקוצר מתייחס בשים של המשתנה, סימן שוויון ואז תנאי עם סימן שאלה. מייד אחרי סימן השאלה מופיע מה שנכנס לתוך המשתנה אם התנאי מחזיר `true`, ואז נקודתיים ומה שנכנס לתוך המשתנה אם התנאי מחזיר `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 המשתנה בשם `message` שמקבל ערך באמצעות התנאי המקוצר. התנאי הוא שהמשנה `age` יהיה גדול מ-18 או שווה לו. אם כן, התנאי המקוצר יחזיר לתוך `message` את 문자ות הטקסט "Drink". אם לא, הוא יחזיר לתוך `message` את 문자ות הטקסט "No drink for you"

אופרטורים המשווים ערכיים

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

נניח שיש את שני המשתנים הבאים:

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

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

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

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

כאשרם חמושים במצורת זו 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 שהתנאי הזה לא יתממש, אף על פי שבשני המשתנים, `foo` ו-`bar`, יש את הערך 1. משתנה אחד הוא מסוג מספר והאחר הוא מסוג 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 `==` (שני סימני שווה ולא שלואה) 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 שלעיל כן יתאפשר, כיון שהאופרטור ההשוואתי `==` (להבדיל מ-`===`) מבצע את המרה של סוג המידע ואז מבצע את ההשוואה – שווי בין ערכים ולא בין סוג מידע.

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

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

– **Falsy**-**Truey**. בשביל זה צריך להכיר את המונחים **מואמת וסקרי**. כאשר ממירים משתנה לסוג מידע בוליאני, בודקים אם הוא **Truey** או **Falsy**. כל סוג מידע וערך בג'אוּהַסְקְּרִיפְט נחשב ל-**Truey** למעט הערכים הבאים: `false`, `NaN`, `undefined`, `null`, `""`, `0`.

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

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

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

למה זה ככה? כי מחרוזת ריקה מתורגמת ל-`false`. זוכרים ששניהם `False`? אם כן, שניהם `false` ואז נעשית ההשוואה, וזה עלול להיות בדיתי. חשבו על מערכת שמקבלת קלט מספרי ועושה איתו דברים. אם נעשתה טעות ונשלח קלט ריק, ואז משתמשים בהשוואה רגילה, תהיה תקללה המערכת. היא תקבל קלט ריק ותחשוב שהוא אפס, וההתנהגות בהתאם.

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

תרגיל:

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

פתרון:

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

הסבר:

המילה השמורה `if` בודקת את המשפט שיש בתחום הסוגרים העגולים. אם הוא נכון, החלק בתחום הסוגרים המסורלים יירוץ. אם לא אז לא. בתחום הסוגרים יש אופרטור לוגי מסוג שווון שבודק אם המשתנה הוא 10. למשל, ידפיס את המשתנה:

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

קטע הקוד זהה, לעומת זאת, לא ידפיס אותו:

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

למה? כי 10 הוא לא 9.

תרגיל:

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

פתרונות:

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

הסבר:

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

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

יפעל. אם לא, קטע הקוד בתוך הסוגרים המסוללים 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.

פתרונות:

```
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 השוואת. אם המשתנה `day` הוא 1, כל קטע הקוד שבין הסוגרים המסלולים שבאים אחר כן יבוצע, והקונסולה תדפיס את השם `Sunday`. מייד אחר כן יש `if` שבודק אם המספר הוא 2, ואז הקונסולה תדפיס את השם `Monday`, וכך הלאה. שימושו לב `else if` ולסוגרים שבתוכם יש את התנאי ולסוגרים המסלולים שפועלים אך ורק אם התנאי נכון. בסוף הקוד יש `else` שמתקיים אם כל ה-`if` שבו לפני לא עבדו, ורק אם הם לא עבדו.

תרגיל:

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

תרגיל:

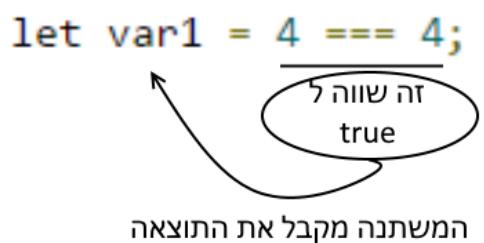
השתמשו בחמישה אופרטורים השוואתיים כדי להכניס `true` או `false` לחמישה משתנים שונים. הדפיסו אותם בקונסולה.

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 שprzedנו, האופרטורים ההשוואתיים מחזירים ערך בוליאני – `true` או `false`. פשוט יוצרים משתנה ומכניסים לתוכו את תוצאות פעולות האופרטורים האלה, כיוון שהאופרטור הבוליאני מקבל כל מספר, אין זה משנה אם הוא בתוך משתנה או לא. הדיאגרמה הבאה תבהיר:



תרגיל:

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

פתרונות:

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

הסבר:

cotבאים את התנאים כרגע ומחברים ביניהם באמצעות האופרטור הלוגי "או", שכותוב `C||I`. יש בעצם שרשרת של תנאים שבין כל אחד מהם נמצא האופרטור הלוגי `||`. כך בעצםcotבאים "או" ג'אווהסקריפט.

תרגיל:

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

פתרונות:

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

הסבר:

יש כאן שני משפטי תנאי עם אופרטורים השוואתיים:

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

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

תרגיל:

כתבו משפט תנאי שבודק אם foo הוא 0 או null ואם כן, מדפיס בקונסולה not I "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 מתכנתיו הג'אוּהַסְקְּרִיפְט היו בוחרים לעשות. באמצעות האופרטור הלוגי ! מקבלים true על foo אם foo הוא null, 0, מחוץ טקסט ריקה או false. הפתרון השני הוא אינטואיטיבי יותר למי שעושה את צעדיו הראשונים בשפה ומכל שני אופרטורים השוואתיים ואופרטור לוגי || שימושתו היא "או": foo === 0 או foo === null. שימושו לב ש-null היא מילה שמורה (לא מוקפת במירכאות). null הוא סוג מידע מיוחד שלמדו עליינו בפרקם הקודמים.

תרגיל:

כתבו משפט תנאי שמשווה בין:

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

ומצליחות.

פתרון:

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

הסבר:

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

פרק 6

SWITCH CASE



switch case

כמו משפט תנאי, גם משפט **switch case** מאפשר לבצע קטעי 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" וכן הלאה. אם לא מדובר במספרים 1–7, הקונסולה מדפיסה הודעה שנייה. איך זה נראה? ככה:

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

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

נשמע מסובך? 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 בהצהרת **switch** שהיא מילה שומרה. **ב-*switch*** שמים משתנה שיכול להכיל כל דבר ואחריו סוגריים מסולסים. בתוכם יש **הצהרות *case***.

1 case למשל אומר שבמקרה שהערך הוא 1 – שימושו לב שמדובר פה במספר ולא במחוזת טקסט – אם הערך עונה על התנאי זהה, כל הקוד שיש בין הנקודותים ל-**break** מתקיים. אם שום תנאי לא מתקיים, כל מה שקרה אחריו ה-**default** מתקיים עד ה-**break**. לא חייבים את ה-**break** האחרון, אבל הצבתי אותו בשבייל 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 **default** תמיד יכולה למנוע בעיות וצורות אחרות.

הבה נציגים זאת בדוגמה אחרת – קוד שמחזיר שם של חודש לפי מספר. עם if else היה קשה מאוד לעשות את זה, אבל עם switch זה ממש קל:

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

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

```
let monthNumber = 5;
let monthName;
let season;
switch (monthNumber) {
    case 1:
        monthName = 'January';
        season = 'Winter';
        break;
    case 2:
        monthName = 'February';
        season = 'Winter';
        break;
    case 3:
        monthName = 'March';
        season = 'Winter';
        break;
    case 4:
        monthName = 'April';
        season = 'Winter';
        break;
    case 5:
        monthName = 'May';
        season = 'Summer';
        break;
    case 6:
        monthName = 'June';
        season = 'Summer';
        break;
    case 7:
        monthName = 'July';
        season = 'Summer';
        break;
    case 8:
        monthName = 'August';
        season = 'Summer';
        break;
    case 9:
        monthName = 'September';
        season = 'Summer';
        break;
    case 10:
        monthName = 'October';
        season = 'Winter';
        break;
    case 11:
        monthName = 'November';
```

```

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

```

גם פה, כמו בקטעי הקוד הקודמים, לוקחים משתנה מסוים ושים אותו בתוך ה-**switch**. **switch** הויא כאמור מילה שומרה שמודיעה למנוע של ג'אווהסקריפט שיש פה תנאי switch case. מיד אחרי ה-**switch** והמשתנה שנמצא בסוגרים העגולים יש סוגרים מסולסים, המכילים תנאים שנקראים **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 זהה (שבהחלט יעבד):

```

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 כמו:

```

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 יעצור ואפשר להכניס כמה תנאים שרוצים. נשאלת השאלה, למה להשתמש בזו? אחרי הכל, בעזרת `if else` אפשר לכתוב את הקוד שלויעיל בלי ללימוד `switch case` מעיק. זו טענה ולידית, 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 מבלב וקשה לקריאה ולתיקון. בלי מעת מקרים, `switch case` יכול להקל את הקריאה ומשתמשים בו הרבה. לפיכך כדאי וצריך להכיר אותו ולהשתמש בו בכל פעם שיש יותר משלושה `else if`. זה לא כלל ברזל, אלא רק במקרה של תנאי לא מסובך מדי – אתם לא רוצים להתחיל להכניס משפטי תנאי והצבות בתוך `switch case`-ו שלכם.

תרגיל:

באמצעות 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:

כתבים את המילה השמורה switch, שמקבלת משתנה בסוגרים העגולים. המשתנה זהה ייבחן נגד ערכים שונים. פורמת הבדיקה הוא זהה:

case value:

המילה השמורה case ומיד אחריה מגיע הערך. אם הערך זהה תואם את הערך שיש במשתנה שנמצא בסוגרים העגולים ליד ה-switch, הקוד שמופיע מיד אחרי הנקודות ועד ה-break יירוץ. במקרה זהה מדובר בהדפסה בקונסולה.

שימוש לב ששמתי ערך דיפולטיבי שמודפס אם במשתנה יש ערך שלא צייןתי במדויק
ב מבחנים. זה נעשה באמצעות התנאי default.

פרק 7

הבועשים

קבועים

עד כה למדנו על משתנים,قلומר על אבני 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 `to undefined`, סוג מידע של מדנו עליו קודם לכך.

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

```
const MY_CONSTANT = 10;
```

קונבנצייה אחת שבה משתמשים כדי להגדיר אותם היא אותיות גדולות עם קו תחתון שטפניד בין מילים, אבל ראוי לציין שימוש לא חובה לעשות את זה. אפשר להגדיר קבוע בדיק נמו כל משתנה אחר. אם מנסים לשנות את הקבוע, מקבלים שגיאה בזמן הרצת הקוד.

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

נשאלת השאלה, מה הגביל את עצמנו ולהשתמש בקבועים? קבועים יכולים יותר מבחינות זיכרון מסיבות שלא אנחנו אליהן כאן. ואם זה נשמע לכם איזוטרי, אפשר לבוא ולומר ששימוש מושכל-ב-`const` מעלה את ערככם בעיניי כל מי שיקרא את הקוד שלכם. לפיכך כדאי וצריך להכיר אותו. בשלב זהה של הלימוד קשה לדון על מערכות מורכבות,

אך במערכות מורכבות זה קרייטי לתת ערכיים שאף אחד לא יוכל לשנות או להשפייע עליהם. זו הסיבה שאנו לומדים ומשתמשים בקבועים.

תרגיל:

הגדרו קבוע בשם `foo` והכניסו לתוכו מחזורת 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 בדיק כמו משנה.

תרגיל:

הגדרו קבוע בשם `foo`, הכניסו מספר, שנו את המספר ובחנו את התקלה שהקונסולה מציגה.

פתרון:

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

הסבר:

מגדירים קבוע בשם `foo` באמצעות המילה השמורה `const` 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". הטקסט המדוקן עשוי להשנות בהתאם לסוג הדף או לגרסתו, אבל שגיאה תהיה שם.

פרק 8

בקרת זרימה - פונקציות



בקרת זרימה – פונקציות

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

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

מבלב? הנה נסתכל על הדוגמה הבאה:

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

ראשית הוגדרה פונקציה בשם `myFunc`. ההגדרה נעשית באמצעות המילה השמורה `function` ושם הפונקציה. במקרה זה נבחר השם `myFunc`. אחרי השם יש סוגרים ריקים ואז סוגרים מסולסים. בתוך הסוגרים המסולסים מופיעה הפונקציה עצמה, ייחידה אוטונומית של קוד שלא תروع אם לא נקרא לפונקציה. במקרה זה אין מי-יודע-מה קוד רק המילה השמורה `return` ו-1. כלומר מי שיקרא לפונקציה יקבל 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 מכנים למשתנה. מה לפि דעתכם יקבל המשתנה? את מה שהפונקציה מחזירה. במקרה זה 1.

הבה ננסה ליצור עוד פונקציה:

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

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

אפשר לחלק את עניין ההגדרה וההפעלה של פונקציה לשלווה חלקים. החלק הראשון – הוא הגדרת הפונקציה באמצעות המילה השמורה `function`. החלק השני הוא ההחזרה – מה הפונקציה מחזירה. החלק השלישי הוא הקראיה, והוא נעשה מחוץ לפונקציה. הקראיה 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"
```

ובן שהפונקציה יכולה לעשות עוד דברים ולא רק להחזיר מחזורת 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` כרגיל ומקפידים לשימוש סוגרים עגולים. בתוך הסוגרים המסורלים מתרחש האksen: מגדירים משתנים, עושים חיבורים ומקבלים תוצאה שאותה מחזירים עם `return`.

כל הדבר הנחדר זהה הפונקציה לא יעבד אם לא תקרוו לו. קוראים לו באמצעות קריאה בשם הפונקציה והסוגרים העגולים והכנסת מה שהיאמחזירה לתוכה המשתנה `foo`. במקרה זה הוא יקבל את מה שהפונקציה מחזירה, זהה 30. החגיגה שמרתחשת בתוך הפונקציה – כלומר הגדרת קבועים ופעולות מתמטיות – מתרחשת אך ורק בתוכה. היא לא זולגת החוצה. זה מה שיפה בפונקציה – אפשר לעשות מה שרצים בתוכה וזה לא יחולג החוצה. בדיקן מולאס וגאס – מה שקרה בפונקציה נשאר בפונקציה, והדבר היחידי שזולג ממנה הוא ה-`return`.

את היכולת המופלאה זו אדגים באמצעות הדוגמה הבאה:

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

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

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

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

יקפי שגיאה ולא ירוץ. מנוע הג'אווהסקריפט לא יכול להריץ אותו כי אחד מכללי השפה הבסיסיים הוא שאי-אפשר להגדיר משתנה קבוע בעלי אותו שם. אז איך זה קרה פה?

התשובה היא כאמור שמה שנמצא בתחום הסוגרים המסולסים, כלומר בתחום הפונקציה, מתקיים ב.TODO ולא זולג החוצה. הממד האחר זהה נקרא "לקסיקלי סקופ" (Lexical Scope), ולפונקציה יש סקופ מיוחד, ככלומר ממד שבו אפשר להגדיר משתנים שיהיו מבודדים לחלוטין מהסקופ של מי שקורא לפונקציה. נושא הסקופ ייחזור בהמשך כנושא סוק בנותאים מתקדמים יותר.

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

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

```
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 השתמשם לבדוק באותו שםות שבם קראתם קבועים בתחום הפונקציה, והכל עובד. למה? כי מה שקרה בתחום הסקופ של הפונקציה לא רלוונטי ולא זולג לסקופים אחרים. הסקופ שבו עובדים נקרא "סקופ הגלובלי" ואם קוראים לפונקציה מתוכו, דבר לא יזלג החוצה אליו. בפונקציה אפשר לעשות כל מה שרצים ולקרא למשתנים כרצונכם. אמשיך לכתוב על סקופים בהמשך.

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

ברגע שיש `return`, הפונקציה מחזירה את הערך שנכתב. אבל מה קורה אם לא נכתב דבר? מהهو בסגנון זהה:

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

מה שיקרה הוא שהפונקציה מחריז `undefined`. זוכרים אותו? למדנו עליו בפרק על סוגי מידע פרימיטיביים נוספים. זהו מבנה נתוני בסיסי והוא חוזר כאשר כותבים סתם `return` או כאשר אין `return` כלל. למשל פה:

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

אם יש כמו `return`, ה-`return` הראשון 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. שימוש ללב: אפשר להגיד כמה פונקציות שרוצים ולקרא להן איך שרוצים. בכל הדוגמאות הוגדרה פונקציה אחת, אבל אין שום בעיה להגיד כמה פונקציות שרוצים.

פונקציה עם ארגומנטים

אפשר להעביר לפונקציה ערכים מוחוץ ולקבל אותם בתוך הפונקציה. זה נעשה בשני אופנים. ראשית בהגדרת הפונקציה. זוכרים את הסוגרים העגולים הריקים? בעצם הם לא יהיו ריקים – אני אכנס לתוכם משתנה. המשתנה הזה מוגדר בתוך הפונקציה כאילו הגדרתי אותו עם `let`. אפשר להכפיל אותו או לחת אותו ולהציב אותו איפה שרצים, בדיקן כמו משתנה רגיל. השם המקובל למשתנה זה הוא ארגומנט.

איך קובעים אותו? אפשר להכניס אותו בקריאה עצמה. הינה, הבינו בדוגמה:

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

הfonקציה `multiply` מקבלת ארגומנט בשם `arg1`. כאמור, ברגע שמכריזים עליו בסוגרים אפשר להשתמש בו בfonקציה עצמה. במקרה זה לוקחים אותו, מכפילים אותו ומכניסים את התוצאה למשתנה `answer`.

איך 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 שהfonקציה 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 Fonקציה, אבל התוצאה שלה משתנה בהתאם לארגומנט שמעבירים. כשקוראים לפונקציה כך:

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

כשקוראים לפונקציה כה:

```
multiply(20);
```

از 1 arg בתוך הפונקציה מקבל את הערך 20 וההתשובה המוחזרת היא 40. ככלומר, הפונקציה נותרת כפי שהיא, רק הקראיה שלה משתנה.

שימוש לב: בדרך כלל משתמשים בפונקציות כאשר כתבים ג'אווהסקריפט כי זו הדרך הטובה ביותר לפרק את הקוד לחלקים קטנים ולהימנע מחזרות על קוד. חזרות על קוד הן דבר שਮוטב להימנע ממנו בקוד כיון שם רוצים לשנות אותו, נדרשת עבודה רבה. אפשר להשתמש במקרה 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 בשםות שרוצים ומפרידים ביניהם באמצעות פסיק בהגדרת הפונקציה. כך למשל מגדירים פונקציה בעלת שלושה ארגומנטים:

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

אפשר להשתמש במקרה ארגומנטים האלו כמשתנים מן המניין בתוך הסוגרים המסולסים של הפונקציה. מה שחשוב הוא שהפונקציה תעשה return. כשתקראו לפונקציה, תכניסו שלושה ארגומנטים בקריאה:

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

כל ארגומנט יופרד בפסיק.

ארגומנטים עם ערכים דינמיים

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

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

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

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

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

אפשר להכניס ערך בריית מחדל (דינמי) לארגומנטים שייכנס אליהם אם מי שקרה לפונקציה לא העביר לה ארגומנטים. עושים זאת באמצעות פונקציית `default` – 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 זה את הערך 1 ואז התשובה תהיה 2. יש לזכור שהשמה של ערך נעשית על ידי הסמן `=`.

אם לפונקציה יש כמה ארגומנטים, בג'אווהסקריפט אפשר להכניס לכולם ערך בריית מחדל.

הסתכלו על הדוגמה זו:

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

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

שימוש לב לדוגמה האחרונה: הארגומנט הראשון שמעבירים הוא undefined, שהוא מילה שומרה שהוזכרה בפרק על סוגי מידע. אני בעצם מצהיר שהารגומנט הראשון arg1 הוא undefined ומשמעותו את ברירת המחדל. שימוש לב שמדובר ב-undefined אמייתי ולא בערך ריק כמו מחוץ לריקה או אפילו null.

שימוש לב: זה נחשב לנוהג רע להכניס יותר מחמישה ארגומנטים לפונקציה. אם בפונקציה שלכם יש יותר מחמישה 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 שלנו. שימוש 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'אווהסקריפט היא מבנה נתונים חדש. בניגוד לערבים הפרימיטיביים שהכרתם עד כה, פונקציה היא ערך לא פרימיטיבי ומורכב יותר. צריך לזכור שפונקציה בסופו של דבר היא מבנה נתונים שחייבים להכניס למשתנה בדיקות 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. למעשה הבדל אחד קטן ומשמעותי שנקרא Hoisting – העמסה.

Hoisting

כאשר מגדירים פונקציות, המנוע של ג'אווהסקריפט מזיז את כלן למעלה. לשם ההדגמה, מה לפיה דעתכם תהיה תוצאה הקוד הזה?

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

משתמשים בפונקציה לפני שהיא מוגדרת, והקוד הזה עובד. למה בדיק הוא עובד? כי המנוע של ג'אויסקייפ, לפני שהוא מריץ את הקוד – מזיז את כל הפונקציות למעלה. כלומר, הקוד מורץ כך בפועל, אפילו שלא נכתב כך:

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

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

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

יקבל שגיאה, כי כאמור קוראים לפונקציה לפני שמגדירים אותה. יש לזה משמעות כאשר עובדים בקוד מתקדם יותר ולא בהתחלה, כמובן.

closure

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

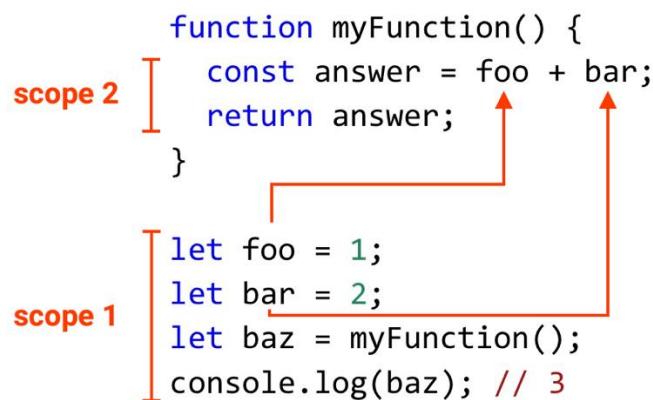
```
function myFunction() {
    console.log(foo);
}
let foo = 'Hello';
myFunction(); // Hello
```

מה קורה פה? הפונקציה `myFunction` היא פשוטה למדי ומדפיסה משתנה בשם `foo`. אבל שימושו לב ש-`foo` מוגדר מחוץ לפונקציה! ועודין, 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 תדוע מה הערך של `foo` ושל `bar` אף על פי שהם לא מוגדרים בה אלא מחוץ לה. ככלומר, `foo` ו-`bar` לא מוגדרים בסקופ של הפונקציה אלא בסקופ הגלובלי וודאי שלא מועברים לפונקציה כארגומנטים, ועודין הפונקציה מכירה אותם.

הדיagramה זו אמורה להבהיר את העניין:



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

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

```

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

כאן אפשר לראות שבתוך הסקופ של הפונקציה דורסים את `foo`. `foo` שמוגדר מחוץ לפונקציה כמו כן לא נפגע, אבל מה שמתתרחש בתוך הפונקציה הוא ש-`foo` הופך לעצמאי לחלוטין.

ובן שה-`closure` הוא דו-כיווני ושהאפשר לשפייע על המשתנים בסקופ האב דרך סקופ הבן.

למשל:

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

שימוש לב שבתוֹך הפונקציה לא הוגדר משתנה foo באמצעות let. מהרגע שעושים את זה משנים נमובן את המשתנה שנמצא בסcopּה האב.

כלומר, כל משתנה שמוגדר מבחוּץ חשוף לפונקציה הפנימית. זה כדי חזק שעלול ליצור כאוס משמעותי. בגרסאות קודמות של ג'אווהסקריפט (ES5 ומטה) הכאוס היה גדול אף יותר כיוון שהגדרת המשתנה, שנעשתה באמצעות var, אפשרה ל-var להיכנס תמיד לסקופּ הגלובלי. ה-let מוגבל יותר.

שימוש לב: כרגע זה נראה תיאורטי למדי ואולי מטופש, אבל יש חשיבות עליונה להבנת ה-closure. בלולאות ובפונקציות רקורסיביות, ובטע ובטח באפליקציות מורכבות יותר, יש הושענות רבה על התוכנה זו. לפיקן כדאי להשקיע זמן בלימוד שלה ועוד אחזר לנושא בהמשך.

פונקציה אNONIMIת ופונקציית CH

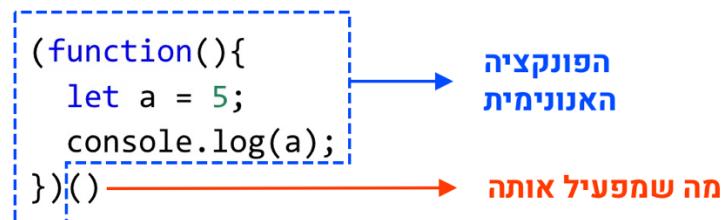
פונקציה אNONIMIת היא פונקציה ללא שם. אחד השימושים המוכרים יותר הוא פונקציה שמריצה את עצמה. איך זה נראה בדרך כלל? ככה:

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

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

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

אם לא תתעלמו ותריצו את קטע הקוד שלעיל, תראו שהוא אכן רץ. למה? כי הפעלתם את הפונקציה האNONIMIת מייד לאחר היצירה שלה.



פונקציות אNONIMIות, ככלומר كانوا ללא שם, משמשות בהמון מקרים ומקומות בג'אוوهסקריפט, בין שמריצים אותן מייד לאחר הפעלה ובין שלא.

בג'אווהסקריפט מודרנית, אפשר לכתוב פונקציה אונונימית ללא שימוש במילה השמורה אלא באמצעות **פונקציית חץ** או, באנגלית, **function**:

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

זה בדוק אותו דבר, אבל מבחינת הסkop יש לפונקציות חץ כמה יתרונות גדולים ונכדי להשתמש בהן. נשאלת השאלה, למה להשתמש בכלל בפונקציה אונונימית? ויש לה כמה תשובות.

פונקציה אונונימית כמשתנה

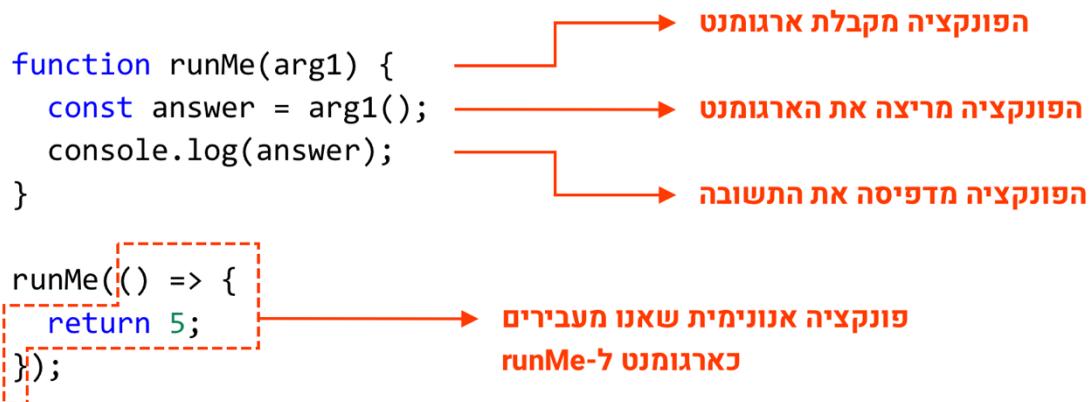
יש פונקציות שמקבלות משתנים שאמורים להיות פונקציות.
מה זאת אומרת? למשל הפונקציה זו:

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

מה שמתתרחש פה הוא שיש שני פונקציות. פונקציית returnSomething ממחישה 100. פונקציית runMe מקבלת משתנה. מה היא עשו בו? לוקחת אותו ו"מירותה" אותו באמצעות סוגרים פותחים וסגורים. ככלומר היא מניחה שהארגון הוא פונקציה. במקרה זה הוגדרה פונקציה בשם returnSomething והועברה כמשתנה לפונקציית runMe. אבל אפשר להעביר לפונקציית runMe גם פונקציית חץ אונונימית:

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

אם תרצו את זה תראו הדפסה של מה שהפונקציה האונונימית ממחישה.



זה נראה מעט אבסטרקטי ואף מיותר, אבל משתמשים בזה בהמון מקומות בג'אווהסקריפט. המוקם הראשון שבו רואים את השימוש בפונקציות אונונימיות הוא בลอלות, אבל לא רק שם. יש לא מעט מקומות שבהם מעבירים פונקציה כארוגומנט. הfonקציה שעוברת כארוגומנט נקראת **"콜בק"** ובאנגלית **callback**. השם הזה נותן לה כי היא **"נקראת בחזרה"** על ידי הfonקציה ברגע המתאים. זכרו את השם **"콜בק"**, עוד אשוב אליו.

אפשר להעביר גם ארגומנטים ל科尔בק. שימו לב לדוגמה זו:

```

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

```

היא זהה כמעט לchlוטין לדוגמה הקודמת, אבל במקרה זה מצפים שבפונקציה האונונימית שמעבירים יהיה ארגומנט אחד. במקרה הזה מעתירם 2 לארוגומנט. אם העניין אבסטרקטי מדי זה בסדר, רק זכרו שקולבקים יכולים לקבל ארגומנטים שהfonקציה הקוראת להם מעבירה. במקרה הזה פונקציית Me_{func}, שאליה מעבירים ארגומנט שהוא פונקציה (או קולבק), מעבירה 2 לארוגומנט ל科尔בק.

פונקציה אונימית שמבודדת מהסקופ הגלובלי

פונקציה אונימית שמריצה את עצמה היא נהדרת בכל מה שקשרו לביצוע מהסקופ הגלובלי ולמתן אפשרות למכנת להגדיר שכבה שرك דרכה אפשר להגיע אל רכיב הקוד שהוא כתב. זה מעט מתקדם מדי נוכח לעכשו, אבל אנסה להסביר בכל מקרה. שימושו בפונקציה 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 האונימית שמריצה את עצמה מקבלת ארגומנט בשם jQuery. כל מה שתרחש בתחום הפונקציה זו נשאר חסוי מסקופים אחרים. אם מגדירים foo (לא משנה אם בעזרת let או var) אז הוא יישאר חסוי. אפשר לבחור לחושף את bar אם מצמידים אותו לאובייקט jQuery.מן הסתם, אם רוצים להשתמש באובייקט jQuery על מנת לתקשר עם הסביבה החיצונית, הסביבה החיצונית תctrck ליצור אותו.

כאמור, אם זה נשמע תלוש מעט זה בסדר גמור. בשלב הזה של הלימוד טוב לזכור שפונקציה אונימית שמריצה את עצמה לגיטימית לשימוש בספריות ג'אוوهסקריפט שונות (כמו ספריית jQuery) ובכל מודול שהוא, שבו רוצים למש סקופ נפרד ופרטי. לא נDIR לראות בראשת דוגמאות של פונקציות אונימיות 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:

יצרים פונקציה באמצעות המילה השמורה function וסוגרים עגולים. בסוגרים המסלולים רואים מה שקורא בתוך הפונקציה. בפונקציה לא קורה כלום והוא מחזיר רק null. מפעילים את הפונקציה באמצעות () myFunc ומכניסים את מה שהוא מחזיר, במקרה זהה null, אל המשתנה foo, שאותו מדפיסים בקונסולה בשורה הבאה.

תרגיל:

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

פתרון:

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

הסבר:

יצרים פונקציה באמצעות המילה השמורה function וקוראים לה ahlaBahla. מיד אחרי שמה שמים סוגרים עגולים (). בתוך הסוגרים המסלולים, שחייבים לשיט, מתקיים הפונקציה. במקרה זה היא לא עושה הרבה אלא רק מחזיר 100. איך היא מושה? באמצעות המילה השמורה return, שמחזירה את מה שכותב אחרת, במקרה זהה .100.

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

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

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

תרגיל:

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

פתרון:

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

הסבר:

יצרים פונקציה בשם `yay` עם המילה השמורה `function`. מיד אחרי ההגדלה יש סגירים מסולסים, שבהם מתרחש כל מה שקרה בפונקציה. במקרה זה, דבר לא מתרחש. הפונקציה מחזירה את מחזורת הטקסט "yay" באמצעות המילה השמורה `return`.

מה שקורא לפונקציה – כי לא הקראיה היא לא תופעל - הוא הקוד הבא:

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

הקוד הזה עושה שני דברים – קורא לפונקציה ועביר את מה שהוא מחזיר למשתנה `foo`. המשתנה `foo` יודפס ויכיל את מה שיש בו, כולם מה שהפונקציה החזירה.

תרגיל:

כתבו פונקציה שמקבלת ארגומנט, מדפיסה אותו בקונסולה ועוד מוסיפה לו 1 ומוחירה אותו. קראו לפונקציה עם ארגומנט 1 והכניסו את התוצאה שלה למשתנה. הדפיסו אותו בקונסולה.

פתרונות:

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

הסבר:

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

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

כלומר, מגדירים פונקציה ובתוכה הסוגרים העגולים מכניסים ארגומנט. עכשו צריך 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. עכשו צריך להוסיף לו 1 ולהחזיר את התוצאה. את זה כבר הכרתם:

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

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

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

מעבירים כารוגמנט את המספר 1 לפונקציה ומכניסים את מה שהיא ממחישה לתוך המשתנה foo, אותו מדפיסים.

תרגיל:

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 קראיות 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:

委副书记 את הפונקציה כמו כל פונקציה אחרת עם ארגומנט שהוא מקבלת. בחרתי את השם number עבור הארגומנט. הארגומנט הזה הוא משתנה לכל דבר בתוך הסkop של הפונקציה, ואפשר לכתוב עליו משפטי תנאי כפי שלמדנו בפרק הקודמים. שימו לב שלפונקציה זו אין return כיון שהיא לא ממחירה ערך אלא מדפסה בקונסולה בלבד. לפיכך גם לא צריך להכניס את מה שהיא ממחישה לתוך משתנה אלא לבצע קראיה בלבד שmapsailה אותה.

תרגיל:

כתבו פונקציה שמקבלת מספר ובודקת אם הוא מספר או סוג מידע אחר. אם הוא סוג מידע אחר היא תדפיס בקונסולה a/h ותסימן 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 %, אופרטור הבודק חילוק לפי שארית. למדנו על האופרטורים האלה בפרקם הקודמים.

פתרונות:

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

הסבר:

את הפונקציה מגדירים כרגע, כמו בתרגילים הקודמים, כולל ארגומנט. הארגומנט חי וקיים בפונקציה כמו כל משתנה.שמו הוא number. ראשית בודקים אם סוג המידע שלו שונה ממספר באמצעות האופרטור typeof שמחזיר את סוג המידע הקיים במשתנה. אם הוא לא מספר מדפיסים בקונסולה a/h ומביצים return. ה-h return זהה, כפי שלמדנו, יסימן את פועלות הפונקציה סופית, אבל הוא יופעל, כמובן, רק אם התנאי יתממש.

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

תרגיל:

צרו פונקציה שמקבלת ארגומנט. הארגומנט-Amor להיות פונקציה. אם הוא לא פונקציה מופעלת שנייה. אם הוא אכן פונקציה מרים אותו ומדפיסים את התוצאה. בדקו פעמיים – עם פונקציה אונונית שמחזירה מחוזת טקסט ועם פונקציה אונונית שמחזירה מספר.

פתרון:

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

הסבר:

פונקציית callMe מקבלת את arg1. בודקים אותו באמצעות האופרטור typeof. אם הוא לא פונקציה, מדפיסים הודעה שנייה ומוחזירים את הפונקציה כריקה. אם arg1 הוא פונקציה, מתיחסים אליו כפונקציה וקוראים לה. את מה שהוא מוחזירה מדפיסים.

עד כאן זה פשוט; הבדיקה מעט יותר מסובכת. על מנת לבדוק צריך ליצור פונקציה שמחזירה משהו. את זה עושים באמצעות פונקציית חץ אונונית. הפונקציה האונונית הרשונה מוחזירה מחוזת טקסט והפונקציה האונונית השנייה מוחזירה מספר.

פרק 6

אובייקטים



אובייקטים

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

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

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

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

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

אובייקט יכול להכיל בתוכו מידע לפי "מפתחות". מפתחות הם דרך להכניס מידע נוסף תחת שם מסוימת לאובייקט – כך שיהיה לנו קל לראות את המידע הזה ולשייך אותו לקטגוריה (שהיא בעצם ה"פתח"). כך למשל מכנים מידע לאובייקט תחת המפתח id:

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

יוצרים אובייקט ומכניםים אותו למשתנה myObject. יצירת המפתח נעשית באמצעות החלטה על שם המפתח (במקרה זה id) והכנסת ערך, בדיקת כמו משתנה. שימוש לבן קודה בין myObject ל-id. זהו אחד המבנים הבסיסיים של השפה.

אפשר להכניס עוד מפתחות, כמובן. למשל מחרוזת טקסט:

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

גישה למפתח נועשית באמצעות שם המשתנה שמכיל את האובייקט, נקודה ואז 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 וכמובן לשנות אותם. בדיק-בדיק כמו משתנים! כאן לדוגמה משנים את ה-*id* כמו פעים:

```
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 הלאה. אפשר ליצור את האובייקט **ישירות**, ממש כך:

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

זה בדיק כמו ליצור את המפתחות בדרך הקודמת. הכל מאוד-מהוד גמיש וצריך לזכור את זה. הדרך הפשוטה ליצור האובייקט – קלומר יצירה שלו כבר עם כל המפתחות מראש – מקובלת יותר וגם עדיפה מבחינת ביצועים.

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

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

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

לא להתבלבל! יוצרים כאן שנישמות משתנים שבמקרה זהים לחלוטין לשםות המפתחות. נcone, name עלול לבלב – אבל הראשון הוא המפתח והשני הוא הערך או, נכון יותר, שם הערך שהמפתח נכנס אליו. חשוב לציין שמדובר ההצבה של המשתנה במפתח המתאים באובייקט, הקשר בין המשתנה לערך נעלם. המשתנה יכול להשנות, אבל הערך יישאר כפי שהוא ברגע ההצבה. 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 – מתכנתים אוהבים מאוד להשווות את שמות הארגומנטים לשמות המפתחות. משהו בסגנון זהה:

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

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

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

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

כאמור, אובייקט יכול להכיל כל נתון שהוא בתור ערך, כולל... אובייקטים אחרים! שימוש 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 זו והאובייקט המלא יוצג לפניכם.

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

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

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

משתמשים בזה בעיקר כאשר שם המפתח נמצא בתחום משתנה מסווג מחרוזת 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 ובלולאות. פעמים רבות מקבלים את המפתח כמחרוזת טקסט. על מנת לקרוא למפתח מתוך האובייקט משתמשים בסוגרים מרובעים, ובתוכם שמיים את המשתנה שמכיל את שם המפתח.

אם תנסה לעשות משהו זהה (propertyName) הוא המשתנה שמכיל את שם המפתח
תקבלו undefined.

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

לפיכך לעולם לא משתמשים בנקודה אם שם המפתח נמצא בתחום 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, כמוון, בנקודה 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 סוגי מידע פרימיטיביים, כמו מספר וטקסט, וגם הראיתי שאפשר להכניס אובייקט כערך לאובייקט אחר. מובן 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` ומכוונים אותו לפונקציה שלל מה שהוא עושה הוא להדפיס בקונסולה את מחזורת הטקסט "`hi!`". יצירת הפונקציה זהה במאה אחוז ליצירת פונקציה רגילה והכנסתה לתוך משתנה. הקראיה לפונקציה בתוך אובייקט גם כן זהה לחלווטין לקרוא אותה לפונקציה ונעשה כפי שקרהים בכל ערך אחר.

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

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

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

קינון ובדיקה מפתח

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

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

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

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

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

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

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

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

השגיאה מתאפשרת כיון ש-myObject.address הוא undefined ו-אנו מנסים להתייחס אליו כאובייקט. הפתרון הוא להשתמש בשרשור אופציונלי (Optional Chaining). שזו נקודה עם סימן שאלה לפניה. השרשור האופציונלי לא יציג שגיאה אלא undefined.

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

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

לא נדיר לראות במערכות מורכבות, שבהן יש אובייקטים מקוונים רבים, את השימוש בשרשור אופציונלי שנכנס לשפה רק בשנת 2020 באופן רשמי.

אובייקט קבוע

אחד הדברים החשובים שצריך לזכור הוא שם מגדרים אובייקט קבוע, אין שם בעיה לשנות אותו. נכון, אי-אפשר לשנות את הסוג שלו (זאת אומרת, קבוע שמשמעותו במקום

אובייקט יכול הקבוע מספר), אבל אפשר להוסיף לו תכונות או לשנות תכונות קיימות, כלומר לשנות את השדות בתוך האובייקט ולא לעשות השמה לאובייקט אחר. מדובר בהבדלמשמעותי:

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

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

יש כמה הבדלים משמעותיים בין סוגי מידע פרימיטיביים לבין אובייקטים וסוגי מידע מורכבים יותר. אחד השינויים הכח מבלבלים הוא עניין הצבעה, 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 כשהיה. הנה דוגמה:

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

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

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

שינוי באובייקט, שאמור להיות במשתנה השני, משפייע על האובייקט המקורי! זה קורה כי כשמעתיקים אובייקט שנמצא במשתנה `objectA` למשתנה `objectB` לא מבצעים העתקה אלא הפניה, והבדל בינהןמשמעותי. הסיבה לכך היא שיקולי זיכרון. אם רוצים לבצע העתקה של אובייקט ולשנות אותו בלי שהוא ישפייע על האובייקט המקורי, צריך לעשות הליך שנקרא `clone` ואפשרי לביצוע באמצעות LOLAOT, ועל כך בפרק הבא.

תרגיל:

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:

יוצרים משתנה ומכו尼斯ים אליו אובייקט, באובייקט יש שלושה מפתחות: id, name ו price. כל אחד מהם קיבל ערך. ההגדרה של האובייקט נעשית באמצעות סוגרים מסולסים, שבתוכם שם המפתח, נקודתיים ואז 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}
```

בפועל יוצרים כאן אובייקט ריק ואז מכניסים את המפתחות שלו בזה אחר זה. הקריאה למפתח נעשית כמו משתנה – שם האובייקט, נקודת 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:

הפונקציה ארוכה, אך לא צריך להיבהל. היא מקבלת ארגומנט שקוראים לו `dayNumber`, והמטרה היא למצוא את מספר היום. לצורך כך משתמשים במשפט **switch case**. אם המספר הוא לא בין 1 ל-7 מופיעים הודעת שגיאת ומצביעים אובייקט ריק. אם המספר הוא בין 1 ל-7 מוצאים את היום ומכניסים אותו למשתנה `dayName`. את `dayName` ואת `dayNumber` מכניסים לאובייקט ומצביעים אותו. זה הכל.

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

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

תרגיל:

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

פתרון:

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

הסבר:

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

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

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

תרגיל:

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

פתרונות:

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

הסבר:

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

תרגיל:

צרו פונקציה בשם displayClass שמקבלת אובייקט בשם user שנראה כך:

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

הפונקציה מחזירה את ה-class שבתוך ה-`enrichedData`. אם הוא קיים. במידה ולא, מוחזר `undefined`.

פתרונות:

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

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

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

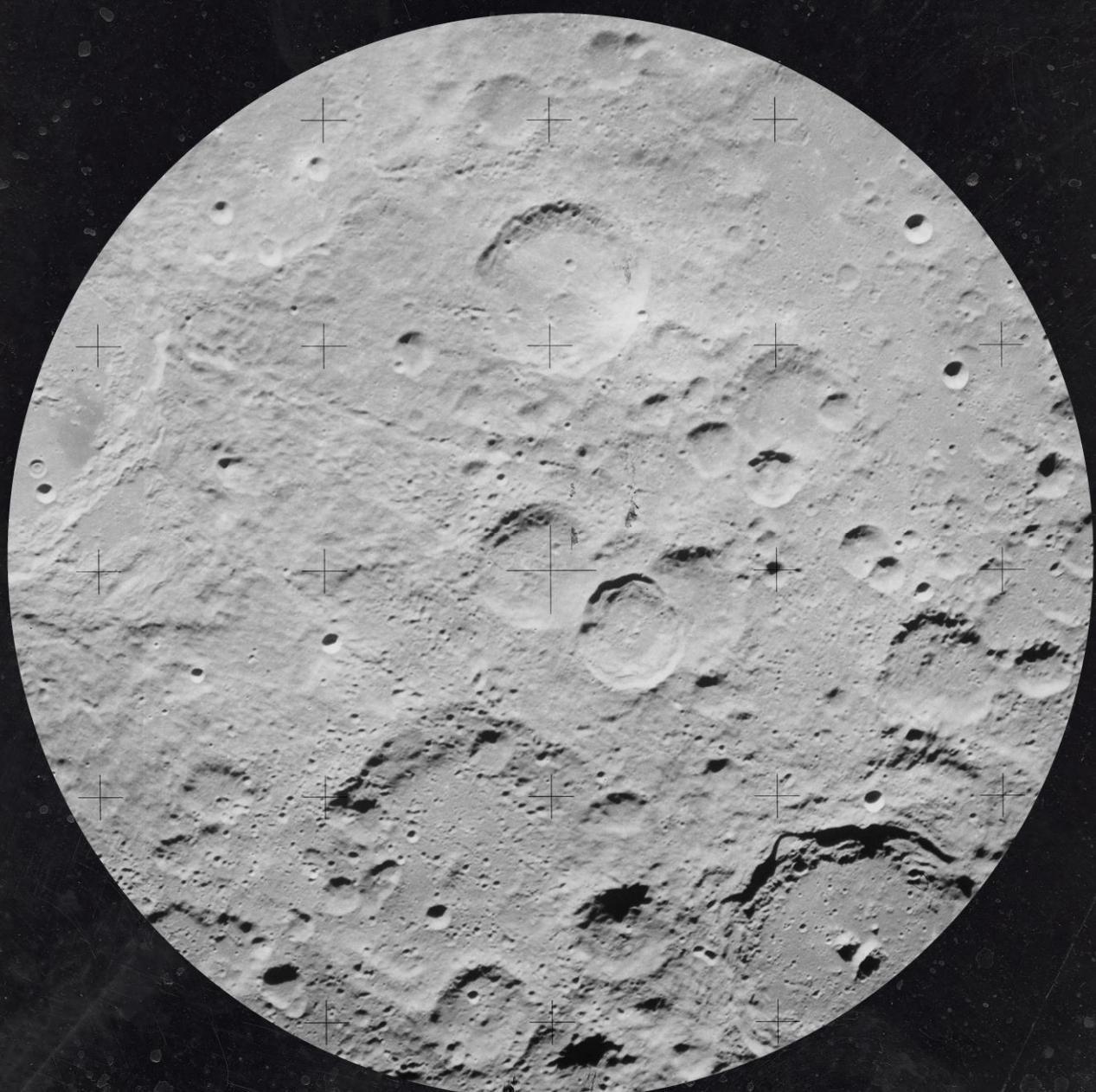
let result = displayClass(userObject1);
console.log(result); // 4
result = displayClass(userObject2);
console.log(result); // undefined
```

הסבר:

אני משתמש בשרשור אופציוני כדי להחזיר אובייקט מוקון. אם אני משתמש בשרשור רגיל (`userObject.enrichedData.class`) אני מקבל שגיאה באובייקט השני שבו אין את תכונת `enrichedData`. לפיכך אני אכנס את (`userObject.enrichedData?.class`)

פרק 10

מערנים



מערכים

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

זה דבר שתרמיז מבבל מתכנתים. כשאדרר על לולאות תבינו את ההיגיון של להתחיל מ-0, אבל חשוב לשנן ולזכור שבכל הנוגע למערכים תמיד סופרים מ-0.

יצירת המערך נעשית כך:

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

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

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

הדפסה של המערך תראה את האיבר שהכנסתי:

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

אם רוצים להכניס עוד איבר, אפשר להוסיף אותו כך:

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

אפשר להכניס 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 ונבר בשלהב הייצירה מכנים 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.

שימוש לב: פנינה לקבלת ערך מתוך מערך נעשית על ידי סוגרים מרובעים וגם האינדקס של האיבר.

אפשר כמובן להכניס גם אובייקטים לתוכה מערכים זהה איפלו 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`, ואז שולפים את האובייקט הראשון מהמערך ואיפלו לא את כל האובייקט הראשון, אלא מפתח נבחר מהאובייקט הראשון.

וכן, בדוק كيف שמערך יכול להכיל כל סוג מידע שהוא, כולל אובייקטים, מערך יכול להכיל גם מערכים. לערך שמכיל מערכים נוספים קוראים מערך דו-ממדית. הנה דוגמה לערך 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 "Ճחיפה", ואפשר למשמש אותה באמצעות שימוש בפונקציה מיוחדת לסוג המידע מערך בלבד, שנקראת `push`:

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

`push` היא פונקציה מיוחדת שעובדת אך ורק על סוג מידע שהם מערכים. ככל מדנו על סוג מידע, הריאתי כל מיני פעולות שאפשר לעשות על מחזוזות טקסט או על מספרים; ובכן, `push` היא פעולה שאפשר לעשות אך ורק על מערך. מכניםים לתוכה ארגומנט את המידע שרצים להכניס לערך בתור האיבר החדש והאחרון – בין שמדובר במספר ובין שבמחוזות טקסט, באובייקט, בערך וכו'. כל דבר יוכל כארוגומנט ב-`push`, ויוצר איבר חדש בסוף.

Ճחיפה מוסיפה איבר חדש לערך על גבי האיברים האחרים. אם היה איבר אחד במיקום 0,Ճחיפה תוסיף איבר חדש למיקום 1.

משינה היא בדיקת הפוך מדחיפה – היא מאפשרת למשוך את האיבר האחרון מהמערך ולקבל אותו כמשתנה. המשינה 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, והמערך שעבר שניוי (מוחציה), או נכוון יותר המחסנית, מתקצר באיבר אחד.

בעוד **push** מכניס איבר לסוף המערך, הפקציה **unshift** מכניסה איבר לתחילת המערך. כמו שתי הפקציות הקודומות, גם **unshift** ייחודה לערך ולא תעבור בסוגי מידע אחרים. הנה דוגמה לאייך **unshift** פועל:

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

בדוגמה יש מערך שבו שלושה איברים, הראשון הוא **a** והאחרון הוא **c**. אם מכניסים את מחוץテקסט z לתוך המערך באמצעות **unshift**, אז האיבר הראשון כבר לא יהיה **a** אלא **z**. האיבר האחרון עדין נותר **c**.

ולבסוף, שיליפת האיבר הראשון במערך נעשית באמצעות פונקציית **shift**:

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

בדוגמה זו רואים מערך שבו האיבר הראשון הוא מחוץテקסט **a** והאיבר האחרון הוא מחוץテקסט **c**. שימוש בפקציה **shift** מוחזר את האיבר הראשון. כרגע במערך האיבר הראשון הוא **a** והאיבר האחרון הוא עדין **c**. המערך קצר משלושה איברים לשניים.

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

| תיאור | פעולה |
|-----------------------------|---------|
| שלילת האיבר האחרון של המערך | pop |
| הכנסת איבר לתחילת המערך | unshift |
| שלילת האיבר הראשון של המערך | shift |
| הכנסת איבר לסוף המערך | push |

תכונה נוספת שקיימת בסוג 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 ומכוונים אליו שני ערכים בלבד – אחד במקומות 0-0 (שהוא המקום הראשון), אנו 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 לערך וקובעים את המיקום שלו, ג'אווהסקריפט תיצור את כל האיברים האחרים במערך עד המיקום שהוכנס. כל איבר כזה יהיה מסוג undefined (שימוש לב – לא null אלא undefined).

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

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

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

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

שימוש לב שכרוצים למחוק את הערך השלישי, צריך לצלין [2] כיוון שהמערך מתחילה מ-0.

הבעיה בדרך זו היא שהמערך נשאר בה גודל קבוע, והאיבר שעושים לו delete הופך undefined. כדי למחוק איבר במערך ולקצר אותו יש להשתמש ב-splice.

כמו המתודות הקודמות שלמדנו עלייהן, גם המתודה splice שייכת לערך בלבד. היא מקבלת כמה ארגומנטים. הראשון הוא מאייזה מקום במערך להתחיל והשני הוא כמה איברים למחוק.

נניח שיש מערך של אבות ורוזצים להוריד את משה, כי הוא לא אחד האבות המקוריים.

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

אם מורידים את משה באמצעות **delete**, יהיה undefined במערך וככינול יהיו "ארבעה אבות" והשיר "אחד מי יודע" ישתבש לגמרי. אז מה עושים? משתמשים ב-splice. מה המקום של האיבר שרצו למחוק (משה)? המוקם ה-2; זה אברהם, 1 זה יצחק ו-2 משה. כמה איברים רוצים למחוק? אחד. אין תיראה המחייבת לכך:

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

מערכות ומחרוזות טקסט

כדי להגדיל את השמחה ואת הבלבול הכללי, בג'אווהסקריפט מחרוזות טקסט חולקות לא מעט תכונות עם מערכיהם והן מתנהגות בחלק מהמקרים כמו מערכיהם שבהם האיבר הראשון הוא האות הראשונה, האיבר השני הוא האות השנייה וכן הלאה. שימושו לב למושג 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 כמערך ולשלוף את האות הראשונה באמצעות [0] ואפשר גם לשלוף את האות האחרון בדיק ניפוי ששולפים את האיבר האחרון במערך. האופרטורים **pop** ו**push** וגם **delete** לא יעבדו, אך כל שאר האופרטורים כן. כרגע אין לזה שימוש אופרטיבי, אבל כדאי לזכור את זה.

תרגילים:

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

פתרונות:

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

הסבר:

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

תרגיל:

למערך הקודם שיצרתם, הוסיפו את המספר 11 בסוף.

פתרונות:

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

הסבר:

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

תרגיל:

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

פתרונות:

```
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:

הדיחה של ערך כאיבר הראשון של המערך והזהה של כל האיברים נעשות באמצעות הפונקציה `unshift`, כמו `push` מקבלת את הערך שרצים שייהי ראשון במערך, במקרה זה 0.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר הראשון.

פתרון:

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

הסבר:

הפונקציה `shift`, שייחודה למערך, מסירה את האיבר הראשון ומציצה את כל שאר האיברים.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר החמישי.

פתרון:

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

הסבר:

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

```
myArray[4];
```

this-1 new

אפשר ליצור אובייקטים מufs באמצעות המילה השמורה **new**. בג'אווהסקריפט משתמשים המון בטכנית 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 הכנסה למשתנה. את הפונקציה זו מתחלים באמצעות **new**. שימוש לב שמשם מתחלים את הפונקציה. ברגע שימושים במילה **new**, הפונקציה מחזירה אובייקט.

הפונקציה שמתוכננת להיקרא באמצעות **new** נקראת **פונקציה בנאית**, ובאנגלית **constructor**. בתעשייה מקובל לכתוב שם של פונקציה בנאית באות גדולה בהתחלה, למשל **ObjMaker** ולא **obj**, 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 שיוצג כתוצאה מהפונקציה. **this** מצביע לאובייקט – כלומר 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 במבנה שהוגדרה קודם, אבל במקום פונקציה ריקה, יש בתחום 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` יקבל ביתוי 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 בנאית. הפונקציה הבנאית הכרזה על שם פרטי, על שם משפחה ועל תכונות נוספות של הלוקוט. אם תיצרו אל האובייקט באמצעות הקונסולה, תראו שהוא מכיל את כל הארגומנטים שהגדרתם באמצעות `this`:

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

משתמשים בדרך כלל בפונקציות בונות על מנת ליצור אובייקטים קבועים. למשל, אם רוצים ליצור אובייקטי 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 בנאית אחידה לאובייקט משתמש. משתמשים במילה השמורה `new` על מנת להחזיר למשתנה בכל הפעלה אובייקט אחר, בהתאם למשתנים שמכניסים לפעולה הבונה.

אפשר להוסיף לפועלה הבונה גם הגדירות של פונקציה שהן חלק אינהרנטי מהאובייקט שהפעולה הבונה יוצרת. לפעולות אלו קוראים "מתודות" (וביחיד "מתודה"). מדובר בפונקציה שמצוירת לאובייקט ואפשר להפעילה בקלות מהאובייקט שהפונקציה הבנאית 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 התכוונות, אבל היא פונקציה ולא מחרוזת טקסט. במקרה זהה היא מחזירה את השם המלא של המשתמש המורכב מתוכנת השם הפרטי, שם המשפחה ורוחם ביניהם.

אפשר להגדיר משתנים 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. בניגוד לחבריו, המשתנה הזה לא מוצמד ל-`this` אלא מוגדר ממש בתחום פונקציה. האם אפשר לגשת אליו מרוחק? לא. רק מה שמצוידים ל-`this` יהיה נגיש החוצה. מה שלא, מוגדר כפרטי.

לכן צריך לכתוב פונקציה נוספת:

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

פונקציה מהסוג הזה נקראת **פונקציית get**, כיון שהיא פונקציה שמשמשת לקבלה (באנגלית **(get)**) של משתנים פרטיים שאין גישה אחרת אליהם. כמו שיש גם **set** 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 בדיקות **set**-**get** אמ' אפשר פשוט לחושף את ה-**this**-**id** כמו שאר המשתנים? התשובה נעוצה במימוש. לעיתים רוצים לעשות ולידzieה. למשל, רוצים לוודא ש-**id** הוא תמיד מספר.

הדרך הכי טובה לעשות זאת היא לאנוסף 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 מכוונים אותו ל-`-p`, ומעכשיו המשנה `id` מכיל את הערך הזה. אם לא, מחזירים שגיאה ולא משנים את ה-`-p` המקורי.

از מה ההבדל בין הגדרה רגילה של אובייקט לפונקציה בנאית? בדרך כלל משתמשים בפונקציה בנאית על מנת להגדיר אובייקטים שיש להם תפקיד – ייחידת תוכנה שיש לה תכונות ומethods. דבר על כך בהמשך. בדרך כלל קוד מודרני של ג'אוּהַסְקְּרִיפְט ארוז באובייקטים שבבסיסם על פונקציה בנאית. ומה עם אובייקטים רגילים? בהם משתמשים בדרך כלל לאחסון מידע בלבד.

בהמשך תראו את התועלות שיש בפונקציה בנאית כאשר מפתחים מערכות מורכבות יותר. בינהמם, זכרו שברגע שאתה רואים `new`, זה סימן שתקבלו אובייקט חדש. הדבר על פונקציות בנאיות בפרק על אובייקטים מובנים בג'אוּהַסְקְּרִיפְט.

תרגיל:

צרו פונקציה בנאית לאובייקט מכונית מקבלת שם, צבע ונפח מנוע. לכל מכונית יש מספר זיהוי ייחודי המורכב מחיבור של דגם, צבע ונפח מנוע. למשל, אם שם היצרן של המכונית הוא opel, הצבע הוא white ונפח המנוע הוא 1,200, מספר הזיהוי שלו יהיה opelwhite1200. צרו לאובייקט מכונית פונקציה המחזיר את מספר הזיהוי. הדגם, הצבע ונפח המנוע הם נתונים שחשופים 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:

যוצרם פונקציה בנאית המקבלת שלושה ארגומנטים: שם, צבע ונפח מנוע. שלושת הארגומנטים האלו נחשפים החוצה באמצעות `this`. בתוך הפונקציה הבנאית יוצרים `modelNumber`, שמורכב משלושת הארגומנטים האלו. לא חושפים אותו החוצה באמצעות `this`. השלב הבא הוא ליצור פונקציה שתחזיר אותו. הפונקציה נחשפת החוצה באמצעות `this` ומחזירה את המשתנה הפרטיאי.

תרגיל:

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

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 שה-`modelNumber` מוגדר כמשתנה.

פרק 11

תבנית טקס



מבנה טקסט

מבנה טקסט מאפשר ליצור מחרוזות טקסט בклות משתנים שונים. למדנו בפרק על מחרוזות טקסט שאפשר לחבר בין מחרוזות טקסט. ככלمر אם קיימים משתנה שיש בו "Hello" וממשתנה שיש בו "World", יתקבל משווה זהה אם תחברו אותם:

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

אם רוצים רווח בין שני המשתנים, צריך להוסיף אותו. למשל משווה זהה:

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

אם לדוגמה יש מספר ורוצים להצמיד אליו את התו \$ (כמו במחירים) צריך לעשות משווה זהה:

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

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

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

הבעיה היא שההסמל \$ מושרבן מאד, וככל שחלף הזמן ותוכנות הג'אווהסקריפט הפתחו והפכו למורכבות יותר. היה אפשר למצוא בתוכנות מבוססות ג'אווהסקריפט תפלצות מהסוג הזה (למשל):

```
const user = {
  name: 'Ran',
  localtime: 'Morning',
};

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

```
console.log(welcomeString); // "Hello, Ran. How are you doing? Good Morning!"
```

כל החיבורים האלה מסורבים מאד. אבל מואוד. לפיכך נוצרה דרך להגדיר "תבנית" בג'אוּהַסְקָרִיפֶט 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 שעבוד בדיקן כמו גרש רגיל' או גרשיהם כפולים " בכל מה שנוגע לטקסט. כמובן, משה זהה:

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

בהחלט יעבד, ו-`myVar` ייחסב למחרוזת טקסט לגיטימית לכל דבר. אבל לגורש המסלולס יש יכולת שאין לגורשיים הרגילים, והוא להכיל התבנית. אם רוצים להכניס משתנה מסוימת בתוך מחרוזת הטקסט צריך להקיף אותה בדולר (\$) ובסוגרים מסולסים, והוא יוכל למכיר מחרוזת הטקסט בשלמותו. הנה דוגמה:

```
const user = {
  name: 'Ran',
  localtime: 'Morning',
};

let welcomeString = `Hello, ${user.name} How are you doing? Good
${user.localtime}!`;

console.log(welcomeString); // "Hello, Ran. How are you doing? Good
Morning!"
```

וכמובן, הדבר הזה:

```
let welcomeString = `Hello, ${user.name} How are you doing? Good
${user.localtime}!`;
```

זהה לחלוטין לדבר הזה:

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

מה נראה טוב יותר ומובן יותר? התשובה ברורה. התבניות מטפלות באופן נאה מאוד בRibivo שורות, ואילו מחרוזות טקסט רגילות לא מסוגלות להתמודד עם Ribivo 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 חשב שהתשובה ברורה. בשנים האחרונות יש מעבר חד-משמעותי אל שימוש בתבניות טקסט בג'אווהסקריפט, ואני ממליץ גם לכם להשתמש בהן.

תרגיל:

נתון מערך שיש בו בוקר וערב.

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

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

פתרונות:

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

הסבר:

מחרוזת הטקסט שאויהה מדפים מוקפת בגרש מסולסל ` מכל צד. הגרש מסמל התבנית טקסט. כיוון שרוצים את האיבר הראשון במערך, קלומר 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, ומחזירה 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:

cotבאים פונקציה רגילה שמקבלת שני ארגומנטים, amount ו-`currency`. שני הארגומנטים האלו נכנסים למחרוזת טקסט אחת שמקפת בגרש מסולסל ` מכל צד. הגרש זהה מצביע על הימצאותה של תבנית טקסט שכל משתחנה שנמצא בתוכה ומוקף בסוגרים מסולסים ו-`-$` – ערכו ייכנס למחרוזת הטקסט. מחרוזת הטקסט פה היא פשוטה:

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

כלומר, מחרוזת הטקסט היא המספר והערך צמודים. מוחזירים את מחרוזת הטקסט זהו. על מנת לבדוק אותה בודקים את הפונקציה – קוראים לה עם שני ארגומנטים, 30 וסמל המטבע של השקל (مוקף בגרשיים כיון שהוא מחרוזת טקסט), ובודקים את התוצאה. הדפסה תראה שצדקתם.

פרק 12

לולאות



לולאות

לולאות מאפשרות לעبور על האיברים של מערך או של אובייקט בклות רבה. יש לא מעט לולאות בג'אוּהַסְקְּרִיפְט. חלון מיועד למכרים וחילון לאובייקטים, וכל אחת מהן משמשת למטרה אחרת עם יתרונות וחסרונות משלها. לשם הבהרה – אני משתמש במונח "lolaea" לכל איטציה שהיא.

לולאות **for**

לולאות מאפשרות, בעצם, להריץ קוד שוב ושוב, כמה פעמים שרוצים. מספר הפעמים תלוי בתנאי מסוים. כך למשל אם רוציםSKU קוד ירוז עשר פעמים, יוצרים משתנה ומצביעים שהקוד הזה ירוז כל עוד המשתנה קטן מ-10 ובכל ריצה מעלים את המשתנה ב-1. כשהמשתנה הגיע ל-10 הריצה תיעצר.

איך עושים את זה? כה:

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

קטע הקוד הזה נקרא "**לולאה for**" והוא ירוז עשר פעמים בדיק, מ-0 ועד 9. הנה ננתח אותו: ביטוי ה-for מורכב מהמיליה השמורה **for** ומסוגרים עגולים שבתוכם ביטוי ה-for, המורכב משלושה חלקים:

| | | |
|-----------------|--------------------------------------|-----------------|
| <u>1</u> | <u>2</u> | <u>3</u> |
| 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 קבועים את שמו וMagnitude שלו יהיה שווה 0.

החלק השני עד متى תרוץ הלולאה. קבועים שהוא תרוץ כל עוד ↓ קטן מעשר. אפשר להשתמש פה בכל אופרטור השוואתי.

החלק השלישי הוא מה שקורה למונה בכל פעולה של הלולאה. במקרה הזה משתמש באופרטור שמוסיף 1. ככלור בכל פעם שהלולאה רצתה, ↓ יעלה ב-1.

בתוך הסוגרים המסוללים נמצא מה שקרה בכל פעם שהלולאה רצתה.

הבה נראה מה קורה בדרכה הראשונה:

דרכה ראשונה

```
מעלים את i באחד      עומד בתנאי      i = 0
↑                           ↑                           ↑
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 מדפיסה את i ומעלים את i ב-1. מגיעים לדרכה הבאה:

דרכה שנייה

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

פה כבר i שווה ל-1. נכון, כתוב פה $i = 0$, אבל זה תקף אך ורק לדרכה הראשונה של הלולאה. בדרכה השנייה תזכור שלא צריך לאתחל את i אלא רק להעלות אותו. כיוון ש-1 קטן מ-10 מרצים את מה שיש בתחום הסוגרים המסוללים ואחריו כן מעלים את i ב-1. בדרכה הבאה הוא יהיה שווה ל-2, ונמשיך לדרכה הבאה.

דרכה שלישית

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

פה i שווה ל-2, כיון שהיא עלה ב-1 בפעם הקודמת. 2 קטן מ-10 והתנאי עדיין תקף. הלולאה מרצים את מה שיש בתחום הסוגרים ומעלת את i ב-1. בדרכה הבאה הוא יהיה שווה ל-3 וכן הלאה. עד שנגיע לדרכה الأخيرة:

ריצה עשירית ואחרונה

מעלים את i באחד עומד בתנאי $i = 9$

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

הሪצה الأخيرة היא הריצה העשירית. i כבר שווה ל-9. מרים את מה שקרה בתחום הסוגרים המסולסים ומעלים את i ב-1. עכשיו הוא שווה ל-10. בריצה הבאה התנאי לא מתקיים, 10 לא קטן מ-10 והלולאה נעצרת.

שימוש לב: כל ריצה או סיבוב של הלולאה נקראים "**איטרציה**". זה המונח הסטנדרטי. מובן שאפשר לעשות לוולה שיורדת. שימוש לב:

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

אם מרים את הלולאה הזו מקבלים את ההדפסות הבאות בקונסולה:

```
"Iteration number 5"
"Iteration number 4"
"Iteration number 3"
"Iteration number 2"
"Iteration number 1"
```

שלושת חלקיו שלולאה הם:

1. אתחול מונה הלולאה והציבתו על 5.
2. קביעת התנאי שהלולאה תרוץ כל עוד i גדול מ-0.
3. בכל איטרציה i יורד ב-1.

הבה נראה מה קורה באיטרציה הראשונה:

ריצה ראשונה

באייטרציה הבאה יהיה שווה ל-4 עומד בתנאי

```

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

```

באייטרציה הראשונה מתחילה את **i** וקובעים אותו על 5. האם הוא עומד בתנאי? כן. 5 גדול מ-0. מדפיסים את מה שיש בתחום הסוגרים המסוללים ומורידים את זנב-1 באמצעות האופרטור **--**.

עוברים לאייטרציה השנייה:

ריצה שנייה

באייטרציה הבאה יהיה שווה ל-3 עומד בתנאי

```

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

```

באייטרציה השנייה זו כבר שווה ל-4. הוא עומד בתנאי, 4 גדול מ-0, לפיכך מה שכתוב בתחום הסוגרים המסוללים קורה (יש הדפסה בקונסולה) ו-1 מופחת-ב-1. הלאה!

ריצה שלישית

באייטרציה הבאה יהיה שווה ל-2 עומד בתנאי

```

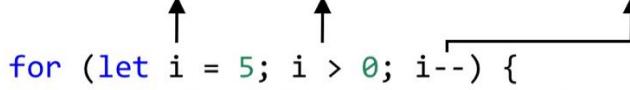
for (let i = 5; i > 0; i--) {
    console.log('Iteration number' + i);
}

```

באייטרציה השלישית זו שווה ל-3, עדין יש עמידה בתנאי. מרכיבים את מה שקרה בתחום הסוגרים המסוללים ואז מורידים את זנב-1.

ריצה רביעית

באייטרציה הבאה יהיה שווה ל-1 עומד בתנאי $i = 2$

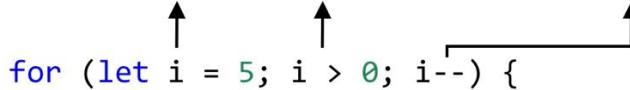


```
for (let i = 5; i > 0; i--) {
    console.log('Iteration number' + i);
}
```

באייטרציה הרביעית נ' שווה ל-2. עדין גדול מ-0. מתבצעת הריצה נוספת של מה שיש בסוגרים המסלולים והפחתה של 1 מ- i .

ריצה חמישית ואחרונה

באייטרציה הבאה יהיה שווה ל-0 עומד בתנאי $i = 1$



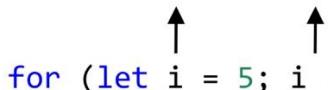
```
for (let i = 5; i > 0; i--) {
    console.log('Iteration number' + i);
}
```

באייטרציה החמישית והאחרונה נ' שווה ל-1. 1 גדול מ-0 ולפיכך מה שיש בסוגרים המסלולים ירוץ. וירד ב-1 ובሪיצה הבאה יהיה 0.

באייטרציה שלא תתקיים יש בדיקה אם נ' גדול מ-0, התנאי לא תתקיים. 0 לא גדול מ-0 ולפיכך האיטרציה לא תרוץ. הולאה נוצרת:

אייטרציה שלא תתקיים

לא עומד בתנאי $i = 0$



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

שימוש לב: מקובל מאוד לקרוא למשתנה של הולאה בשם `i` או `index`.

אמנם לולאות של הפחתה או הוספה הן הלוואות הקלאסיות, אבל אפשר באמצעות להתרען. הינה לדוגמה שמוסיפה ל-*i* בכל איטרציה 4 ותפעל כל עוד *i* קטן מ-12 או שווה לו:

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

הפלט של לולאה כזו יהיה:

```
"Iteration number 0"
"Iteration number 4"
"Iteration number 8"
"Iteration number 12"
```

תרגול טוב יהיה לנתחוב את הלוואה על נייר ולנסות להבין למה הפלט הוא זהה. הבה נתחיל באיטרציה הראשונה, מ-0. הסתם הוגדר כך שוויה ל-0. כיוון ש-0 קטן מ-12, ההדפסה בקונסולה מתבצעת ואז מעלים את *i* ב-4.

איטרציה מס' 1

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

באיטרציה השנייה, מ-4. מ-4 הוסיף 4. מ-8 הוסיף 4. מה שיש בתוך הסוגרים המסולסים יקרה. ו-8 עלה ב-4.

איטרציה מס' 2

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

באיטרציה השלישית, מ-8. 8 עדיין קטן מ-12 והפעולה בלולאה תתקיים. ו-8 עלה ב-4.

אייטרציה מס' 3

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

זו האיטרציה האחורונה 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. ברייצה הבאה נראה ש-16 גדול מ-12, ולפיכך הפעולה לא מתאפשרת לעולם. **בום!**

אייטרציה מס' 4 ואחרונה

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

דיברתי על **סקופים** בפרק על הפונקציות. גם בלולאות יש סkop. כל מה שකורה בתחום הסוגרים המסוללים נחשב לסקופ מסויל. גם אחת החוזקות של **let** לעומת **var** הישן יותר, יהיה נהוג בעבר. **let** שומר על סkop 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 אליו מבחוץ, וזה מילכך את הסkop 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 לשמור על הסkop הכללי שלכם נקי – ואתם רוצים, תאمينו לו שאתם רוצים – **אל תשתמשו ב-var**. יש בתקן החדש `let`, הבה נשתמש בו. צריך לזכור שהסקופ עובד גם פה, ול-`for` יש סkop משלו שמתפוגג בכל איטרציה.

לולאת `for` משחתקת יפה מאד עם מערכים. הבה נציגים. נניח שיש מערך שרוצים להדפס את כל תוכנו. איך אפשר לעשות את זה? כך:

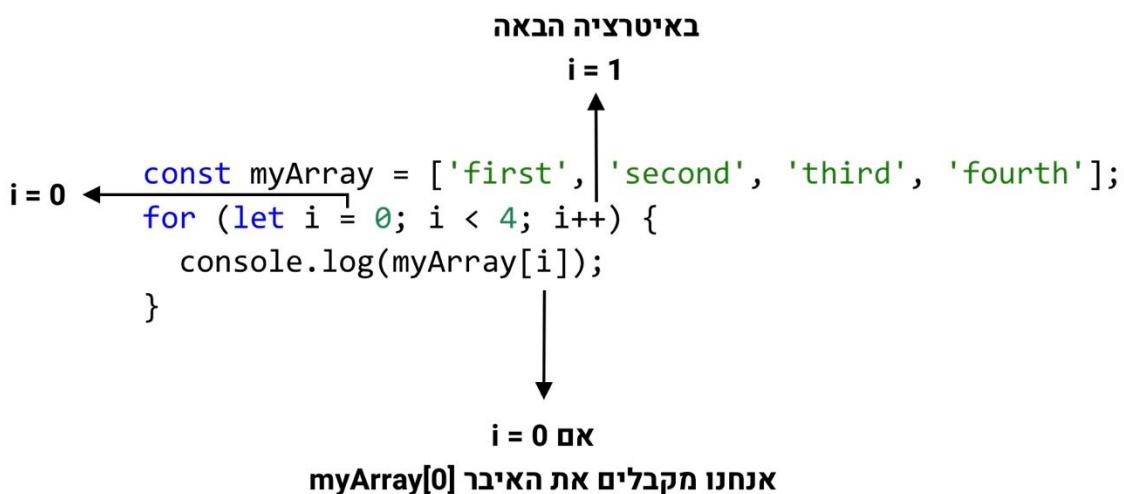
```
const myArray = ['first', 'second', 'third', 'fourth'];
console.log(myArray[0]);
console.log(myArray[1]);
console.log(myArray[2]);
console.log(myArray[3]);
```

אבל זה ארוך ומעצמן, במיוחד אם יש הרבה איברים. במקרה זה אפשר לכתוב לולאת `for` פשוטה:

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

כידוע, משתנה בכל איטרציה, מ-0 ועד 4 (3, 2, 1, 0). אפשר להציב אותו בתור מספר האיבר במרקם ולקבל אותו! הבה נבחן את האיטרציות:

1 איטרציה מס' 1



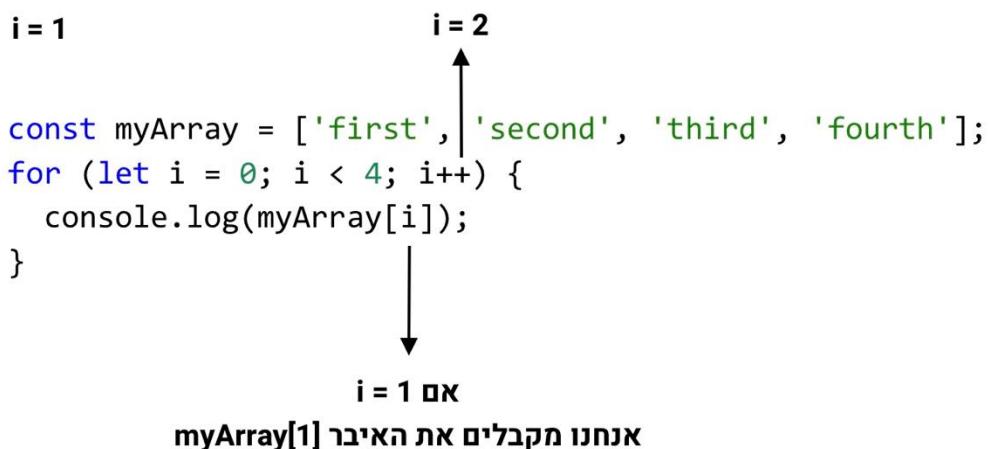
באייטרציה הראשונה ? מאותחל ושווה ל-0. מן הסתם 0 קטן מ-4, ומה שיש בתחום הסוגרים המסולסים מתקיים. כיוון ש-0= אפשר להשתמש בו. זה בדוק כמו כתוב:

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

פשוט במקרה הזה ? כבר קיים. לא צריך להגדיר אותו. מה שיודפס בקונסולה הוא first. עוביים לאייטרציה הבאה.

אייטרציה מס' 2

באייטרציה הבאה



באייטרציה הבאה ? כבר שווה ל-1. התנאי מתקיים כי 1 קטן מ-4, ומה שיש בתחום הסוגרים המסולסים רצ. אפשר להדפיס את האיבר הראשון (שהוא האיבר השני, כי במערכות מתחילה מ-0). עוביים לאייטרציה הבאה, שבה ? עולה ל-2.

אייטרציה מס' 3

באייטרציה הבאה

```
i = 2
      ↑
const myArray = ['first', 'second', 'third', 'fourth'];
for (let i = 0; i < 4; i++) {
    console.log(myArray[i]);
}
      ↓
אם i = 2
אנחנו מקבלים את האיבר myArray[2]
```

גם פה, כמו באיטרציות הקודמות, התנאי מתקיים. 2 קטן מ-4. אפשר להשתמש בו, שכרגע הוא 2, להדפיס את האיבר השלישי במערך ולבור לאייטרציה הבאה, שבה זה עולה שוב, הפעם 3-4.

אייטרציה מס' 4

באייטרציה הבאה

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

באייטרציה الأخيرة שווה ל-3 ועדיין עומד בתנאי. אפשר לקחת אותה ולהדפיס את האיבר הרביעי בסדרה. באיטרציה הבאה שווה ל-4, אבל זה כבר לא מעניין כי 4 לא קטן מ-4 והתנאי לא מתקיים. זו הריצה الأخيرة של הלולאה הזאת והמערך הודפס.

הבעיה בגישה זו היא שמניחים שגודל המערך הוא 4. אבל מה אם לא יודעים מה גודל המערך? זכריהם שאפשר לחלץ את גודל המערך באמצעות שימוש בתוכונה המיוחדת `length`? אפשר להשתמש בה בולולאה:

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

למערך `myArray` הוי 4, 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 שהיא לא תעצור לעולם. למה? כי היא תתקיים כל עוד i גדול מ-0, ו- i גדול ב-1 כל הזמן ולעולם לא יהיה קטן מ-0. הלולאה זו, שלא تستופט לעולם, נקראת לולאה אינסופית. בדרך כלל, דברים כאלה יקריםו אחרי המון איטרציות. אם מרכיבים את זה בדף, מתישהו הלשונית תהיה לא יציבה ותקרוס. כשכתבם לולאה (ולא משנה מאייה סוג) והדף קופא, סביר להניח שזאת הסיבה.

לולאות while

לולאה נוספת היא לולאת **while**. היא פשוטה יותר מlolאת **for** ומכללה תנאי אחד בלבד: עוד הוא נכון הלולאה מתקיים. למשל:

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

כל עוד התנאי מתקיים, הלולאה רצה.

```
let i = 0;
                    ↑
                    →   כל עוד התנאי הזה שווה ל-true
while (i < 10) {           ←   הלולאה תרוץ
    console.log(i);
    i++;
}
```

בלולאות כאלה, כמובן, חייבים להגיד את מהו לולאה, אחרת בכל ריצה של הלולאה יתאפשר משתנה ה- i .

התנאי יכול להיות כל תנאי שהוא, כולל תנאים מורכבים, ובגלל זה הלולאה זו 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 **ח'יובי** – קטע מ-100. ברגע שאחד משנייהם לא נכון, היא תעצור.>Kצת מורכב ומבלבל, אבל כדאי לזכור את זה. אחד השימושים שלה הוא אם (מסיבה מסוימת) רוצים ליצור לולאה אינסופית, אז כתבים ממשו זהה:

```
while (true) {
```

אבל צריכה להיות לכם סיבה טובה מאוד ליצור לולאה אינסופית.

לולאת do while

עוד סוג של לולאת while הוא לולאת **do**. סוג הlolאה זהה מאוד לא שכח אבל כדאי להכיר אותו. lolאה זו ייחודית כי ברגע לולאת for ולולאת while, היא תמיד מבצעת פעולה אחת ואז בודקת את התנאי, במקום לבדוק את התנאי ואז לrox. כלומר, תמיד הlolאה רצה לפחות פעם אחת.

הינה דוגמה:

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

יש כאן מילה שמורה, ספ, ומיד אחריה סוגרים מסולסלים. מה שיש בתוכם תמיד יקרה, ורק אחר כך יגיע ה-while שבודק את התנאי. כאן, למשל, הlolאה תתקיים כל עוד `i` הוא שלילי. לפניה התנאי קבועים את `i` על 100. לפי כל הlolאות שלמדנו זה אומר שהlolאה לא תרוץ לעולם, אבל כיוון שימושם ב-`while do`, הlolאה תרוץ לפחות פעם אחת.

foreach

lolאות שלמדנו עד כה הן לולאות שעובdot עם תנאים, אבל בעולם הג'אווהסקריפט מקובלות יותר לולאות שעובdot עם אובייקטים או עם מערכim באופן שלם יותר. למה צריך את זה? באחד התרגילים בפרק על מערכim 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 while עלולים להסתבך כי אי-אפשר להיות בטוחים שהאיבר אכן מוגדר. במקום להתחיל לשבור את הראש על גודל המערך ועל התאמתו ללולאת for, שלא לדבר על מה שיקרה אם יש איברים ריקים, יש דרכיים אלגנטיות יותר לעبور על כל האיברים בו. הדרך הראשונה והידועה מכולן היא שימוש ב-`foreach`.

מדובר במתודה שקיימת אך וرك במבנה נתונים מסווג מערך (לא אובייקט) ומתקבל 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 לנו מערך בשם `myArray` ובו שלושה איברים. האיבר במיקום 0 הוא מחרוזת טקסט `first`. האיבר במיקום 1 הוא מחרוזת טקסט `second` והאיבר האחרון, במיקום 2, הוא מחרוזת טקסט `third`. כיוון שמדובר במערך, אפשר להציג לו את מתודות `:forEach`

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

המתודה הזו היא פונקציה שצריכה לקבל ארגומנט. מה הארגומנט? פונקציה נוספת, שבדרך כלל היא אנונימית. על פונקציות חוץ אנונימיות למדנו בפרק על הפונקציות. הפונקציה שאותה מעביריםפה תורוץ בכל איבר של המערך. הארגומנטים שלה הם האיבר עצמו והמספר שלו:

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

זה כמו לכתוב משהו זהה:

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

הקוד שלעיל זהה כמעט לחלוtin לקוד של פונקציה אנונימית, אבל בה יוצרים עוד פונקציה ו"מלככים" את הסkop. מקובל מאוד להשתמש בפונקציה אנונימית זהה מה שעשושים כאן. הפונקציה האונימית רצתה על כל איבר. מאיפה באים הארגומנטים `value` ו-`key` בלי שקראו להם? זה הכוח של `forEach`, היכולת אוטם שם.

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

הבה נבחן את הרצאה. 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 תתבללו מהסינטקס המזרע עם החץ, בסופו של דבר זו פונקציה. מה שיש בתוך הסוגרים המסלולים רץ וرك האריגומנטים מתחלפים. ברגע הראשונה של `forEach`, תור האיבר הראשון להיכנס לפונקציה האנונימית. יש `value` שהוא האיבר של המערך ויש `key` המיקום של האיבר במערך. למה `0`? כי מערך תמיד מתחיל מ-`0`.

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

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

second 1

הפונקציה רצתה שוב, אבל ה-`key` וה-`value` שלה שונים. הפעם אלו ה-`key` וה-`value` של האיבר השני. ה-`value` הוא מה שיש בתוך האיבר השני במערך, ובמקרה זה - מחוץ לטקסט. ה-`key` הוא המיקום של האיבר השני במערך. למה `1`? כי מערכים מתחילה מ-`0`.

בפעם השלישית והאחרונה, האיבר השלישי ייכנס לפונקציה האנונימית:

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

The diagram illustrates the execution of the `forEach` loop. The array `myArray` contains three elements: 'first', 'second', and 'third'. The loop iterates over these elements, with `value` representing the current element and `key` representing its index. In the final iteration, `value` is 'third' and `key` is 2. The red arrows show the mapping from the array elements to the corresponding placeholders in the `console.log` statement.

שמות הארגומנטים נקבעו בהתאם 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 אין-אפשר לשבור את הלולאה. היא תרוץ לכל אורך איברי המערך. למה רצימ לשבור את הלולאה? בגלל אלף ואחת סיבות. בדרך כלל אם מחפשים משהו ומוצאים אותו, אין צורך בהרצת כל הלולאה. בדיק בשביל זה יש את לולאת `for of`.

הפונקציה שאנו מעבירים יכולה להיות גם לא אונומית. אבל מקובל להעביר אונומית.

lolat 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. משוו בסגנון זהה:

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

לא צריך להיבהל. מדובר במערך שבמוקם מספרים או מחרוזות טקסט יש בו אובייקטים. לכל אובייקט יש שתי תכונות – `name` ו-`price`. `name` מכיל מחרוזת טקסט ו-`price` מכיל מספר. לולאת `for 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`, שאotta הכרנו בollowat. `for`. אך במקום הסינטקס המוכר יותר, יש הגדרת משתנה שהוא בעצם האיבר התוון במערך, המילה השמורה `of` ושם המערך. בתוך הסוגרים המסלולים יירוץ בכל פעם האיבר התוון 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}`);
}

```

הבה נראה את האיטרציה הראשונה:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
for (let order of orderArray) {
  console.log(`the price of ${order.name} is ${order.price}`);
}
{ name: 'pen', price: 11 },

```

באייטרציה הראשונה, האיבר הראשון הוא שנכנס לתוך המשתנה `order`. יש לו תכונה בשם `name` ותכונה בשם `price`. כיוון שאנו מדפיסים אותו, מה שרואים הוא:

```
// "The price of pen is 11."
```

בהרצתה השנייה, האיבר השני נכנס אל תוך המשתנה זהה ומה שיש בסוגרים המסמלים
כך 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
שלפנינו.

עכשו נverb לשבירת לולאה. בואו נניח שהדרישה היא להציג את המוצר הראשון
שהמחיר שלו נמוך מ-10. לא את כל המוצרים אלא את המוצר הראשון. דרישת צו יכולת
כמובן להיות אמיתית לחלווטין. 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) {
    answer = order;
    break;
}
}
```

וש הדפסה של:

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

אך התנאי אינו מופעל כיון שהמחיר הוא 11 והתנאי הוא מחיר קטן מ-10.

באיורציה השניה:

```
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;
    הלואה נשברת
    break;
  }
}
```

וש הדפסה:

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

כיוון שהתנאי מתקיים, המשפט `break` מופעל והלואה לא עוברת לאיורציה השלישית.
זה היתרון הגדול של `for of` – שהוא ממזגת את היכולת של `while` מבחינות שבירה ואת האלגוריות של `.forEach`.

לולאות map

פעמים רבים רוצים לשנות את המערך עצמו. למשל, הבה נניח שלקוח שמנג'ע מישראל רואה את המחיר ורוצה להציג לו אותו ב שקלים, כלומר להכפיל את המחיר פי ארבעה. איך עושים את זה? אפשר להשתמש בלולאת `for`, בלולאת `while`, בלולאת `forEach` או בלולאת `of` – אך אם רוצים לשנות את האובייקטים בתוך המערך מקובל מאד להשתמש ב-`map`, שמאחורי הקulisם לוקחת את האובייקטים במערך המקורי ומשנה אותם ומחזירה את המערך המקורי עם המשתנים החדשניים. לולאת `map` מיועדת לשינוי אובייקטים מערכים או במערכות והיא פועלת באופן דומה ל-`forEach`, אלא שהפונקציה האנונימית יכולה להחזיר את האובייקט החדש שייכנס במקום האובייקט המקורי, כלומר לעשות לו מותציה. 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 ללולאת `forEach` שעליה כבר למדנו. ה-`value` שהפונקציה הפנית מקבלת הוא האיבר שיש במערך. אבל אפשר לראות שכן הפונקציה מחזירה משהו. מה היא מחזירה? את האובייקט שנמצא באיבר החדש ש עבר שינוי. האובייקט החדש ייכנס במקום האיבר הקודם, כולל כל השינויים שהכנסתם אליו:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value; → { name: 'pen', price: 44 },
});

```

נכנס למקומ הראשון
במערך המקורי

אפשר להחזיר מה שרצויים, אפילו מחירוזת או אובייקט מורכב יותר. כל מה שמננישים
ויכנס במקום האיבר הראשון של המערך. באירועה השניה מושנים את האיבר השני, ומה
שמוחזרים ונכנס במקום האיבר השני. הנה:

```

let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 }, ←
  { name: 'TV', price: 2345 }
];
orderArray.map((value) => {
  let newPrice = value.price * 4;
  value.price = newPrice;
  return value; → { name: 'pencil', price: 20 },
});

```

נכнес למקומ השני
במערך המקורי

ולסימן, באיתרציה האחרונה מקבלים את האיבר השלישי ומה שמחזירים יוכנס למקום השלישי במערך המקורי:

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

נכns למקומ השליישי
והאחרון במערך
ה המקורי

```
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 שהיה פעם אלא האובייקטים החדשניים.

אם אנו משנים מערך פשוט קיימ, הערכים לא השתנו ויש להורות לפונקציית `map` להחזיר את המערך שהשתנה לערך אחר:

```
let orderArray = [11, 12, 13, 14];
let newArray = [];
newArray = orderArray.map((value) => {
  let newPrice = value * 4;
  return newPrice;
});
```

`console.log(orderArray); // [11, 12, 13, 14]`

`console.log(newArray); // [44, 48, 52, 56]`

lolat filter

הלוֹלָאַה האַחֲרָוָנָה הִיא לוֹלָאַה שְׁמוֹצִיאָה אַיְבָּרִים מִהְמָעָרָךְ. בָּעוֹד **map** רַק מְשֻׁנָּה אַיְבָּרִים, בָּאמְצָעָות לוֹלָאַה **filter** אָפָּשָׂר לְצִמְצָם אֶת המָעָרָךְ. הַבָּה נִסְתְּכֵל עַל המָעָרָךְ הַזֶּה, לְמַשְׁלַח:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
```

נִנֵּיח שְׁמַבְקָשִׁים לְהַצִּיג לְלִקְוֹחַ רַק דְּבָרִים שְׁהַמְּחֵיר שְׁלָהֶם גְּבוֹה מ-7. אָפָּשָׂר לְעַשְׂתָּה מְשֻׁהָוָה כֵּזה:

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

כְּלֹוֹמֶר, לְקַחַת אֶת המָעָרָךְ וְלַעֲבֹר עַלְיוֹ עַם לוֹלָאַה. בְּכָל אִיטְרָצִיה (מַעַבָּר) של הַלוֹלָאַה בְּוֹדֶקִים אֶת המְחֵיר של האַיְבָּר הַנוֹּכָחִי. אִם הוּא קָטָן מ-7 או שׁוֹהָה לוֹ, מַוחְקִים אֹתוֹ. האַיְבָּר הַרְאָשׁוֹן, לְמַשֵּׁל, הוּא `pen`, שָׁה-`price` שְׁלָוּ הוּא 11. הַתְּנָאי לְאִתְקִיּוּם כִּי `value.price` כִּי-`7`. בָּמָקָרָה 11 וְהוּא לֹא קָטָן מ-7 או שׁוֹהָה לוֹ. האַיְבָּר הַשְׁנִי הוּא `pencil` וְה-`price` שְׁלָוּ הוּא 5. בָּמָקָרָה הַזֶּה הַתְּנָאי מַתְקִיּוּם. כִּיּוֹן שְׁלֹוֹלָאַת ה-`forEach` נוֹתָנָת גַּם אֶת ה-`key`, הַלֹּא הוּא מְסֻפָּר האַיְבָּר, שְׁבָמָקָרָה שְׁלָוּ האַיְבָּר הַשְׁנִי הוּא 1 (זָכְרִים שְׁמַעָרָךְ מִתְחִיל מ-0, נִכּוֹן?), אָפָּשָׂר לְהַשְׁתָּמֵשׁ בְּאוֹפְרָטוֹר `delete`, שְׁעַלְיוֹ לִמְדָנוּ בְּפָרָק עַל מִעָרָכִים, כִּדי לְמַחְזֹק אֶת האַיְבָּר הַזֶּה, וְכֵךְ הַלָּאַה. מֵה שְׁנָקְבָּל בְּסוֹף הוּא מָעָרָךְ עַמְשִׁי `undefined`, הַיְכַן שְׁהִיוּ כָּל האַיְבָּרִים שְׁמַחְיִים נִמּוֹךְ מ-7.

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

זֶה קָצֶת מִבָּאָס, וְאָפָּשָׂר לְעַקְוֹף אֶת זה בְּכָל מִינִי דְרָכִים או בָּאמְצָעָות `filter`. מַדוּבָּר בְּלוֹלָאַה שְׁזָהָה ל-`map` אֵיךְ הַפּוֹנְקִזְצִיה שְׁרָצָה בָּה מְחַזֵּרָה רַק `true` או `false`. אִם הִיא מְחַזֵּרָה `true`

האיבר נשאר במערך. אם היא מחזירה `false` האיבר מתפוגג מהמערך וכך לא יהיה שם. ברגע לולאת `map`, הלולאה הזאת לא משנה את המערך המקורי וצריך להחזיר את התוצאה למשתנה אחר.

הינה דוגמה:

```
const writingArray = [
    { name: 'pen', price: 11 },
    { name: 'pencil', price: 5 },
    { name: 'paper', price: 10 },
    { name: 'brush', price: 12 },
    { name: 'ink', price: 6 },
];
let filteredArray = writingArray.filter((value) => {
    if (value.price <= 7) {
        return false;
    } else {
        return true;
    }
});
```

הו `writingArray` הוא מערך המכיל אובייקטים של מוצרים כתיבה. לכל אחד מהם יש שם ומחר. כדי לסנן את כל האיברים שהמוצרים שלהם נמוך מ-7, משתמשים בפונקציה ייחודית לערך שנראית `filter`. הפונקציה הזאת מקבלת פונקציה אנוונימית שאזורה כתובים כפונקציה חז. אני רק מזכיר - פונקציית חז היא דרך מסוימת לכתוב פונקציה אנוונימית (כלומר פונקציה שאין לה שם אלא עותרת כארוגומנט). הפונקציה האנוונימית מחזירה `true` או `false`. אם היא מחזירה `true` האיבר נשאר במערך. אם היא מחזירה `false` הוא עף מהמערך. פונקציית `filter` מוחזירה את המערך החדש.

הבה ננסה לפרק את זה. בסבב הראשון, מי שנכנס לטור הבדיקה של ה-filter הוא האיבר הראשון:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) { price = 11
    return false;
  } else {
    return true; הfonkzia mchzira true
  }
}); הaiver zoche lahisar b'murak
```

באייטרציה השנייה מגיע תור האיבר השני:

```
let writingArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'paper', price: 10 },
  { name: 'brush', price: 12 },
  { name: 'ink', price: 6 },
];
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) { price = 5
    return false; הfonkzia mchzira false
  } else {
    return true; הaiver hza la zoche lahicnes laiiver ha'adash
  }
});
```

הוא כבר לא זוכה להיכנס, כיון שהמחיר שלו הוא 5 והתנאי הוא שכל מה שקטן מ-7 או שווה לו מקבל `false`. וכך הלאה. כמו כל לולאה אחרת, הלולאה הזאת עוברת על כל המערך.

הפונקציה שהעבכנו מחזירה true או false לגבי כל איבר. האיברים שמקבלים true זוכים להיכנס לערך החדש, ואלו שמקבלים false – לא. בסופו של תהליך חייבים להחזיר את מה ש-filter מחזירה אל משתנה חדש. המערך המקורי נשמר כמו שהוא:

חייבים להחזיר את התוצאה אל מערך חדש



```
let filterArray = writingArray.filter((value) => {
  if (value.price <= 7) {
    return false;
  } else {
    return true;
}
});
```

lolāt sort

lolāt sort נוספת משמשת למילון מערכים. lolāt הזו לא משנה איברים ולא את גודל המערך, אלא פשוט מסדרת את האיברים בערך. בנגדו לאייטציות אחרות, לא חייבים להעביר פונקציה אונומית שתעשה את הסדר. כבירות מחדל האיטציה הזו מסדרת לפי סדר הא'-ב' – היא מミירה את הערכים לחרוזות טקסט ועורכת השווואה ביניהם, וכן מבצעת את הסדר. הינה דוגמה פשוטה מאוד שמדגימה את זה:

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

ובן שהענינים מתחילה אם בערך יש אובייקטים ולא רק מספרים או מחרוזות טקסט, אבל לא נכנסה את אייטציית sort במלואה בספר זהה.

lolāt על אובייקטים

כל lolāt שלמדו עד כה משחיקות יפה עם מערכים ובאמצעותן אפשר לעבור על כל אובייקט בערך. אך נשאלת השאלה – איך לעבור על אובייקט? הכוונה למשהו בסגנון זהה:

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

מדובר באובייקט שיש בו נתונים על משתמש מסוים. נניח שרוצים להדפיס את כל נתונים האובייקט (בלי לדעת מה השמות התכונות שלו). איך עושים את זה? lolāt for או lolāt while לא יעזור ולולאות forEach או of מיועדות למערכים בלבד ויניבו שגיאות אם תנסו להשתמש בהן. יש שתי שיטות לעבור על אובייקטים. הראשונה "לפי הספר", שמעטם משתמשים בה, והשנייה הפופולרית יותר.

לולאות **for in**

מדובר בלולאה שעובדת על כל תכונות האובייקט בדומה ל-**for**. כך היא נראה:

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

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

/*
"key: userId, value: 12"
"key: userName, value: barzik"
"key: age, value: 40"
*/
```

לולאת **the-in for** מכילה כמה חלקים. ראשית המילה השמורה **for**, ומיד אחריה סוגרים עגולים. בסוגרים העגולים מגדירים משתנה שהוא המפתח. בתוך הלולאה יקבל המשתנה זה את שם המפתח התורן. אחר כך מכניסים את המילה השמורה ח'ו ואת המשתנה שבתוכו נמצא האובייקט.

בתוך הסוגרים המסורלים מתחוללת האיתרציה של כל מפתח. ברגע שיש את המפתח **key**, אפשר לgesht אל הערך שלו באמצעות סוגרים מרובעים כפי שמדנו בפרק על האובייקטים. המפתח של האובייקט הוא המשתנה **key** והערך הוא:

`userObject[key]`

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

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

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

באייטרציה הראשונה שמתרכשת, key הוא המפתח הראשון שיש באובייקט, הלווא הוא מחזורת הטקסט userId. באמצעות המפתח אפשר לחוץ את הערך. איך? למדנו בפרק על אובייקטים שאם המפתח נמצא בשם של משתנה אפשר לחוץ אותו באמצעות שימוש בסוגרים מרובעים, וזה בדוק מה שעושים. את המפתח ואת הערך מדפיסים באמצעות שימוש בתבנית טקסט פשוטה, גם עלייה למדנו:

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

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

userObject[userId] = 12
```

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

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

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

userObject[userName] = 'barzik'
```

באייטרציה الأخيرة, key הוא age.

```

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

for (let key of userObject) { key = age
  console.log(`key: ${key}, value: ${userObject[key]}`);
}
  userObject[age] = 40

```

הלוואה זו ניתנת לשבירה על ידי `.break`.

בסק הכל פשטונח ונהemd, נכון? אבל האמת היא שנדיר למצוא מתכנת שמשתמש בשיטה זו, מפני שלולאת `in` מביאה נתונים גם מאובייקטים אחרים שהם **פרוטוטייפ** – אבל טיפוס – של האובייקט שלנו. כרגע זה נשמע סינית ועודין לא דיברנו על זה, אבל על קצה המזלג אזכיר שבדרך כלל אובייקטים לא מגעים כלוח חלק והם העתק של אובייקטים אחרים שיש להם מפתחות נוספים. לרוב לא מעוניינים בפתחות של אובייקטים האחרים, ולפיכך לולאת `in` עלולה להיות הרת אסון, וזה הסיבה שבkowski משתמשים בה. אם כבר משתמשים בה, מקובל מאד לוודא שהתמונה שבודקים היא התמונה של האובייקט, באמצעות שימוש בפונקציה `hasOwnProperty` שנקראת `hasOwnProperty`, באופן הבא:

```

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

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

```

ואם כל מה שכתבתי נשמע לכם כמו סינית זה בסדר. זכרו שלולאת `in` לשימוש באובייקטים היא בעייתית ונדר שמשתמשים בה.

Object.keys

הדרך השנייה והפופולרית ביותר היא להשתמש בפונקציה שנקראת `Object.keys` לחילוץ כל המפתחות של האובייקט כערך, ואז לעבור על המערך זהה באיזו לולאה שורציהם, כי זה מערך. אין ניגשים לערך באובייקט? כמו בולולאת `for`; ברגע שיש את המפתח יש גם את הערך.

שימוש לב דוגמה הבאה:

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

const userObjectKeys = Object.keys(userObject); // ["userId",
"userName", "age"]

userObjectKeys.forEach((key) => {
    console.log(`key: ${key}, value: ${userObject[key]}`);
});

/*
"key: userId, value: 12"
"key: userName, value: barzik"
"key: age, value: 40"
```

מה קורה פה? יש כאן את אותו אובייקט מהדוגמה הקודמת. באמצעות שימוש בפונקציית `Object.keys` מקבלים את כל המפתחות כערך שנכנס ל-`userObject`, ועליו עושים

`forEach`. כיוון שמדובר בערך, אפשר לעשות עליו `forEach`. נחדר ואלגנטי.

נסכם בטבלה המסבירה על כל סוג הולולאות:

| חסרוןות | יתרונות | שם לולאה |
|---|---|----------------------------|
| lolalot l'maranim | | |
| לא תנאים מורכבים צריך לדעת את גודל המערך קל להיכנס לlolאה אינסופית מגיעים גם לאיברי <code>undefined</code> | פשוטה למימוש ניתנת לשבירה באמצעות <code>break</code> | lolalat <code>for</code> |
| צריך לדעת את גודל המערך קל להיכנס לlolאה אינסופית מגיעים גם לאיברי <code>undefined</code> | פשוטה למימוש ניתנת לשבירה באמצעות <code>break</code> אפשר לכתוב תנאים מורכבים | lolalat <code>while</code> |

| | | |
|--|--|----------------------|
| צריך לכתוב פונקציית חצ'ק אין ניתנת לשבירה | לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים | לולאת forEach |
| מקבלים רק את הערך של אברי המערך וצריכים לחלץ את המפתח בלבד | פשוטה למימוש ניתנת לשבירה באמצעות break לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים | לולאת of |
| לא מוחקת איברים מהמערך לא משנה את המערך וויצרת ממנו מערך חדש באותו גודל. אפשר לשנות את האיברים בקבוקות. | לולאה שעוברת על המערך מוחקת איברים | לולאת map |
| לולאות להמרת מערכים | | |
| צריכים להחזיר את התוצאה למשתנה אחר לא משנה את המערך המקורי | מוחקת איברים | לולאת filter |
| לולאות לאובייקטים | | |
| מקבלים רק את המפתח של האובייקט וצריכים לחלץ את הערך בלבד חייבים להשתמש ב- hasOwnProperty | פשוטה למימוש ניתנת לשבירה באמצעות break לא צריך לדעת את גודל האובייקט אין לולאה אינסופית לא נכנסים לאיברים ריקים | לולאת in for |
| מקבלים רק את המפתח של האובייקט וצריכים לחלץ את הערך בלבד אפשר להשתמש בלולאות של המערך ואז מקבלים את כל היתרונות של הלולאה שבחורתם מקבלים את כל החסורונות של הלולאה שבחורתם | אפשר להשתמש בלולאות של המערך ואז מקבלים את כל היתרונות של הלולאה שבחורתם | שימוש ב- Object.keys |

תרגיל:

באמצעות לולאות `for`, הדפיסו שלוש פעמים על המסך "I know how to use for loop".

פתרונות:

```
for (let i = 0; i < 3; i++) {
    console.log('I know how to use for loop');
}
```

הסבר:

זו לולאה `for` פשוטה שמתחליה כאשר `i` שווה ל-0. היא אמורה להימשך כל עוד `i` קטן מ-3. באיטרציה הראשונה כאמור `i` שווה ל-0. בשנייה הוא שווה ל-1, בשלישית הוא שווה ל-2 וברביעית 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 המדפיסת את שמות הלקוחות.

פתרונות:

```
const customersArray = ['Moshe', 'Yaakov', 'Yossi', 'Baruch', 'Yael'];
for (let i = 0; i < customersArray.length; i++) {
    console.log(customersArray[i]);
}
```

הסבר:

משתמשים בלולאת `for`. המונה הוא `i` ששויה ל-0 והתנאי הוא שהולאה תרוץ כל עוד המספר קטן מאורך המערך, במקרה זה 5. משתמשים במונה `i` שזמן על מנת להדפיס את האיבר המתאים בערך. למשל, באיטרציה הראשונה `i` שווה ל-0. התנאי מתקיים כי `i` קטן מ-5 (אורך המערך). משתמשים בו על מנת להדפיס את האיבר שנמצא במקומות 0. באיטרציה השנייה, `i` שווה ל-1, התנאי מתקיים כי `i` קטן מ-5 (אורך המערך) ומשתמשים בו כדי להדפיס את האיבר שנמצא במקומות 1, וכך הלאה.

תרגיל:

נתון מערך של אובייקטי הזמנות:

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

הציגו את סכום הזמנות במערך.

פתרונות:

```
const invoices = [{ id: 1, price: 10 }, { id: 2, price: 32 }, { id: 3,
price: 40 }];
let amount = 0;
for (let i = 0; i < invoices.length; i++) {
    amount = amount + invoices[i].price;
}
console.log(amount); // 82
```

הסבר:

যוצרים לולאה שעוברת על המערך. המערך הוא לא של ערכים פרימיטיביים אלא של אובייקטים, אבל גם אובייקטים אפשר לשולחן באמצעות [i], כשה-*i* משתנה לפי מספר האיטרציה. לפני שהלולאה רצה, יוצרים משתנה בשם *amount* בשם שווה ל-0, ובכל פעם מוסיפים לו את ה-*price* באובייקט שיש באיטרציה. זה נעשה באמצעות הנקודה. להזכירכם, אפשר לגשת למפתח מסוים באמצעות שם האובייקט והנקודה. במקרה הזה האובייקט נמצא באיבר במערך ולא במשתנה עם שם משלו, אז אפשר לגשת למחיר באמצעות:

invoices[i].price

כש-*i*, להזכירכם, משתנה לפי האיטרציה.

תרגיל:

צרו לולאת while שמקבלת מספר וסופרת ממנו עד 0.

פתרונות:

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

הסבר:

לולאת while פשוטה שעובדת כל עוד i גדול מ-0 או שווה לו. בלולאה עצמה מדפיסים את המספר ואז מורידים 1 מ- i באמצעות כתיב קצר.

תרגיל:

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 while do היא אידיאלית אם רוצים פועלה אחת לפחות, גם אם התנאי לא מתקיים. במקרה הזה קודם הלולאה עובדת לפחות פעם אחת אז התנאי נבדק. כיוון שהמספר הוא שלילי, התנאי לא מתקיים והלולאה לא תמשיך לroz, אבל היא עבדה פעם אחת והדפסה את המספר.

תרגיל:

נתון מערך:

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

מזכיר בפוחטילים של משתמשים באתר היכרויות. כתבו פונקציה `forEach` שתדריס בקונסולה אך וرك את המשתמשים שהם בני פחות מ-30.

פתרונות:

```
let myArray = [
    { userName: 'Moshe', age: 20, },
    { userName: 'Yaakov', age: 25, },
    { userName: 'Ran', age: 40, },
];
myArray.forEach((value, index) => {
    if (value.age < 30) {
        console.log(value.userName);
    }
});
/*
"Moshe"
"Yaakov"
*/
```

הסבר:

משתמשים בפונקציית `forEach` שמקבלת פונקציה אונונימית מסווג חז. הפונקציה הזו נראה כך:

```
(value, index) => {
    if (value.age < 30) {
        console.log(value.userName);
    }
}
```

לא צריך להיבהל ממנו. זו פונקציה רגילה שריצה על כל איבר במערך באופן אוטומטי. יש ערך שבחרתי לקרוא לו `value` והמספר של האיבר במערך. מה שמשמעותו הוא הערך. הוא בעצם האיבר התוון במערך. בכל איטרציה יש איבר אחר. באיטרציה הראשונה מדובר באיבר הראשון, באיטרציה השנייה מדובר באיבר השני ובאיטרציה השלישית מדובר

באיבר האחרון. זה הכל. בודקים את האובייקט שיש באיבר אן, נכון יותר, את תכונת ה-`age` שלו, ומדפישים 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 וערך ומחזירה אך ורך מערך משתמשים שהגיל שלהם שווה למספר זהה או גדול ממנו.

פתרונות:

```
function getUsersAge(requestedAge, allUsersArray) {
    const answer = allUsersArray.filter((value) => {
        if (value.age < requestedAge) {
            return false;
        } else {
            return true;
        }
    });
    return answer;
}
const age40Users = getUsersAge(40, myArray);
console.log(age40Users); // [{ userName: 'Ran', age: 40, }]
```

הסבר:

יצרים פונקציה שמקבלת שני ארגומנטים – מספר מסוים וערך. משתמשים בפונקציית `filter` על מנת לסנן את המערך ולהשאיר שם רק את המשתמשים שרוצים. איך נראה פונקציית `filter`? בדוק ככה:

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

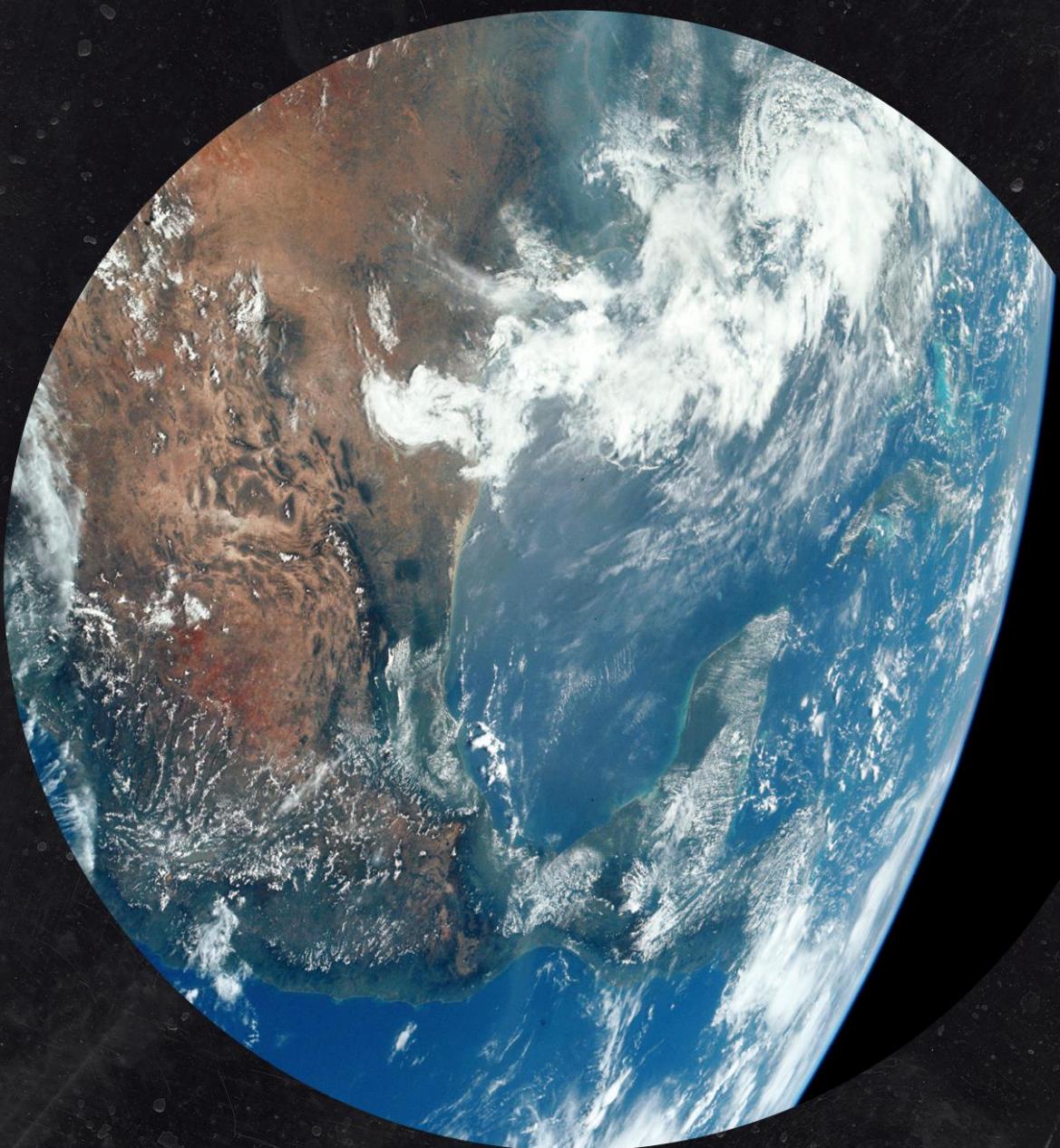
הפונקציה פשוטה יחסית ולא צריך להתבלבל מהחץ. ב-`value` יש בכל פעם איבר אחר מהמערך. אם פונקציית החץ מחזירה `true` הוא נשאר במערך. אם לא, הוא עף החוצה בביטחון פנימי. במקרה הזה מחזירים `false` על כל מי שהגיל שלו קטן מהמשתנה `requestedAge`, שאותו מקבלים מהפונקציה.

אל תשכחו שחייבים להחזיר את מה שמתקבל מה-`filter` אל משתנה אחר, כיוון שפונקציית `filter` לא משנה את המערך המקורי.

כל מה שנותר לעשות הוא לבדוק את הפונקציה. מעבירים לה 40, לדוגמה, ואת המערך של המשתמשים וראים שמקבלים משתמש אחד שהוא בן 40, כיוון ש-40 לא קטן מ-40, הוא מקבל `true` בפונקציית הפילטר.

פרק 13

ג'אווה סקייפט בסביבת דפסון



ג'אוהסקרייפט בסביבת דף

עד כה, כל התרגולים היו עם הקונסולה בלבד. סביבת העבודה היא בדף ועם עורך טקסט, אבל בגודל הדבר הייחודי שעשויהם בדף היה בקונסולה. הגיע הזמן להתחיל למש את הידע שלמדנו ולראות מה אפשר לעשות בעזרתו במקרים.

כיום ג'אוהסקרייפט רצה במגוון אדריכלות סביבות, אבל בעבר הסביבה הייחודית שבה היא רצתה הייתה הדף. כאמור, באתר אינטרנט השתמשו בה כדי ליצור התנהלות עם המשתמש, אнимציות, אפקטים, וידאו ועוד. באופן עקרוני, כל הידע אינטראקטיביות עם המשתמש, אнимציות, אפקטים, וידאו ועוד. באופן עקרוני, יכול להיות שלמדנו עד עכשיו הוא הבסיס הנדרש ליצור כל הדברים האלה בדף. יכול להיות שלולאות, אובייקטים ופונקציות נראים לא מעניינים לעומת אнимציות וסרטונים, אבל הם מה שפועל מאחורי הקלעים כדי לאפשר את כל הדברים היפנים בדף.

הספר הזה מלמד ג'אוהסקרייפט ולא HTML, שהוא עולם ומלאו. פרק זהה מלמד HTML באופן בסיסי בלבד, רק מספיק כדי להבין איך ג'אוהסקרייפט קשורה לכך. לשפת ג'אוהסקרייפט יש קשר הדוק עם HTML של הדף.

הסבר כללי על HTML

ראשי התיבות של **HTML** הם Hyper Text Markup Language והוא בעצם הטקסט המרכיב את כל דפי האינטרנט שאתה מכיר. כשהנכנים אל דף כלשהו בראשת, ולא משנה אם מדובר באתר חדשות או באתר משחקים או סרטים – רואים דף HTML. דף זה הוא בעצם דף טקסט. בדיקן כמו קוד ג'אוהסקרייפט שאפשר לכתוב ב-notebook, גם קוד HTML הוא קוד שאפשר לכתוב בכל מקום. אם אתם רוצים לראות את קוד ה-HTML של כל דף, לחצו על הכפתור ימני בעבריו ואז על "הראה מקור" או "view source". תראו מיד המונ-המון טקסט עם סימנים נאלו < וכאלו >. הדברים שנמצאים בתוך החצים נקראים "אלמנטים של HTML" ויכולים להכיל טקסט או אלמנטים אחרים. אלמנט בסיסי של HTML נראה כך:

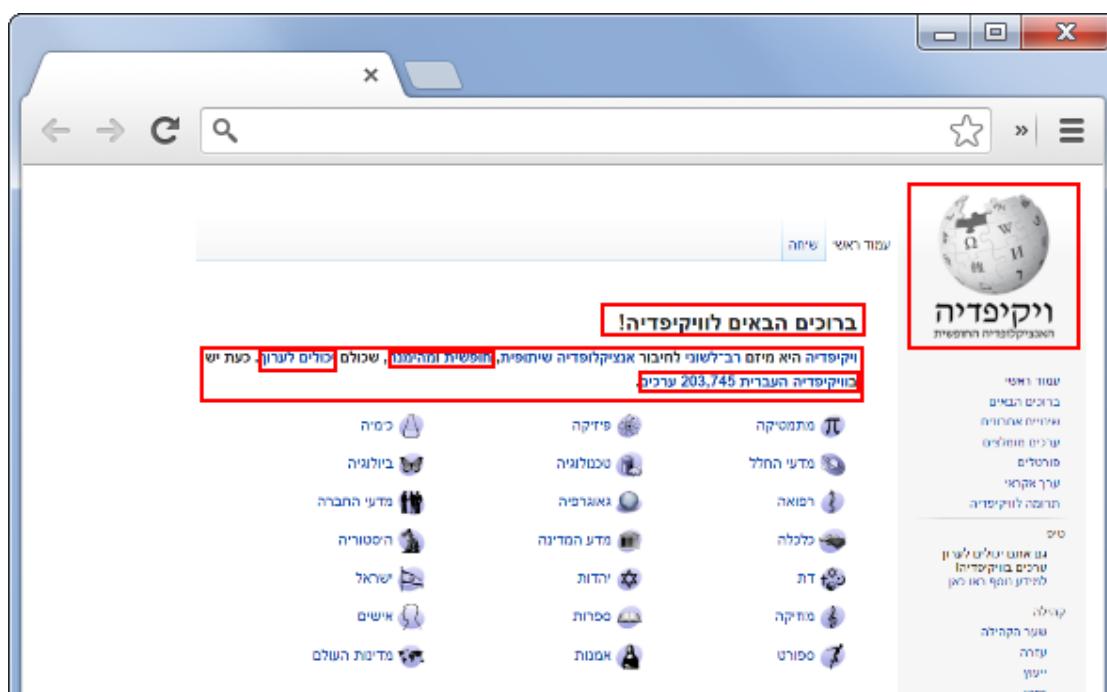
```
<elementName>Element Content</elementName>
```

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

אם מדובר באלמנט שנקרא "אלמנט שנסגר מalto", הוא נראה כך:

```
<elementName />
```

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



כאן רואים את דף הבית של ויקיפדיה. כל אובייקט, קישור או טקסט בו הם אלמנט HTML. הקפתוי כמו אלמנטים בריבועים. למשל הלוגו של ויקיפדיה הוא אלמנט מסווג תמונה. הטקסט "ברוכים הבאים לוויקיפדיה!" הוא אלמנט מסוג 1ה, ככלומר כותרת ראשית. יש גם אלמנטים בתוך אלמנטים. טקסט הפתיחה הוא אלמנט ויש לו אלמנטים בניים של קישורים (סימניתי 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 שנייה, `k` הוא פסקה, `a` הוא קישור, `div` הוא אלמנט כללי וכן הלאה. יש عشرות סוגים של אלמנטים.

לכל אלמנט יכולות להיות תכונות (attributes). למשל, אם יש קישור, אחת התכונות של האלמנט היא המקום אליו ה קישור מפנה. התכונה זו נקראת `href`. תכונה אחרת

שיכולה להיות ל קישור היא `title` – הטקסט שמוופיע אם מציבים את העכבר מעל הקישור.

הינה דוגמה למה שראויים אם כתבים בתוך מסמך HTML אלמנט של קישור:

```
<a href="https://google.com" title="Link to google">Google.com</a>
```



אפשר לראות איך כל תכמה של האלמנט יכולה לשנות משאו בהתנהגו.

יש אלמנטים שכיליהם טקסט כמו קישור או פסקה:

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

יש כאלה שלא, כמו תמונה. למשל:

```

```

הבדל הוא שהאלמנטים ללא הטקסט נסגרים בלוכסן בסופם ולא בשם האלמנט. קלומר

ממש לא כה:

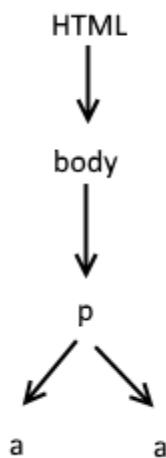
```
</img>
```

כאמור, יכולים להיות אלמנטים בתוך אלמנטים.

הינה לדוגמה פסקה שכילה 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 שוחשב אבא, תגית ה-`body`, ואז אלמנט הפסקה שהוא אלמנט האב. לאלמנט הפסקה (האב) יש שני ילדים שהם ה-`a` – שני קישורים, אחד לגולגל ואחד לדuckduckgo. בכל דף HTML ממוצע יש מאות אלמנטים כאלה מסוגים רבים ו��ונים ועל כולם אפשר להשפיע באמצעות ג'אווהסקריפט.

זהאים של תגיות HTML

לאלמנטים של HTML יכולים להיות זהאים, משהו שבעזרתו אפשר לזהות את האלמנט לצרכים שונים. בדיק נמו בני אדם שיש להם שם פרטי ושם משפחה, גם לאלמנטים שיש שני זהאים. המזהה הראשון הוא כללי וקוראים לו `class`. לא להתבלבל עם קלאס של ג'אווהסקריפט. במקרה של HTML, `class` הוא מקביל לשם משפחה, ובבדיקה כפי שיש כמה אנשים בעולם עם שם המשפחה שלי, אפשר לחת את אותו `class` לכמה וכמה אלמנטים אחרים. הם יכולים להיות בקשר אב-ילד או לא. אין גבולות! אפשר גם לחת כמה שמות משפחה לאותו אלמנט. המזהה השני נקרא `p` והוא מזהה ייחודי. אסור לחת את אותו `p` לשני אלמנטים באותו דף. ה-`p` הוא ייחודי ומוחד לכל אלמנט, ואפשר לחת `p` אחד בלבד לכל אלמנט.

אפשר כמובן להציג גם מזהה `class` וגם מזהה `p` לאלמנט אחד. במא משמשים? תלוי بماה שורצים לבצע ומבנה הדף. לאלמנטים מיוחדים שורצים הגיעו אליהם בנסיבות מסוימות ג'אווהסקריפט נותןים `p` בלבד.

גישה אל תגיות באמצעות ג'אוּהַסְקְרִיפֶט

לתרגול, צרו דף HTML לדוגמה ותציבו בו ג'אוּהַסְקְרִיפֶט. הדף זהה לדף שהסבירתי עליו בפרק על בניית סביבת העבודה, אבל חשוב להזכיר על דבר אחד – דףדף מריצ' את קוד הג'אוּהַסְקְרִיפֶט ומרנדר את האלמנטים בסדר מקביל למלמעלה למטה. לצורך התרגול חשוב מאד שקובצי הג'אוּהַסְקְרִיפֶט והקוד של הג'אוּהַסְקְרִיפֶט יהיו בתחום הדף, כיון שם הקוד יחפש את האלמנטים של ה-HTML או יתייחס אליהם, הדףדף יהיה חייב להכיר אותם לפני שהוא מריצ' את הקוד – ככלומר קובצי הג'אוּהַסְקְרִיפֶט שמתייחסים לאלמנטים חייבים להיות מתחת לאלמנטים. זו הסיבה שבגללה יש להזכיר, לפחות כרגע, קוד הג'אוּהַסְקְרִיפֶט יהיה מתחת לאלמנטים.

```
<!doctype html>
<html>

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

<body>
  <!-- HTML elements will be here -->
  <script src=".//source.js"></script>
</body>

</html>
```

בקוד יש הערת HTML. כל מה שמדובר ב-HTML בתגיות <--> נחשב להערה ולא מודפס, בדומה להעת ג'אוּהַסְקְרִיפֶט. היכן שיש הערת HTML – זה המקום שבו כתבים את האלמנטים. תגיית הסקריפט תכיל קישור לקובץ הג'אוּהַסְקְרִיפֶט, ולפיכך היא חייבת להיות מתחת ל-HTML.

אלמנטים של HTML מתרגמים לאובייקטים של ג'אוּהַסְקְרִיפֶט

כל אלמנט שיש ב-HTML יכול להיות מתרגם לאובייקט ג'אוּהַסְקְרִיפֶט כשר למהדרין. כשקיים ג'אוּהַסְקְרִיפֶט חי בתוך HTML הוא יכול לגשת אל ה-HTML הזה באופן ישיר אליו צורך בטעינה. ה-HTML מתרגם לסוג של ישות שנקראת DOM, ראשי התיבות של Document Object Model. הישות הזאת נמצאת בזיכרון זמין דרך קוד

הג'אוوهסקריפט. נשמע אבסטרקטי מדי? הבה נראה. אחרי שמציבים את הג'אוوهסקריפט בתוך ה-HTML, יוצרים בתוכו אובייקט `h1`, שהוא בסגנון זהה:

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

בקובץ של הג'אווהסקריפט מכנים אותו הקוד הבא:

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

אם טוענים את העמוד אפשר לראות שהטקסט שבכותרת הוא לא "Headline" אלא "Hello world". איך עושים את זה? ראשית משתמשים באובייקט הגלובלי `document`. האובייקט הזה קיים בכל ג'אווהסקריפט שנמצא בתוך HTML ומכיל את כל ה-DOM. הוא אובייקט כמו כל אובייקט אחר ויש לו מתודות.

אחת המתודות היא `getElementById`, שמקבלת ארגומנט אחד. איזה ארגומנט? את ה-`id`. במקרה שלנו ה-`p` הוא הערך שהושם ב-`attribute` מסווג `p` של האלמנט `h1`. המתודה מחזירה את הייצוג של האלמנט ב-DOM עם כל המתודות שלו. ומעכשיו? חנוכה שלמה. אחד המאפיינים הוא `innerHTML`, שמאפשר לשנות את תוכן האלמנט.

אפשר גם ליצור אלמנטים חדשים לוחוטין ולהכניס אותם למסך:

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

כאן לדוגמה משתמש במתודה `createElement` כדי ליצור אלמנט מסווג פסקה או תגית `<p></p>`, להכניס אליו תוכן ואז להציגו אותה אל ה-`body` של המסמך באמצעות `appendChild`. עד שימושים במתודה `appendChild`, הייצוג של האלמנט נמצא בזיכרון. עד שהוא לא מחובר ל-DOM אין לו ייצוג ויזואלי.

אירועים עם HTML וג'אווהסקריפט

אפשר גם להאזין לאירועים שתרחשים בדף. אירוע יכול להיות קליק, מעבר של העכבר על אלמנט והמון דברים אחרים. דוגמאות לכך אתמקד באירוע של קליק. הנה ניצור ב-HTML נפוץ שיש לו פו:

```
<!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 כפתור עם הכתוב "Click Me!". בקובץ הג'אווהסקריפט תופסם את אלמנט הceptor כרגע עם מוכנים אותו למשתנה:

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

עכשיו נרצה שבכל פעם שיקליקו על הceptor יראו משהו בקונסולה. את זה עושים בעזרת אירוע DOM. אירוע DOM מופעל על ידי אינטראקציה עם המשתמש שלוחץ על הceptor, והטיפול באירוע מתבצע באמצעות הפונקציה `addEventListener` שמגדירים אותה כמתפלת באירוע. מתודת `addEventListener` זמינה בכל אלמנט. היא מקבלת שני ארגומנטים. הראשון הוא סוג האירוע, ובמקרה זה: `click`. השני הוא פונקציה שפועלת בכל פעם שיש קליק. משתמשים במקרה זה בפונקציה חוץ 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 אתזה תוכלו לראות שבל פעם שלוחצים על הcptor, 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 הזה קוד שאתם כבר מכירים, שיוצר אלמנט. הקליק של המשתמש הוא בעצם הקלט. הפלט הוא הרינדור של האלמנט. אפשר ליצור קלט משוכל יותר באמצעות אלמנט ה-`HTML` שנקרא `input`. האלמנט הזה יוצר שדה טקסט והוא נראה כך:

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

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

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

הצעד הבא יהיה לחת את מה שיש ב-`input` ולהציב אותו ב-`HTML` בכל פעם שהמשתמש לוחץ על קליק:

```
let button = document.getElementById('myButton');
let input = document.getElementById('myInput');
button.addEventListener('click', (event) => {
    let myP = document.createElement('p');
    myP.innerHTML = input.value;
    document.body.appendChild(myP);
});
```

סביר שוב על הקוד המלא: ב-`HTML` יש שני אלמנטים, אלמנט אחד שיש לו `id` בשם `myButton` והוא cptor, ואלמנט נוסף של שדה טקסט שה-`id` שלו הוא `myInput`. שנייהם

נכנים אחרים לאחר כבוד למשתנים המתאיםים כדי שיתאפשר להתייחס אליהם. בשלב הבא מצלמידים אירוע קליק לנפתור. האירוע עובד בכל פעם שיש לחיצה. מדובר בפונקציה טהורה, כלומר צו שהוא ג'אוوهסקריפט בלבד, שבמקרה זהה יוצרת אלמנט מסווג פסקה. ב-HTML פסקה היא תגית <k>. היא לוקחת את מה שהמשתמש הכניס לתוך שדה הטקסט, מכניסה אותו לתוכנת `innerHTML` של אלמנט הפסקה ומצלמידה את אלמנט הפסקה אל גוף ה-HTML, וכן אפשר לראות אותו.

מה שחשוב להבין הוא שמדובר כאן בג'אוوهסקריפט לכל דבר אך בשילוב ה-DOM – כלומר אובייקטים וAIROUTS שמנגנים מהאלמנטים שיש ב-HTML. ברגע שימושים בו-`document` כדי ליצור אלמנט או לקרוא לו, הוא מקבל גם תכונות כמו `innerHTML` ואפשר להציג אליו AIROUTS מיוחדים כמו "קליק" באמצעות `.addEventListener`.

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

בכל הדוגמאות עד כה הריאתי אין מצלמידים את האלמנט שיצרנו ישירות ל-`document.body`, שהוא בעצם תגית ה-<body> שיש ב-HTML. זו תגית ייחודית שמופיעה רק פעם אחת במסמך HTML. אפשר להציג אלמנטים לכל אלמנט אחר! אdegim באמצעות תגיות <a> ו-<a>. מדובר בתגיות של רשימה ב-HTML, שנראית כך:

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

הבה ניצור דף אינטרנט שבו רשימת קניות שהמשתמש יוכל להכניס אליה פריטים. הדף HTML יוכל חלון של שדה טקסט, כדי שהמשתמש יוכל להכניס פריטים, קופטור של "הכנס" ומובן שلد 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 הג'אווהסקריפט יהיה פשוט. בדוק כמו בדוגמה הקודמת, מגדירים את שלושת השחקנים במשחק: הכתור שמאזיל את כל הפונקציה של ההוספה, שדה הטקסט שבו המשמש מכניס את מה שהוא רוצה להכניס לרשימה והרשימה הריקה.

```

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

```

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

```

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

```

הקוד שבתו הפונקציה הוא פשוט. יוצרים אלמנט חדש מסוג `li`, מאכלסים את ה-`innerHTML` שלו בערך המתקבל 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, ובגדיול אפשר לשנות בעזרתו על העיצוב ועל המיקום של כל אלמנט HTML. לכל תכונת CSS יש הערך שלה. למשל, אם רוצים לקבוע את רוחב האלמנט, התכונה היא width והערך הוא (למשל) 10px, שמשמעו רוחב של 10 פיקסלים. תכונת background-color קובעת את צבע הרקע, והערך יכול להיות צבע (כמו red) או ערך הקסדצימלי של צבע כמו #ff0000.

על מנת לשנות תכונת CSS או ליצור תכונה חדשה משתמש בתכונת style, שקיים בכל אלמנט DOM. לשם ההדגמה אוצר אלמנט HTML מסווג div עם id מסוים ב-HTML. אלמנטים מסווג div הם אלמנטים בסיסיים שאין להם עיצוב:

```
<!doctype html>
<html>

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

<body>
  <div id="myDivId"></div>
  <script src=".js"> </script>

</body>
</html>
```

אם תציגו את הדף זהה באמצעות הדפסן לא תראו כלום. הנה נכניס קצת צבע לאלמנט זהה באמצעות ג'אוּהַסְקְּרִיפְט:

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

יווצרם תכונות של גובה, של רוחב ושל צבע ורקע על מנת ליצור ריבוע אדום. זה ממש נחמד. אפשר להכניס, לשנות או לקרוא כל תכונה של style באופן חופשי כמובן. שימוש לב שמדובר בהשפעה על תגיota style של האלמנט ולא בשינויו קובץ CSS הקשור למסנן. לרוב לא מקובל לשנות צבעים באופן ישיר בג'אוּהַסְקְּרִיפְט, אלא ליצור class

של CSS שמכיל את התכונות המבוקשות ואז להוסיף את הקלאסים האלה לאלמנטים או להוריד אותם באמצעות ג'אוوهסקריפט.
למשל, בקובץ ה-CSS יהיה סלקטור של `:activated`:

```
.activated {
    background-color: red;
    height: 100px;
    width: 100px;
}
```

ובקוד עצמו:

```
const div = document.getElementById('myDivId');
div.classList.add('activated');
```

`classList.remove` מוסיף `class` מסיר אותו מהאלמנט שמחזיקים במשתנה. אם כל מה שכתוב לעיל נשמע לכם ג'יבריש – זה הזמן ללימוד CSS. כאמור, אני לא מכסה HTML או CSS בספר זה.

סלקטורים של DOM

סלקטור הוא שם כללי למетодות שמוצאות אלמנט. עד כה השתמשנו ב-`document.getElementById`, שהוא סלקטור שמוצא אלמנטים לפי ה-`id` שלהם. אך יש סלקטורים אחרים שאינם מבוססים על `id`. הסלקטור של `p` הוא ייחודי כי יוצאים מנוקדת הנחה שתמיד יש אלמנט אחד שיש לו את ה-`p` הזה. במקרים שיש סלקטורים מבוססים על `class` או על תכונה אחרת, יתקבל מערך של תוכאות שאפשר להתייחס אליו כאל כל מערך, אלא שבמערך זה יהיו אלמנטים של DOM.

הבה נדגים באמצעות ה-`getElementsByClassName`. ה-`getElementsByClassName` מוחזיר מערך של כל האלמנטים שיש להם `class` מסוים. קיימ HTML זהה:

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

(**שימוש לב:** לשם הנוחות אני לא מציב כאן את ה-HTML המלא כולל ה-`body` והקישור לקובץ הג'אווהסקריפט החיצוני.)

על מנת לבחור את כל האלמנטים שיש להם את ה-class זהה ולבצע אותו באדום (לצורך העניין)

עושים שהוא זהה:

```
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 הזה בדיק כמו על כל מערך רגיל בג'אוوهסקריפט, במקרה זהה באמצעות of, אבל אפשר להשתמש בכל לולהה שהיא. למדנו על לולאת for of בפרק על לולאות, אז אתם אמרוים להבין מה קורה פה. עוברים על כל איבר בלולאה ונונתנים לו גובה, רוחב וצבע. אם רוצים לתת אותם רק לאלמנט הראשון, צריך לעשות שהוא זהה:

```
recs[0].style.backgroundColor = 'red';  
recs[0].style.height = '100px';  
recs[0].style.width = '100px';
```

חשוב להבין שאף על פי שמדובר באלמנטי DOM, אנחנו בג'אווהסקריפט. כל מה שלמדנו עד עכשיו – אובייקטים, לולאות, מערכימים, סוגים מדיע – הכל רלוונטי גם לג'אווהסקריפט בסביבת דף. כשריצים ג'אווהסקריפט בסביבת דף – גם ה-DOM וגם הג'אווהסקריפט חיים בתוך הדף והוא מנהל את עצ ה-DOM ואת המנווע של הג'אווהסקריפט ודואג לחבר בין השניים. למען האמת, מפני שאתם כבר מכירים את היסודות התיאורתיים של השפה, אתם אמרוים לשЛОט גם בג'אווהסקריפט בסביבת דף. מדובר באותה שפת ג'אווהסקריפט בדיק.

אפשר להעביר כארגומנט רשימה של קלאסים עם הפרש של רווח ביןיהם. אפשר גם לבחור אלמנטים לפי שם התגית. למשל, הקוד הזה בוחר את כל התגיות שהן מסוג `div` וצובע אותן באדום. העיקרון הוא אותו עיקרון 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. CAN נדרש הינה ב-CSS, אבל כל סלקטור מורכב של CSS יכול להיות שימושי גם בג'אווהסקריפט. למשל נסתכל על ה-HTML הזה:

```
<div class="good_parent">
    <div class="rectangle">Rec 1</div>
    <div class="rectangle">Rec 2</div>
</div>
<div class="bad_parent">
    <div class="rectangle">Rec 3</div>
    <div class="rectangle">Rec 4</div>
</div>
```

אם רוצים לבחור אך ורק את ה-`div` שה-`class` שלו הוא `rectangle` והם יולדים של `.good_parent div.rectangle`, אפשר להשתמש בסלקטור של CSS שהוא:

`.good_parent div.rectangle`

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

הקוד ייראה כך:

```
const divs = document.querySelectorAll('.good_parent div.rectangle');
// [div.rectangle, div.rectangle]
for (el of divs) {
  el.style.backgroundColor = 'red';
  el.style.height = '100px';
  el.style.width = '100px';
};
```

גם פה זה לא מסובך במיוחד. ברגע שambilנים שיש סלקטורים שמחזירים מערך של אלמנטי DOM במקומ אלמנט DOM אחד, ושהמערך הזה הוא מערך ג'אווהסקריפטி לכל דבר ועניין אלא שהוא מכיל אלמנט DOM – נגמר הסיפור.

ל-`All`-`querySelector` יש גם גרסה של פונקציה בשם `querySelectorAll` רק את התוצאה הראשונה של הסלקטור 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 לchlוטין 7:

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

לשם הנוחות אצף רשימה של סלקטורים נפוצים ותפקידם:

| תפקיד | שם הסלקטור |
|--|---------------------------------|
| בוחר אלמנט אחד ויחיד לפי ה-ID מחזיר אלמנט אחד | document.getElementById |
| בוחר כמה אלמנטים לפי קלאס אחד או יותר (הklassים מועברים כמחרוזת טקסט עם רווח) מחזיר מערך של אלמנטים | document.getElementsByClassName |
| בוחר כמה אלמנטים לפי שם האלמנט - סימן לכל האלמנטים ב-DOM מחזיר מערך של אלמנטים | document.getElementsByTagName |
| בוחר כמה אלמנטים לפי תכונת השם – name מחזיר מערך של אלמנטים | document.getElementsByName |
| בוחר כמה אלמנטים לפי סלקטור של CSS אפשר להכנס כמה סלקטורים עם פסיק ביןיהם מחזיר מערך של אלמנטים | document.querySelectorAll |
| בוחר את האלמנט הראשון שמציאת סלקטור של CSS- מחזיר אלמנט אחד | document.querySelector |

אירועים נוספים

עד כה למדנו רק על אירוע קליק, אך יש אירועים רבים נוספים הקשורים לאלמנטים של DOM. למשל `hover`, המופעל בכל פעם שהעכבר עובר מעל אלמנט מסוים. למשל, להלן HTML זהה:

```
<div id="rectangle">Rec 1</div>
```

אפשר לקבוע את הצבע ואת הגודל באמצעות סגנולות CSS:

```
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
```

אבל אפשר גם לגרום לשינוי את הצבע באמצעות אירוע, אם העכבר יעלה עליו. את זה עושים באמצעות הצמדת אירוע בשם `mouseover` בשים `mousemove`. הטכניקה זהה לטכניקה שכבר למדנו:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('mouseover', (event) => {
    div.style.backgroundColor = 'yellow';
});
```

האירוע `addEventListener` מקבל שני ארגומנטים. הראשון הוא שם האירוע והשני הוא הפונקציה שעבדת כשהאירוע מופעל. משתמשים בפונקציית `vez`, בדוק כמו ב-`click`. אם תריצו את הקוד הזה תגלו שצבע הריבוע משתנה לצהוב ברגע שהעכבר עולה על הריבוע האדום.

חשוב לציין שעושים את זה בג'אויסקייפ לשם התרגול והסביר. במקרה אפקטיבם ככלו נעשים באמצעות CSS בלבד.

אני רוצה להתעכ卜 קצת על הארגומנט שפונקציית החץ מקבלת, והוא ה-`event`. נשלמדנו על פונקציות חץ ועל קולבקים בפרק על הפונקציות, הסברתי שנל קולבק יוכל לקבל כמה ארגומנטים שמי שמפעיל אותו מעביר.

במקרה הזה הארגומנט שמקבלים הוא אובייקט האירוע עצמו, שמניל שירות פרטיים. למשל, אם הԿפטורים Alt, Shift או Ctrl היו לחוצים, מה המיקום המדויק של האירוע, באיזו שעה הוא התקיים וכו' וכו'. כך למשל אפשר להפעיל את אירוע הקליק רק אם הԿפטור Alt לחוץ באופן הבא:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('click', (event) => {
    if (event.altKey === true) {
        div.style.backgroundColor = 'yellow';
    }
});
```

במקרה של הקוד הזה, רק לחיצה בשילוב כפתור Alt לחוץ תגרום לריבוע להשתנות לצבע צהוב. אובייקט ה-`event` (אפשר לקרוא לו בכלל שם נमובן) הוא אובייקט כמו כל אובייקט. אפשר לעשות עליו `console.log` כדי לתחות על קנקנו ואפילו מומלץ להביט בו. אפשר לעשות אותו דברים מעניינים מאוד.

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

שימוש לב שיש משמעות לאיורים. למה לא השתמשתי ב-`mouseover`? כיון שהוא מופעל בכל פעם שהעכבר נמצא מעל האלמנט, ככלומר המון פעמים – בכל פעם שהעכבר זו והאזור אליו הוא מעלה האלמנט. האירוע `mouseenter` מופעל רק פעם אחת – במקרה העכבר נכנס לתוך האלמנט. זה הכל.

יש המון איורים, מאירועי מקלדת ועד אירועי עכבר. יש גם איורים של אלמנטים (למשל אירוע שמוספעל אם מישוה משנה את ה-CSS של האלמנט) וαιורים הקשורים לאירוע `clipboard`. הנה רשימה של כמה אירועים עיקריים:

| הסבר | אירוע |
|--|-------------------------|
| אירוע של הקלקה באמצעות העכבר | <code>click</code> |
| סמן העכבר נכנס אל תחום האלמנט | <code>mouseenter</code> |
| סמן העכבר יוצא מתחום האלמנט | <code>mouseleave</code> |
| המשתמש לווח על כפתור העכבר בתוך האלמנט | <code>mousedown</code> |
| המשתמש שחרר את כפתור העכבר בתוך האלמנט | <code>mouseup</code> |
| המשתמש מגלגל את גלגלת העכבר כאשר הסמן נמצא על האלמנט | <code>wheel</code> |
| לחיצה על כפתור במקלדת | <code>keydown</code> |
| שחרור של כפתור במקלדת | <code>keyup</code> |
| המשתמש מבצע פוקוס על האלמנט (למשל באמצעות הטאב או העכבר) | <code>focus</code> |
| המשתמש יוצא מפוקוס על האלמנט (למשל באמצעות הטאב או הקלקה על אלמנט אחר) | <code>blur</code> |

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 שבתוכו יש כפטור עם id של button. בתחתית יש div שכותב בו result.

קוד הג'אויסקייפ הבא רץ באותו דף. מקבלים את שלושת האלמנטים כאלמנטים של parent DOM באמצעות getElementById בהתאם ל-id של כל אחד מהם, צובעים את ה-`style` באפור ונוטנים לו מדדים:

```
const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
```

התוצאה צריכה להיות משהו זהה:



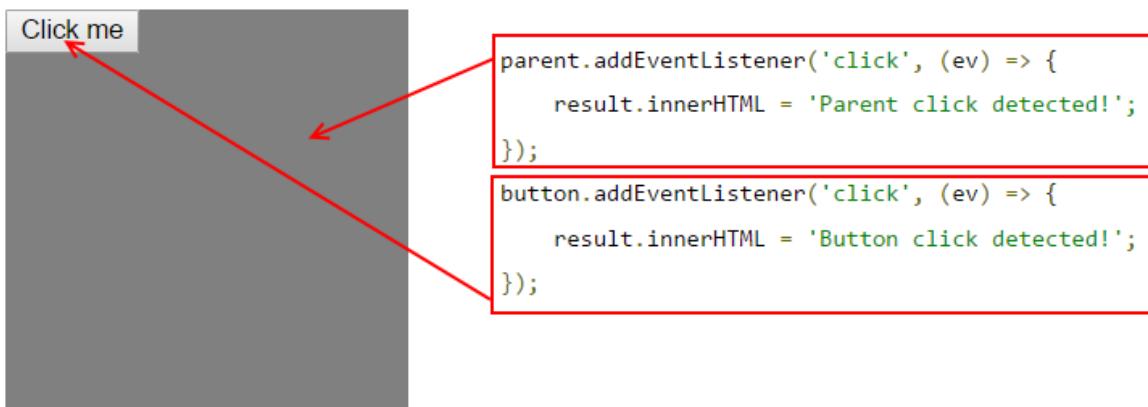
מה קורה אם מגדירים לאלמנט האב אירוע? תזכורת: אלמנט האב הוא ה-`div` שמכיל את הcptor. בדוגמה זו לאלמנט האב יש `po` ששמו `parent` (דיברנו על אלמנטים אבות וילדים מוקדם יותר בפרק).

```
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 cptor? אותו הדבר! למה? כי האירוע מופיע גם לאלמנטים של ההורים. האירוע מתחילה בcptor ומופיע לעלה עד שיש אלמנט שמתפל בו (ובמקרה זה האב). כלומר, אם אני אלמנט ויש אירוע על האלמנט שמכיל אותו (או על האלמנט שמכיל את האלמנט שמכיל אותו וכו'), אז האירוע יופיע לפני מעלה עד שיופיע באירוע כלשהו (או עד שהוא `body`). לוחצים על cptor, אין בו אירוע לחיצה. האירוע עולה למעלה. האם באלמנט האב יש אירוע? כן! אז האירוע מופעל וממשין לעלות לעלה, והוא מפעיל את כל האירועים באלמנטי האב השונים.

זה נחמד ו שימושי, אבל העסק מתחילה בסתובן 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 cptor? האירוע של cptor יתרחש, ואכן ב-`div` של התוצאה יודפס Button click detected, אבל מיד אחר כך, האירוע יופיע לעלה ויפעל את האירוע של parent, שבתווך ידפיס מיד ב-`div` של התוצאה Parent click detected.

לעולם לא תראו את האינדיקציה של הלחיצה על הכפתור ב-`div` ה-`result`! זה קצת בעייתי אם אנחנו רוצים לראות בליחיצה על הכפתור, Button click detected, ווק כשלוחצים על ה-`div` של האב לקבל Parent click detected.



מה יקרה אם לא רצים שהאירוע יופיע למעלה? באירוע שבו רצים לעזר את הפעוף חיברים להפעיל את:

`event.stopPropagation();`

שזמן באירוע עצמו, לצד שאר תכונות האירוע. קחו למשל הקוד הזה:

```

const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
parent.addEventListener('click', (event) => {
    result.innerHTML = 'Parent click detected!';
});
button.addEventListener('click', (event) => {
    result.innerHTML = 'Button click detected!';
    event.stopPropagation();
});

```

הוא יעשה בדיקת מה שרצים שהוא יעשה. לחיצה על האב תדייס שלחצתם על האב. לחיצה על הכפתור תפעיל את האירוע של הכפתור, אך האירוע לא יחולחל להלאה והאירוע של האב לא יופעל.

הצמדת אירועי ג'אווהסקריפט לאלמנטים ב-HTML

השיטה של `addEventListener` נחשבת לעדיפה בהרבה, אבל אפשר גם להצמיד אירועים של ג'אווהסקריפט ב-HTML באמצעות תכונת `onclick` לקליקים. למשל מהו זה:

```

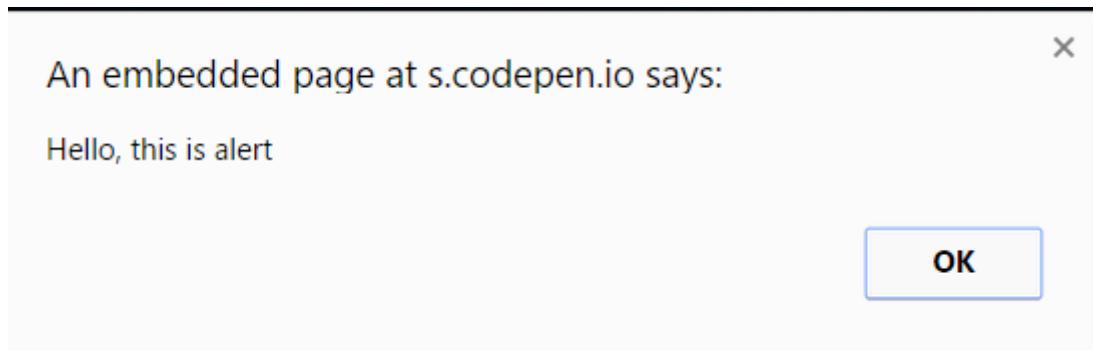
```

יצירת `prompt` ו-`alert`

עוד דרך מיושנת לתקשר עם משתמשים היא באמצעות `alert`. מדובר בפונקציה גלובלית זמיניםה בכל ג'אווהסקריפט שחייה בסביבת דף-דף. משתמשים בה כך:

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

מה שהמשתמש יראה הוא חלון לא גרפי שיקפוּץ עם ההודעה.



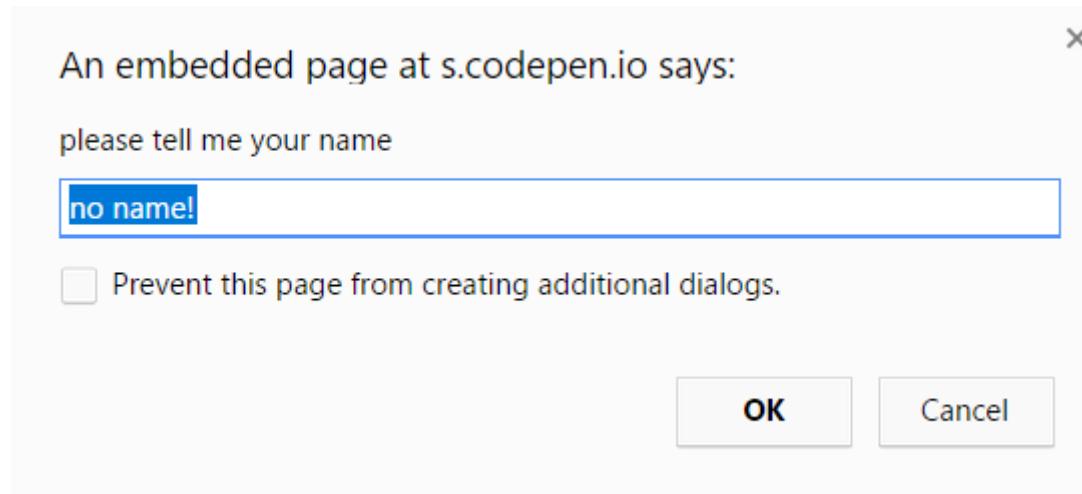
עיצוב החלון תלוי בדף-דף, בעיקר מפני שהוא בחלון של מערכת הפעלה. כאמור, מדובר בדרך מיושנת להציג הודעות למשתמש וכיום מעתים משתמש בה. מומלץ מאוד להציג למשתמש הודעות בדרך אחרת, כפי שהראיתי לכם – הקצתה `div` מיוחדת והדפסת הודעות עליה.

דרך נוספת לתקשר עם המשתמש היא באמצעות הפונקציה הגלובלית `prompt`, גם היא זמיניםה בכל ג'אווהסקריפט שחייה בסביבת דף-דף. הפונקציה זו מקבלת שני ארגומנטים. הארגומנט הראשון מכיל את המחרוזת למשתמש. כמשמעותו, מוצגת למשתמש המחרוזת שהוכנסה בארגומנט הראשון עם שדה טקסט שבו הוא יכול להכניס מחרוזת טקסט מסוילו, ואוותה פונקציית `prompt` מחזירה. הארגומנט השני הוא ערך ברירת המחדל שומותג למשתמש.

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

alert(result);
```

הקוד הזה למשל יציג את השאלה:



אם המשתמש יכנס שם, מייד אחר נסמן alert עם השם שהוא הכניס. גם כאן מדובר בדרך מינonta מאוד ועדייף לא להשתמש בה. אם רוצים לתקשר עם המשתמש, עושים זאת ב-[טקסט](#), כפי שראינו בדוגמאות הקודמות.

תרגיל:

צרו אלמנט מסוג `span` עם תוכן של "אני יודע קוד" והכניסו אותו ל-HTML.

פתרון והסביר:

יצרים HTML בסיסי עם קישור לקובץ ג'אוوهסקריפט, שייהי בתחתית קובץ ה-HTML.

```
<!doctype html>
<html>

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

<body>
  <!-- HTML elements will be here -->
  <script src="../source.js"></script>
</body>

</html>
```

בקובץ ה-HTML מוכנסים את הקוד הבא:

```
let myObj = document.createElement('span');
myObj.innerHTML = 'אני יודע קוד';
document.body.appendChild(myObj);
```

הקוד עצמו פשוט למדי. יוצרים אלמנט באמצעות `createElement` שנמצא באובייקט `document`. האובייקט זמין בכל קובץ ג'אוوهסקריפט שנמצא ב-HTML. את התוצאה של המתודה מוכנסים למשתנה `myObj`. מעכשו יש אלמנט DOM שיש לו מתודות משלו, וכל אחת מהן משפיעה עליו. אחת החשובות שבהן היא `innerHTML`, שמנήישה את התוכן לאלמנט שנוצר. כל מה שנותר לעשות אחרי שיוצרים את האלמנט ואת התוכן שלו הוא להכניס את האלמנט לגוף ה-HTML באמצעות `appendChild`.

תרגיל:

צרו דף אינטרנט שבו שדה טקסט וכפתור. אם המשתמש מכניס מספר לשדה הטקסט ולוחץ על הכפתור, הדף מציג רשימה ממושפרת באורך של המספר. למשל, אם המשתמש המכניס 3, תתקבל הרשימה:

- 1 •
- 2 •
- 3 •

פתרונות:

קובץ HTML:

```
<!doctype html>
<html>

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

<body>
  <button id="myButton" type="button">Click me!</button>
  <input id="myNumber" />
  <ul id="myList"></ul>
  <script src=".//source.js"></script>
</body>

</html>
```

בקוד עצמו:

```
const button = document.getElementById('myButton');
const number = document.getElementById('myNumber');
const list = document.getElementById('myList');
button.addEventListener('click', (event) => {
  const numberValue = number.value;
  for (let i = 0; i < numberValue; i++) {
    const myListItem = document.createElement('li');
    myListItem.innerHTML = i + 1;
    list.appendChild(myListItem);
  }
});
```

הסבר:

ב-HTML יוצרים שלושה אלמנטים: שדה טקסט שבו המשתמש יוכל להכניס מספר, רשימה ריקה וכפטור להפעלת הפונקציה שתיקח את המספר ותכנס פורטימ לרשימה. ב-HTML יהיה קישור לקובץ ג'אוوهסקריפט שבו יהיה הקוד. אלו שלושת השחקנים.

בקוד הג'אווהסקריפט מגדרים את שלושת השחקנים באמצעות `document.getElementById`.

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

מצמידים אירוע קליק שמתקבל שני ארגומנטים באמצעות `addEventListener`. הראשון הוא שם האירוע והשני הוא מה שקורה כשהוא מופעל. שימוש לב שהשתמשי כאן בפונקציית `cz`:

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

בפונקציה עצמה מגדרים את המספר שמקבלים מהמעותש:

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

ברגע שיש מספר, מפעילים לו את `for` שתרוץ לפיו:

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

בתוך הלולאה יוצרים אלמנט `li` (`li` הוא אלמנט של ITEM ברשימה) ומכניסים אותו לרשימה. מכךשים את תוכנו באמצעות `+ i`. זה הכל.

תרגיל:

צרו כפתור ואלמנט מסוג `div`. לחיצה על הכפתור תהפוך את האלמנט לצהוב, ברוחב של 100 פיקסלים וגובהה של 100 פיקסלים.

פתרון והסביר:

קוד ה-HTML:

```
<!doctype html>
<html>

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

<body>
  <button id="myButton">Click to change color</button>
  <div id="myDiv"></div>
  <script src=".source.js"></script>
</body>

</html>
```

קוד הג'אוהסקריפט:

```
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 לג'אוהסקריפט.

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

תרגיל:

במה שאל תרגיל הקוד, כתבו קוד שיגרום לכך שלחיצה על הכפתור תקבע את האלמנט בצהוב, לחיצה נוספת יתבצע נספה תקבע אותו שוב בצהוב וכן הלאה.

פתרון והסביר:

קוד ה-HTML:

```
<!doctype html>
<html>

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

<body>
    <button id="myButton">Click to change color</button>
    <div id="myDiv"></div>
    <script src=".source.js"></script>
</body>

</html>
```

קוד ה-ג'אויסקייפ:

```
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 המקורי מגדירים את שני השחקנים כאלמנטים של DOM באמצעות `getElementById`. מגדירים משתנה בוליאני בשם `activated` ומגדירים אותו כ-`true`. נוסף על כן מתחלים את הריבוע הצהוב. צובעים את הרוחב, את הגובה ואת הצבע ההתחלתיים.

מצמידים אירוס קליק לנפתור באמצעות `addEventListener`, שכבר חरנו בתרגילים קודמים. הפונקציה ש-`addEventListener` מפעילה בכל קליק מורכבת מעט. ראשית בודקים אם הוא `true`. אם כן, צובעים את האלמנט בצהוב ומשנים את ה-`false`-ל-`activated`.

אם הוא `false`, הוא נקבע באדום ומשנים את ה-`activated` ל-`false`.

פרק 14

דיבאג'ינ'



דיבאגינג

בכל כלי מפתחים שהוא – בין שמדובר בכרום ובין שבפיירפוקס או באdag' – יש אפשרות להפעיל כלי שנקרא "דיבאגר". כל כלי מפתחים הוא רכיב תוכנה שנמצא בכל דףון ללא צורך בהורדה או בהפעלה. מדובר בשם כללי שמשייע למפתחים לבחון את הקוד שלהם. בדףון הכללי זה עוזר לפטור תקלות בג'אוوهסקריפט ולהבין מה קורה בנבci הקוד. עד כה השתמשנו בעיקר ב-`console.log` על מנת להדפיס דברים בקונסולה. הכללי הזה חוסך את התהליך של הדפסת פלט בקונסולה ומאפשר לראות בדיקות מה קורה בג'אווהסקריפט.

כדי לתרגל שימוש בדיבאגר, נניח שבפרויקט שלכם יש קובץ HTML סטנדרטי וקובץ ג'אווהסקריפט שמקשור אליו, בדיקת כמו שהסבירתי בפרק על התקנת סביבת העבודה. קובץ HTML נראה כך:

```
<!doctype html>
<html>

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

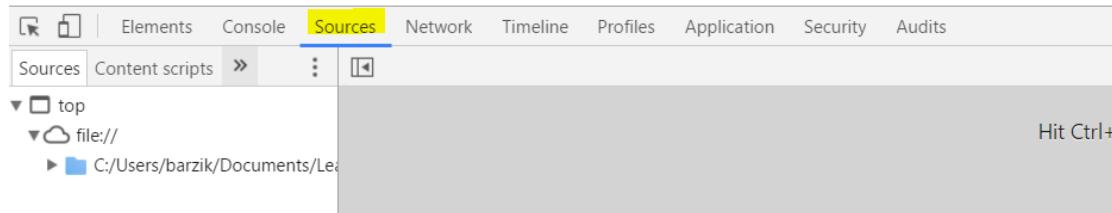
<body>
  <script src="./source.js"></script>
</body>

</html>
```

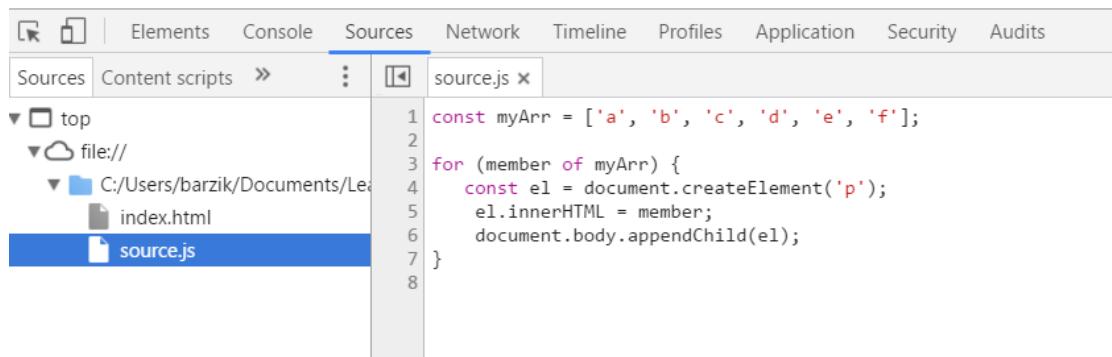
אפשר לראות שיש כאן קישור ל-`source.js`, שנמצא באותה תיקייה של קובץ HTML. התוכן ב-`source` יהיה משהו כזה:

```
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 Cmd + Shift+N + אם אתם במק. עכשו מזוין את לשונית sources:

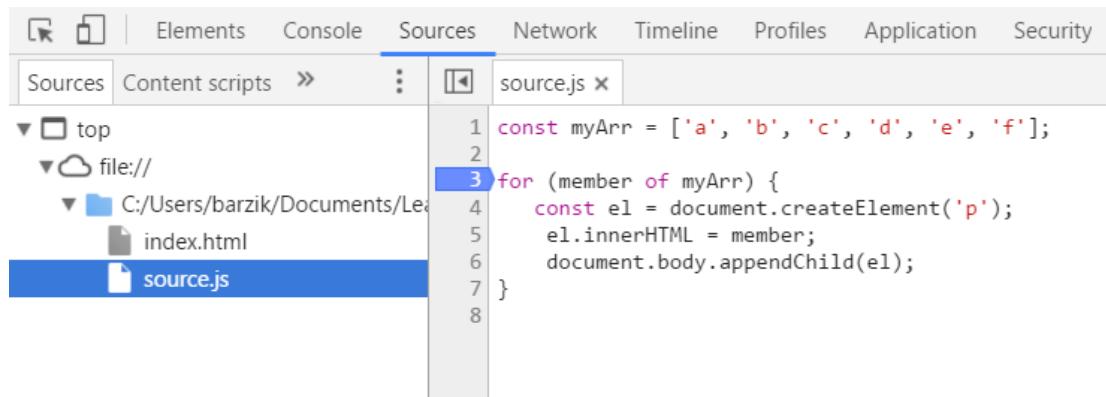


מצד שמאל אפשר לראות את כל עץ הפרויקט. פותחים אותו ומחפשים את קובץ `ג'אווהסקריפט.js`. ברגע שלוחצים עליו אפשר לראות את כל הקוד מצד ימין.



במקרה הזה העץ מצומצם מאוד כיון שהוא כולל רקייה אחת בלבד. אם העץ מורכב, אפשר ללחוץ על k (או k + Cmd + Ctrl + k במק) ולהקליד את שם הקובץ.

הדייבאגר עובד באופן עקרוני עם נקודות עצירה (באנגלית breakpoints). אפשר לבחור נקודה (או כמה נקודות) בסקריפט שבו הוא יעצור ויאפשר לבדוק את מה שקרה. הנה נראה! נלחץ על השורה השלישית, ממש היכן שנמצאת לולאת `for of`. מיד יופיע סימון כחול המאשר שנקבעת העצירה קיימת:



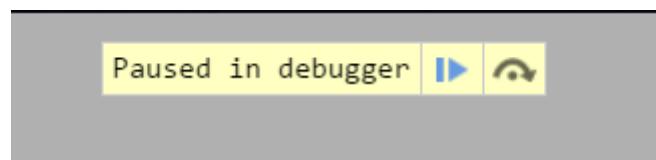
The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. In the left sidebar, under 'File://', there is a folder for 'C:/Users/barzik/Documents/Le...' containing 'index.html' and 'source.js'. The 'source.js' file is currently open. A blue rectangular highlight is placed over the third line of code, which contains the opening brace of the for-of loop. The code itself is as follows:

```

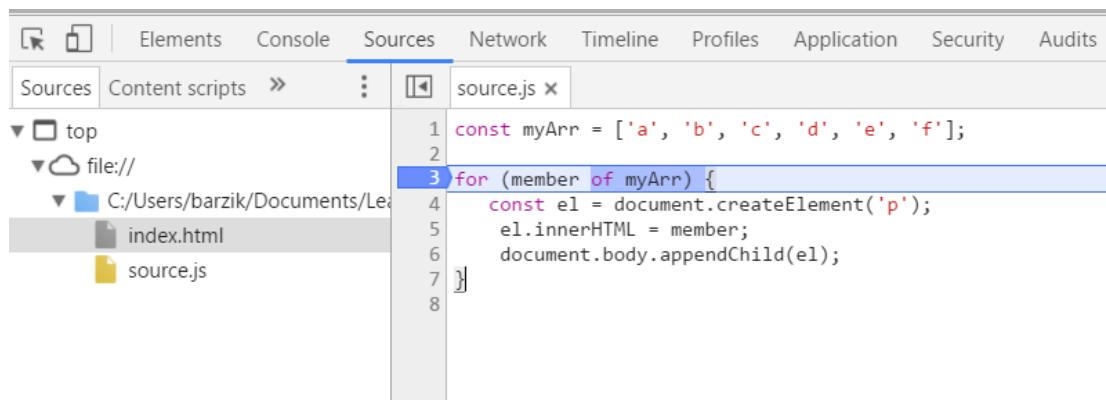
1 const myArr = ['a', 'b', 'c', 'd', 'e', 'f'];
2
3 for (member of myArr) {
4   const el = document.createElement('p');
5   el.innerHTML = member;
6   document.body.appendChild(el);
7 }
8

```

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

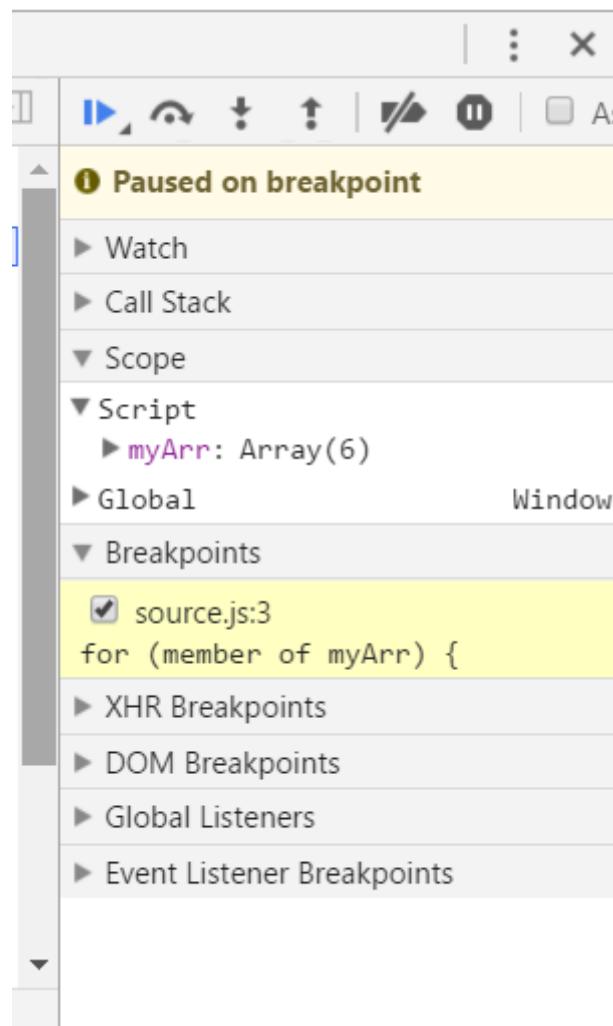


בדיבאגר עצמו תוכלו לראות שערכותם בנקודות העצירה:

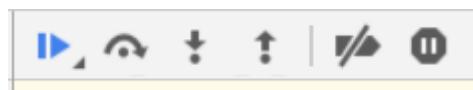


The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. The left sidebar shows the same file structure as before. In the main pane, the code for 'source.js' is displayed, identical to the previous screenshot. In the bottom right corner of the code editor, there is a small blue rectangular highlight over the variable 'myArr' on line 1. This indicates that the variable is being watched. The bottom of the screen shows the 'Watch' pane, which lists the variable 'myArr' with the value '[a, b, c, d, e, f]'. The 'source.js' file is also listed in the watch pane.

מצד ימין אפשר לראות פרטיים על המצב הנוכחי:



ב-`scope` למשל אפשר לראות את כל המשתנים הקיימים באותו `scope`, ובמקרה זה רק `myArr`. אפשר גם לראות את כל האלמנטים הגלובליים (כמו `document` שלמדו עליון בפרק על HTML וג'אווהסקריפט) ואובייקטים מובנים אחרים שעוז נלמד עליהם. אם עוברים לკונסולה, אפשר להקליד פקודות שייעבדו בנקודת הזמן זו של הסкриיפט. מעל דיווח המצב יש שורת פקדים שבו ניתן להפעיל את הדיבאגר:



בלחץ הראשון אפשר להריץ את הסкриיפט עד נקודת העצירה הבאה, אם יש כזו. אם אין כזו, הסкриיפט ורוץ עד הסיום. בלחץ השני אפשר לעשות `step over` – קלומר להריץ את הסкриיפט לשורה הבאה. הלחץ השלישי והרביעי, `step out` ו-`step in`, שמורים לכינוסה אל פונקציות ולא נזון בהם בשלב זה. אפשר גם למחוק את כל נקודות העצירה.

הבה נלחץ על step over. אפשר גם ללחוץ על F10. תוכלו לראות שהתקדמתם צעד אחד מעבר לנקודת העצירה וכרגע אתם בתוך הלולאה:

עכשו יש בסkop שני חלקים – גם זה של הבלוק, כלומר של ה-for-of. אפשר לראות שיש את ה-el, אבל הוא עדין לא מוגדר. הוא יהיה מוגדר רק בשורה הבאה. בואו נتقدم אליה באמצעות לחיצה שנייה על הפקד או על F10:

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

בגדיול, אין פה יותר מדי, אך מדובר באחד הכלים החשובים ביותר למתקנים וכדיי מאד ללמידה להשתמש בו – וזה להשתמש בו – כבר בהתחלה. היכולת המרהיבה של הדיבאגר לסייע במציאת תקלות היא חשובה מאוד, והוא גם נותן תמונה מלאה על מה שתרחש בקוד – כולל המשתנים המופיעים בסקוב או בקלוזרים השונים (ואם שכחتم מה זה, חזרו לפרק על הפונקציות).

שימוש לב: מطبع הדברים עיצוב של דפדף משטנה כל הזמן וייתכן שצילומי המסך המופיעים פה לא מעודכנים מספיק. אם כן, לא צריך להיחוץ – העיקרון הוא אותו עיקרון גם אם לשונית קוראים בשם אחר או אם האיקון נראה קצת אחרת.

פרק 15

אובייקטים גלובליים ואובייקטים מוגנים



אובייקטים גלובליים ואובייקטים

מוכנים

הראיתי בכמה ההזדמנויות שקיימים אובייקטים שיש גישה אליהם. הדוגמה הטובה ביותר היא `document`. על אף שה-`document` היא לא מילה שמורה 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 הקיימים לג'אווהסקריפט. ב"אובייקטים גלובליים" אני מתכוון לאובייקטים שמוגדרים בrama הגלובלית ומשם מחלחים לכל סCOPE:

```
console.log('I am global');
(() => {
  console.log('I am inside my scope');
  const myArray = [1, 2, 3, 4];
  for (member of myArray) {
    console.log(member);
  }
})();
```

בדוגמה הקוד הזה, למשל `console` הוא אובייקט שזמן מיד בrama הגלובלית, בסCOPE של הפונקציה האנונימית המרצה את עצמה (שלמדנו עליה בפרק על הפונקציות) וגם בסCOPE של לולאת `for`. אם דורסים את `console` בrama 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 הסCOPE של הפונקציה האנונימית את `console`.`console` שלא מדובר במילה שמורה אלא באובייקט גלובלי, אפשר לעשות זאת זה בקלות. זה אומר

שכל `log` בסkop של הפונקציה האנונימית וכל `the-log` שהוצבו באותו skop בנסיבות closure (שגם עליו למדנו בפרק על הפונקציות), לא יעבדו כיון שפונקציית `log` תהיה ריקה ב-`console`. לעומת זאת, הטקסט שנעביר ב-`log` פשוט לא יוצג.

כל ההקדמה הארוכה זו נועדה להסביר שיש הבדל משמעותי בין מילה שמורה, שהיא עצם מעצמותיה של ג'אווהסקריפט, לבין אובייקטים גלובליים שהם מכובדים וחשובים, אך עדין רק אובייקטים שמצויתים לכללי השפה. יש אובייקטים גלובליים שזמינים אך `window` ורך לג'אווהסקריפט שעבוד בסביבת HTML, `HTML`, כמו `document` שהכרנו קודם, `Screen`, `URL` וכוכו, ויש כמובן שזמינים אך ורק לג'אווהסקריפט שעבוד בסביבת שרת, כמו `process` למשל. אובייקטים גלובליים רבים זמינים לכל ג'אווהסקריפט שהוא, כמו `console`. כל האובייקטים האלה נקראים Web APIs.

לכל אובייקט גלובי יש תכונות או מетодות. על מנת לבחון אותן, אפשר לחפש אותן בתיעוד בראש או פשוט להסתכל בקונסולה. בדוגמה הבאה פותחים את כל המפתחים בכרום, עוברים לკונסולה, מקלידים `console.log` ולוחצים על `Enter`. בבת אחת רואים את כל המethodot שיש לკונסולה.

```
> 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 כתבים בקוד:

```
console.error('This is error!');
```

רואים שהקונסולה שולחת את הטקסט על רקע אדום כשגיאה. אם משתמשים ב-`console.info` רואים שמתקבלות התרעות בגוון אחר.

כאמור, ה-`API` Web מאפשר לקבל המון מידע מהתקנים שונים. אחד האובייקטים השימושיים ביותר הוא אובייקט `window`, שיש לו תוכנות וمتודות רבות. למשל `window.location`, שמחזיר את ה-`URL`, כלומר 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 שהם לא מגיעים משום מקום אלא מזורקים לסקופ הגלובלי על ידי ג'אווהסקריפט עצמה. מי שרוצה לראות את רשימת האובייקטים המלאה של `Web APIs` מוזמן לגשת לאתר המקיף והטוב ביותר של תיעוד ג'אווהסקריפט: [MDN Mozilla Developer Network](#). שם יש פירוט מקיף ביותר על כל האובייקטים. עם כל שינוי והתקדמות של הדפינים נוספים אובייקטים.

שימוש לב: ניתן לגשת לאובייקט הגלובלי `window` גם באמצעות `globalThis`. האובייקט זהה הוא כינוי לאובייקט הגלובלי המשתנה מסביבה לסביבה. במקרה של סביבת דפדפן הוא מחליף את `window`. כך למשל, ניתן לכתוב:

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

למשל, בשנת 2015 נוספו אובייקטים גלובליים שמאפשרים תקשורת באמצעות ג'אווהסקריפט עם מצלמת הרשת, וכך כל קוד ג'אווהסקריפט יכול לגשת אל מצלמת הרשת, לאחר בקשת רשות מהמשתמש, ולהשתמש בפייד המגיע ממנו למטרות שונות. הרשימה, 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 מזורקים מהדף או מסביבת העבודה אלא מוגדרים בתוכניות של השפה עצמה, כלומר הם חלק מהשפה. נשמע אולי שמדובר בהבדל דק, אך הוא ממשועודי מאד.

אובייקט document מגיע מהדף. הוא לא חלק رسمي מהשפה אלא סוג של API שהדף מיישם. אובייקט מוגנה מסוג Math, למשל, והוא אובייקט שמניע מהשפה עצמה ומוסיפה בהגדרות שלה. בעוד אובייקט גלובלי כמו console.error יכול להיות מיושם בדף כروم בצורה אחרת, בפירופוקס בצורה אחרת ובסביבת שרת בצורה אחרת, אובייקט מוגנה PI Math שמחזיר את ערך הפאי מתנהג באופן זהה לחלווטין בלי שום קשר לדף.

או אם ג'אווהסקריפט רצה בסביבת שרת.

חלק מהאובייקטים הם אובייקטים של ממש וחלקם ממומשים כפונקציה (שגם היא אובייקט).

גם את האובייקטים המוגנים אפשר לדרוס. כאן לדוגמה את האובייקט Math וגורמים לא-PI להציג ערך של 5:

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

בהמשך הפרק נעבור על כמה אובייקטים מוגנים חשובים.

parselnt

הfonקצייה הגלובלית `parselnt` מקבלת ממירה מחרוזת טקסט למספר. הפונקציה מקבלת שני ארגומנטים. הראשון הוא מחרוזת הטקסט והשני הוא הבסיס. המלצת הרשמית היא להשתמש בשני הארגומנטים, אבל בפועל צריך רק ארגומנט אחד. כך זה עובד:

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

אפשר לראות שהารוגומנט הראשון הוא מחרוזת טקסט, אבל מייד אחרי שמעבירים אותו ב-`parselnt` רואים בקונסולה ש-`convertedNumber` הוא מספר. הפונקציה תעבור גם אם יש המון רווחים:

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

אבל אם היא מקבלת משוה שהוא לא יכול להתחזק אליו, היא מחזירה `NaN` (כאמור, הקיצור ל-`Not a Number`, לא מספר):

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

ראוי לציין שבמקרה ההפוך פונקציית `parselnt` דוקא עובדת. אם היא נתקلت במספר שיש אחריו טקסט, היא נזנתת 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'אווהסקריפט. כמובן הוא מעריצה אותו (evaluation) כאילו הוא ג'אווהסקריפט ומחזירה את התוצאה.

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

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

אפשר לנקח את זה רוחק יותר ולהכניס לתוך מחרוזת הטקסט גדרות של משתנים ממש:

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

פה למשל מציבים במחרוזת הטקסט מספר לתוכה `a` שהוגדר בסkop הגלובלי. כיוון שפונקציית `eval` ממש מריצה את מחרוזת הטקסט כמו ג'אווהסקריפט, משתנה `a` מקבל ערך. כל מה שמוקף ב-`eval`, לא משנה מה אורכו, יירץ כמו ג'אווהסקריפט. כתיבה של שהוא זהה:

```
let a;
let result = eval('a = "Hello";');
console.log(a); // Hello
```

דוגמא לכתיבה של שהוא זהה:

```
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 באמת מה אתם עושים. הסיבה היא ענייני אבטחת מידע – בדרך כלל מחרוזת הטקסט מגיעה בדרך זו או אחרת משתמש, וממן אפשרות למשתמש להריץ מה שהוא רוצה בקוד והואفتح אדיור לביעות אבטחה ולביעות אחרות. לפיכך למדנו כאן את `eval`, אבל רק לצורך ההסביר מדוע מומלץ לא להשתמש בה לעולם...

Math

Math הוא אובייקט גלובלי שמאפשר לבצע פעולות הקשורות ל... מתמטיקה. אל תופלו מהכיסא פה. יש לאובייקט לא מעט מתודות ותכונות, שאחת מהן (זאת) הכרנו בתחילת הפרק. רוב המתכנתים פוגשים את האובייקט זהה כאשר הם רוצים ליצור מספר רנדומלי, למשל כשמותיהם למשתמש שם משתמש חדש או כאשר מדמים משתמש, משחקים, אНИמציות 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 חלקיק הזמן ויוצר מספר רנדומלי. אבל התהילה הזה רנדומלי מספיק כדי שייתקבל מספר אקרי. נשאלת השאלה, איך לוקחים את המספר הזה והופכים אותו למספר הרנדומלי שרוצים לקבל – למשל מספר מ-1 עד 10. האמת היא זהה די קל – רק צריך להכפיל את המספר שיוצא ב-10 ואז לעגל את התוצאה. למשל, אם המספר הרנדומלי שיצא הוא 0.9181803844895979 מכפילים אותו ב-10 ומעגלים כלפי מטה. התוצאה תהיה 9. איך מעגלים? באמצעות תכונה נוספת של `Math` שמשמשת ליצור עיגול, שנראית `round`. כל:

```
let result = Math.random();
result *= 10;
result = Math.round(result);
console.log(result);
```

בגרסאות ישנות יותר של ג'אווהסקריפט, לפני שהייתה קיימת אופרטור `*` שעליו למדנו באחד הפרקים הקודמים, היו משתמשים במתודת `pow` (קיצור של power) לביצוע חזקה. המתודה קיבלה שני ארגומנטים, המספר שעליו החזקה פועלת ומספר החזקה. בדוגמה הבאה מחשבים את המספר 2 בחזקת 3 (התוצאה היא 8):

```
let result = Math.pow(2, 3);
console.log(result); // 8
```

Date

התאריכים בג'אוהסקריפט מחושבים כמלילשניות (אלפיות השנה) ומותאמים לשעון של המחשב. בעידן המודרני, שבו השעונים מסונכרנים באופן אוטומטי עם הרשת, התאריך מדויק במידה ניכרת. התאריך הוא מספר המילישניות שעברו מ-1.1.1970 והוא מוסכם על כל מערכות המחשב כ-epoch time, הזמן שבו כביכול מערכת היזנוקס הראשונה החלה לפעול. כמעט כל שפות התכונות המוכרכות משתמשות במסנןה זו. חל肯 סופרות את השניות שעברו וחילקן את המילישניות. בג'אוהסקריפט סופרים מילישניות.

כשנדרשים לחשב תאריכים בג'אוהסקריפט, צריך לחשב אותם עם Date. בנגדוד לאובייקטים הגלובליים שלMdno עד כה, Date הוא פונקציה בנאיית SMAACHIM עם המילה השמורה new. על פונקציה בנאיית ועל new לMdno בפרק על this ועל new. לשם TZCOROT, new מוחזר אובייקט שלהם. הפונקציה הבנאיית Date מקבלת ארגומנט של הזמן שמעוניינים לחקור. כאשר אין ארגומנט, הזמן הוא השנה שעברה מה-epoch time. ברגע שמאתחלים את Date, נוצר אובייקט שמכoon לתאריך האתחול של האובייקט – קלומר היום, ברגע זהה בדיק.

הבה נדגים:

```
const dateObject = new Date();
let epochElapsedTme = dateObject.getTime();
console.log(epochElapsedTme);
```

כאן יוצרים date ללא ארגומנט, קלומר תאריך האתחול והוא הרגע הזה ממש. משתמשים במתודה getTime כדי לקבל את מספר המילישניות שעברו מהתאריך של ה-epoch time (עד היום. נסו ותראו! 1.1.1970)

אם רוצים לאותחל את אובייקט 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, אז 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 דקות ו-500 מילישניות. אחרי שמאתחלים את האובייקט אפשר להשתמש

ב-`getTime` כדי לבדוק כמה זמן עבר מהתאריך של ה-`epoch time` עד התאריך זהה. שזה 245,849,420,500 מילישניות בדיק.

```
const dateObject = new Date(1977, 9, 16, 14, 30, 20, 500);
let myBirthdayElapsed = dateObject.getTime();
console.log(myBirthdayElapsed);
```

זה נראה מעט אבסטרקטי. הנה נראה שימוש אמיתי באפשרות הזאת. נניח שמקבלים את גילו של לקוח ורוצים לדעת בן כמה הוא בדיק. ראשית יוצרים אובייקט עם יום הולדתנו, ואז יוצרים אובייקט של התאריך הנוכחי ומחסירים ממנו את הזמן של יום הולדתנו. זה אפשרי כי הזמן הוא במספר, מילישניות, משנת 1970:

```
const customerDateObject = new Date(1977, 9, 16);
const currentDateObject = new Date();
let gap = currentDateObject - customerDateObject;
console.log(gap);
```

יש לנו את אובייקט התאריך של הלוקח ואת אובייקט התאריך הנוכחי, ופושט מחסירים אחד מהאחר. אם השנה עכשו היא 2018 ונולדתם בשנת 1978, חישוב הגיל הוא פשוט: $2018 - 1978 = 40$. כאן במקום בשנים סופרים במילישניות, אבל קשה לומר לבן אדם, "הי אתה בן 1,305,505,055 מילישניות!" – צריך להשתמש בפורמט של בני אדם. פה נכנסת המתמטיקה לסייע, ועשה חישוב פשוט שבו מחלקים את הפער במילישניות במספר המילישניות לשיש 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:

| | |
|-----------------|-------------|
| מילישניות בשניה | 1000 |
| שניות בדקה | 60 |
| דקות בשעה | 60 |
| שעות ביוםמה | 24 |
| ימים בשנה | 365 |

מספר המילישניות בשנה הוא לפיכך: $365 \times 24 \times 60 \times 60 \times 1000 = 31,536,000,000$. קלומר `Math.round` מילישניות, שזה המון. אבל אם מחלקים את מספר המילישניות במספר זהה, מגלים את מספר השנים. מובן שיצא מספר עם הרבה ספרות אחרי הנקודה, אך בדיקות לשם כך קיימות.

נהוג לשמר תאריכים כמעט תמיד כמספר המילישניות (או מספר השניות, אז להכפיל אלף כSEGUNDOS לג'אווהסקריפט) ולא כמחרוזות טקסט. זה נראה מעט מסורבל בהתחלה, אבל בזכות האובייקט Date ממש קל לעשות את זה.

JSON

לא מעט פעמים נדרשים להעביר נתונים אל סקריפט מסוים או ממנו, למשל לקבל נתונים מרשת ולבוד אותם או לטען או לשמור קובץ אם עובדים בג'אוوهסקרייפט בצד השרת. ג'אווהסקרייפט בנזיה לבוד היטב עם פורמט נתונים שנקרא **JSON**, ראשי תיבות של JavaScript Object Notation. בגדול, הפורמט זהה הוא אובייקט ג'אווהסקרייפטי שעליו למדנו כבר בעבר, אך בשינויים קלים:

1. מירכאות כפולות בלבד.
2. מירכאות סביב כל מחוזת טקסט, גם סביב התוכנות.
3. אין פסיק בסוף התוכנה האחורה.
4. אסור לנתחם העורות בתוך JSON.
5. אין מתודות.

למשל, הקוד הבא הוא קוד ג'אווהסקרייפטי תקין המגדיר אובייקט, אך הוא אינו JSON תקין:

```
let myObject = {
    name: 'Ran',
    birthdate: 245849420500,
}
```

הקוד הבא הוא אותו קוד, אבל בפורמט JSON:

```
{
  "name": "John",
  "birthdate": 245849420500
}
```

מה ההבדל? כפי שנכתב לעיל – מירכאות כפולות סביב כל מחוזת טקסט וגם סביב כל התוכנות ואין פסיק בסוף התוכנה האחורה.

כאמור, JSON אמור להיות אינטואיטיבי לכל מי שמכיר ג'אווהסקרייפט מספיק זמן. למעשה השינויים הקלים הללו, מדובר בפורמט פשוט וקל, וכמעט כל השפות בשירותים השונים עובדות אותו, מה שהופך אותו למש נוח. אבל כדי לעבוד אותו צריך להמיר אובייקט ג'אווהסקרייפטי-JSON ולהפנ. עושים את זה בקלות באמצעות אובייקט JSON הגלובלי, שיש לו שתי מתודות – האחת הופכת אובייקט ג'אווהסקרייפט ל-JSON תקין והאחרת ממירה JSON תקין לאובייקט ג'אווהסקרייפטי שאפשר להשתמש בו:

```
let someObject = {
  name: 'Ran',
  birthdate: 245849420500
}
let JSONObject = JSON.stringify(someObject);
console.log(JSONObject) // {"name": "Ran", "birthdate": 245849420500}
```

בדוגמה שלעיל לוקחים אובייקט ג'אווהסקריפט והופכים אותו באמצעות `JSON.stringify` לאובייקט JSON מן המניין בפורמט של מחוץ, ומכאן השם של המתודה: `stringify` להפוך למחוץ. את האובייקט זהה אפשר לשגר לשרת באמצעות פקודת AJAX, לשומר על השרת או להעביר אותו בכל צורה אחרת. הוא נחשב מחוץ טקסט לכל דבר. במקרה של שרת, כל שפה יודעת להתמודד עם JSON, שהפוך דה פקטו לפורמט המרכזי של הרשת.

שימוש לב שאובייקט JSON הוא מחוץ טקסט לכל דבר. ככלمر דבר זה:

```
let someObject = {
  name: 'Ran',
  birthdate: 245849420500
}
let JSONObject = JSON.stringify(someObject);
JSONObject.name = 'Moshe'; // Will get Uncaught SyntaxError: Unexpected token o in JSON at position 1
```

לא יעבד ויירוק הודעת שגיאה. על מנת לעבוד עם מחוץ טקסט שהוא אובייקט JSON כשר למחדرين, צריך להמיר אותה לג'אווהסקריפט. את זה עושים באמצעות `JSON.parse`

באופן הבא:

```
const JSONObject = '{"name": "Ran", "birthdate": 245849420500}';
let regularObject = JSON.parse(JSONObject);
regularObject.name = 'Moshe';
console.log(regularObject); // Object {name: "Moshe", birthdate: 245849420500}
```

כאן היה אובייקט JSON שהוא בעצם מחוץ טקסט. במקרה הזה הוא הוקליד ידנית, אבל במקרים אחרות הוא יכול להגיע משרות, מסד נתונים או בכל צורה אחרת. באמצעות `JSON.parse` הוא מומר לאובייקט רגיל שעליו אפשר לעבוד כרגיל, למשל להחליף את שם הלוקו-`Moshe` **כפי שנעשה כאן**.

setTimeout

מדובר בפונקציה גלובלית שמאפשרת לעכב את הקוד או לזמן אותו. בדרך כלל בקוד רגיל אין בה שימוש, אבל בלי מעת דוגמאות משתמשים בקוד הזה. כמו כן, בקוד ג'אוּהַסְקְּרִיפְט שבודק UI יש חשיבות רבה לזמן או לעיבוב, אבל בדרך כלל השימוש בפונקציה זו נחשב לתכנות גרווע. 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.

נוספּ על כן, זו פונקציה שכאמור נמצאת בדוגמאות רבות בראש בכל מה הקשור לאספקטים מתקדמים יותר של ג'אוּהַסְקְּרִיפְט (כמו גנרטורים – שם חלק מתקדם בשפה שלא הגיע אליו בסוף הזה, אבל אפשר לגשת ל-MDN שהזוכר קודם כדי לבדוק), אחרת לא הייתה שום סיבה לכתוב עליה במיוחד. אז מומלץ להתודע לסייעת שלה ולהכיר אותה, אבל מאוד לא מומלץ להשתמש בה בקוד אמיתי בלי סיבה ממש- ממש טובה (סיבה טובה היא יצירת אнимציות או משחקים, למשל).

חמושים באזהרות האלו,בואו נלמד את הפונקציה של **timeout**. מדובר בפונקציה גלובלית שזמין נמעט בכל הנסיבות – דפדפן או סביבת שרת. איך היא עובדת? זו פונקציה שמקבלת שני ארגומנטים, פונקציה אנוונימית (הmovebraת כפונקציית חצ) שרצה מיד אחרי זמן העיכוב, בזמן העיכוב במלישניות.

הינה הפונקציה:

```
console.log('start the code');
setTimeout(() => {
  console.log('After 5 seconds');
}, 5000);
console.log('end the code');
```

פונקציית החצ, שמועברת כארגומנט הראשון, תروع מיד לאחר זמן העיכוב, שמועבר כargonment השני, יסתהים. הקוד לפיך ידפיס את מה שיש בקונסולה בשורה הראשונה – "start the code", ויריץ את פונקציית העיכוב שתחכה חמיש שניות. הסקריפט לא יעצור וירוץ את השורה الأخيرة "end of code". רק לאחר חמיש שניות, שנן 5,000 מילישניות, יוצג "After 5 seconds".

```
start the code
end the code
undefined
After 5 seconds
```

סדר הרצה

```
1 console.log('start the code');

аргумент ראשון - функция анонимная →
2 setTimeout(() => {
3   [console.log('After 5 seconds')];
}, [5000]);
аргумент второй - количество миллисекунд →
2 console.log('end the code');
```

מה שחשוב להבין הוא ששאר הסקריפט רץ, וברגע שהזמן שהוקצב יסתתיים, מה שהועבר בפונקציית החץ יירוץ מיד. לא כל הסקריפט עוצר; הרצה של שאר הסקריפט מתקיימת כרגע.

פונקציית setTimeout מחזירה מספר חיובי שבו אפשר להשתמש כדי לzechות אותה בהמשך הקוד. לאיזה צורך זה שימושי? אם רוצים מסיבה כלשהי בהמשך להשמיד את ה-setTimeout. ההשמדה של setTimeout נעשית באמצעות פונקציה מובנית אחרת clearTimeout, והיא מקבלת כารוגמנט את המספר המזהה של ה-setTimeout שאוטו רוצים להרוג:

```
console.log('start the code');
const timeoutID = setTimeout(() => {
  console.log('After 5 seconds');
}, 5000);
console.log('end the code');
console.log(timeoutID); // Some positive number
clearTimeout(timeoutID); // setTimeout will never happen
```

בדוגמה שלעיל יוצרים setTimeout בדיק כmo בדוגמה הקודמת, אבל הפעם מקבלים את ה-ID שלה ומשתמשים בו כדי להרוג מידית את ה-setTimeout, שלאulos לא יתבצע.

תרגיל:

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:

ויצרים פונקציה עם שם כלשהו. ראשית יוצרים מספר רנדומלי באמצעות `Math.random()`. המספר הוא בין 0 ל-1 (לא כולל 1). מכפילים את התוצאה ב-100. כך למשל אם המספר הרנדומלי שיצא הוא 0.512, התוצאה תהיה 51.2. כיוון שהחיר מספרים שלמים, לוקחים את המספר, מעגלים אותו באמצעות `Math.round` ומוחזירים אותו כמו כן פונקציה רגילה.

תרגיל:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד הארגומנט שהפונקציה מקבלת

פתרונות:

```
function getRandom(span = 10) {  
    let result = Math.random();  
    result *= span;  
    result = Math.round(result);  
    return result;  
}  
const randomNumber = getRandom(1000);  
console.log(randomNumber);
```

הסבר:

כמו בפתרון הקודם, רק שכאן מכפילים את המספר הרנדומלי בארגומנט שמקבלים מהפונקציה, כך שהטוווח הוא בין 0 לארגומנט.

תרגיל:

צרו פונקציה שמקבלת גיל משתמש בשנים (למשל 1980) ובחודש (למשל 1) ומחשבת אם המשתמש בן יותר מ-21. אם כן, היא מחזירה true. אם לא, היא מחזירה false.

פתרונות:

```
function verify21(costumerYear, costumerMonth) {
    const currentDate = new Date();
    const costumDate = new Date(costumerYear, costumerMonth);
    const age = currentDate - costumDate;
    const years = age / (1000 * 60 * 60 * 24 * 365);
    if (years > 21) {
        return true;
    } else {
        return false;
    }
}
console.log(verify21(1977, 9)); // true
console.log(verify21(2017, 1)); // false
```

הסבר:

যוצרים 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 שנשאר לעשות הוא לחשב את הפער. הוא מתקבל במיילישניות, וכך להסב אותו לשנים מחלקיים 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.

תרגיל:

צרו פונקציה שמקבלת מספר X ומדפיסה בקונסולה את המשפט "רצתי לאחר X שניות".

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 זה מכפילים ב-1,000 כדי לקבל מספר במילישניות ומכניסים אותו לקבוע delaySeconds. קוראים לפונקציה הגלובלית setTimeout. הפונקציה זו קיימת בכל ג'אווהסקריפט חלק מהשפה, לא צריך ליצור אותה. היא מקבלת שני ארגומנטים. הראשון הוא פונקציה שרצה אחרי מספר המילישניות שמעבירים בארגומנט השני. בארגומנט הראשון מעבירים פונקציה אונימית מסווג Hz, שעליה למדנו בפרק על הפונקציות, שכל מה שהיא עשוה הוא להדפיס בקונסולה "אחרי X שניות". שימוש בההדרסה נעשית עם תבנית שגם עליה כבר למדנו, בפרק על תבנית טקסט.

בארגומנט השני שהוא מקבלת מעבירים את delaySeconds – מספר המילישניות. זה הכל. נותר רק לבדוק את הקוד ולראות שהוא עובד.

ביטויים רגולריים

בביטוי רגולרי משתמשים על מנת לעבוד עם טקסטים, והוא פשוט טקסט שמאפשר לבצע חיפוש בטקסט אחר. לא מעט פעמים מקבלים לתוכה משתנה כמות גדולה של טקסט שרוצים לעבוד איתה – למשל לבדוק אם מילים מסוימות נמצאות או לא נמצאות שם וכו'. אם רוצים לדוגמה לבצע אימות של אימיל, אפשר להשתמש בביטוי רגולרי פשוט כדי להבין אם מדובר באימייל או לא. הדרך הכי טובה למצוא מילים, משפטיים או כל דבר אחר בטקסט היא באמצעות ביטוי רגולרי.

קשה לתכנת בכל שפה ובמיוחד בג'אווהסקריפט ולא להיתקל בביטוי רגולרי. למשל, אם משתמש בחומר סיסמה שלא מציאות לנכל מסויים או לחיפוש והחלפה של טקסט. בכל פעם שעושים פעולה מורכבת בטקסט או מנתחים אותו, צריך להשתמש בביטוי רגולרי. בביטויים רגולריים נמצאים בהמון שפונות ולא רק בג'אווהסקריפט.

בג'אווהסקריפט, ביטוי רגולרי הוא מוקף בסלאשים (לוכסנים) בלבד, אף על פי שהוא מסוג אובייקט, וככה אפשר לזרות אותו. ביטוי רגולרי יכול להיראות כך:

```
const regularExpression = /abc/;
```

כל מה שיש בין הסלאשים הוא הביטוי הרגולרי. מיד אחרי הסלASH השערון יש אופציות (modifiers). למשל:

```
const regularExpression = /abc/ig;
```

מקובל מאוד להציב מיד אחרי הביטוי הרגולרי אופציות. במקרה זה האותיות `i` ו-`g` מסמלות אופציות – וגם הביטוי הזה תקין לחלוtin. יש לא מעט אפשרויות שונות שאפשר להצמיד. הנה טבלה קצרה של אפשרויות לדוגמה:

| מה הוא עושה | התו המיצג את ה-modifier |
|---|-------------------------|
| הביטוי הרגולרי לא יתייחס להבדל בין אותיות ראשיות לרגילות. <code>c</code> יהיה זהה ל- <code>ABC</code> או ל- <code>ABC</code> וכך הלאה | i |
| הביטוי הרגולרי יתחשב בრיבוי שורות בסימן של התחילת או סוף | m |
| כאשר יש כמה התאמות בחיפוש, הביטוי הרגולרי יחזיר את כלן במקומן אחד בלבד | g |

אפשר ליצור ביטוי רגולרי גם באמצעות פונקציה בנית:

```
const regularExpression = new RegExp('abc', 'ig');
```

אם משתמשים בפונקציה בנית, אפשר להשתמש במשתנים בביטוי הרגולרי. כל הדוגמאות להלן ייעשו בדרך הראשונה והפשיטה יותר, אבל אתם מוזמנים לנסות גם את הדרך השנייה, באמצעות הפונקציה הבנית, כאשר אתם מתרגלים.

הביטוי הרגולרי הראשון הוא `/abc/` שאומר פשטוטו כמשמעותו – כל ביטוי שכולל את האותיות `abc` ללא רווח ביןיהן ולא אותיות אחרות, כולל המחרוזת פשוטה `abc`. נראה איך ג'אווהסקריפט עובדת עם ביטויים רגולריים ואחר כך עמוק יותר היביטויים ומשמעותם, ולשם כך הביטוי הבסיסי הזה יספיק. כאשר יוצרים ביטוי רגולרי, יש לו באופן אוטומטי את המתודות של הביטוי הרגולרי (בדיקות כמו `slimark` יש את המתודות של `forEach` למשל).

הmethodה הראשונה שנלמד היא `test`, שמאפשרת לבדוק שהביטוי הרגולרי נמצא במחוץ לテקסט כלשהו. נניח שרצים 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 מחוץ לテקסט שאותה בודקים. ביחסם האמיתיים, מן הסתם, מחוץ לテקסט הזו תועבר על ידי המשמש או על ידי שירות חיצוני כלשהו, אבל כאן, לשם הדוגמה, נכתב אותה לתוך משתנה קבוע.

כיוון שהמשנה `regularExpression` הוא ביטוי רגולרי, יהיה לו מתודות של ביטוי כזה. מתודת `test` מקבלת כารוגמנט מחוץ לテקסט ואם הביטוי הרגולרי נמצא במחוץ לテקסט, מחזירה `true`. זה בדוק מה שהוא עשה כאן, כיוון שיש `abc` ב-`defg`.

אם ננסה את זה במחוץ אחר, שלא כולל את הצירוף `abc`, יתקבל `false`:

```
const regularExpression = /abc/;
const textString = 'Nothing is here';
const result = regularExpression.test(textString);
console.log(result); // false
```

ובן שביטוי רגולרי יכול להיות הרבה יותר מורכב מזה. אפשר לסמן, למשל, שרצים שהביטוי יהיה נכון רק אם מחוץ לテקסט 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 לחת את הדוגמה זו ולנסות לשנות את מחזורת הטקסט לנכון דבר אחר: `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 מסתיים ב-`:`

```
const regularExpression = /abc$/;
const textString = 'defabc';
const result = regularExpression.test(textString);
console.log(result); // true
```

הביטוי הרגולרי זה יהיה תואם אך ורק את מה שנגמר ב-`.abc`. אפשר לשלב ולסמן גם התחלה וגם סוף, מה שאומר שרק הביטוי `abc` בלבד יתאים:

```
const regularExpression = /^abc$/;
const textString = 'abc';
const result = regularExpression.test(textString);
console.log(result); // true
```

כל ביטוי אחר לא יתאים, כי 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), שמשמעותו תו אחד (בלבד) מכל סוג:

```
const regularExpression = /^a.bc$/;
const textString = 'a1bc';
const result = regularExpression.test(textString);
console.log(result); // true
```

אפשר לציין גם ביטוי תו אחד לפחות באמצעות הסימן `+`. + משמעותו 1 או יותר.
וותר מזה? מהה שמשמעותו לפניו. למשל:
`+a` – הכוונה היא לאות `a` אחת או יותר.
`+.` – הכוונה היא לכל סימן אחד או יותר.

כלומר הביטוי:

`^a.+bc$`

תקף גם `l-a1bc` וגם `l-a12bc` וגם `a123bc` וכך הלאה. ולא רק מספרים נכללים כאן, אלא גם אותיות, רווחים, סימנים מיוחדים וכל דבר אחר. נקודה(.) היא סוגתו כללית. אפשר גם לציין את סוגתו. 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? שהtekסט חייב להתחיל ב-`a` ולהסתיים ב-`bc`. באמצע יש מספר (`[0-9]`) אחד או יותר (זו משמעותו `+`). אפשר להשתמש גם בתווים. למשל, כאןאפשרות גם אותיות קטנות וגם מספרים:

`^a[0-9a-z]+bc$`

הביטוי הזה מתחילה ב-`a` ומידי אחריו מופיע הביטוי `[z-9a-0]`, שמשמעותו כל אות בין `0` ל-`9` או בין `a` ל-`z`, כולל אותיות קטנות ומספרים. כמה פעמים? `"+"` – כולל פעם אחת או יותר.

אפשר להשתמש באיזה טווח שרוצים, למשל `A-Z` או `0-5`.

לא חייבים להשתמש ב-`+`, יש עוד כמה ביטויי נמות:

* – כוכבית, 0 עד אינסוף.

? – סימן שאלה, 0 עד 1.

אפשר לסכם את הביטויים הרגולריים בטבלה הבאה:

| פירוש הסימן | סימן |
|---|------|
| התחלת – מה שאמור הגיע בתחילת הביטוי | ^ |
| סוף – מה שיש בסוף הביטוי | \$ |
| כל תו | . |
| בתוך הסוגרים המרובעים יהיו טווחים של אותיות | [] |
| אחד לפחות מהביטוי המופיע לפני | + |
| אפס או ביטוי אחד מהביטוי המופיע לפני | ? |
| אפס או יותר מהביטוי המופיע לפני | * |

תחום הביטויים הרגולריים הוא עולם ומלאו, ומספרים שלמים מוקדשים לו. חשוב לדעת שיש דבר זה ומשתמשים בו לניתנו טקסט. שליטה בביטויים רגולריים היא מעבר לתחומו של ספר זה, אבל אתם חייבים לדעת שיש דבר זה ולשלוט בו באופן בסיסי ביותר, מכיוון שמדובר בחלק משפט ג'אווהסקריפט שמתכונתים משתמשים בו.

בעברית הביטויים הרגולריים קשים יותר. אי-אפשר לכתוב טקסט בעברית אלא צריך להשתמש ביצוג היוניקוד שלו. יוניקוד הוא הקידוד שבו מקובל להשתמש בשנים האחרונות והוא מונע את כל הגייריש. כל צורות הכתב בעולם נמצאות ביוניקוד, וכל אותן באן מערכת כתוב מיוצגת על ידי מספר סידורי. כך למשל האות א' ביוניקוד היא 0D05+U, וזה מה שצריך לכתוב, וכן, יש גם יוניקוד ל... ניקוד. כאמור, הנושא הזה לא מכוסה בספר זהה.

תרגיל:

נתונה מחרוזת הטקסט "ahla bahla". כתבו ביטוי רגולרי הבודק אם היא כוללת את המילה ahla.

פתרונות:

```
const regularExpression = /ahla/;
const textString = 'ahla bahla';
const result = regularExpression.test(textString);
console.log(result); // true
```

הסבר:

יצרים קבוע בשם regularExpression ומכניסים אליו את הביטוי הרגולרי ahla, שמשמעו מחרוזת טקסט של ahla. הביטוי הרגולרי מוקף ב-/. כך מסמנים ביטויים רגולריים. לאחר מכן יוצרים קבוע שבו יש את מחרוזת הטקסט שאותה בודקים. בשלב הבא הוא להשתמש במתודת test. כיוון שלכל ביטוי רגולרי יש אותה באופן אוטומטי, יש את המתודה הזו גם ל-regularExpression שמניל ביטוי רגולרי. המתודה הזאת מקבלת טקסט כארגומנט, וזה בדיק מה שהועבר לה.

תרגיל:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אות גדולה אחת לפחות.

פתרונות:

```
const regularExpression = /[A-Z]+/;
const textString = 'passworD';
const result = regularExpression.test(textString);
console.log(result); // true
```

הסבר:

הביטוי הרגולרי הוא מה שחשוב פה. הסוגרים המרובעים מציננים טווח: A-Z. המשמעו כל אות גדולה שהיא. כמה אותיות גדולות אנחנו רוצים? אחת או יותר. לפיכך משתחשים ב-+. הביטוי:

[A-Z]+

משמעו אותן אחת לפחות מהטווח A-Z, ככלMORE אות גדולה אחת. אחרי שמנסחים את הביטוי הרגולרי, אפשר לבדוק אותו בכל מחרוזת טקסט. יוצרים אותו באמצעות // ומכניסים אותו

לקבוע בשם regularExpression. לוקחים מחרוזת טקסט (כמו בדוגמה: password) ומשתמשים במתודה test, שקיימת באופן אוטומטי בכל בייטוי רגולרי, על מחרוזת הטקסט הזו. המתודה מחזירה true אם המחרוזת מציינת לביטוי הרגולרי. כיוון שיש בה אות אחת גדולה, היא מציינת ויתקבל true.

תרגיל:

נתונה סיסמה. כתבו בייטוי רגולרי הבודק אם היא כוללת אות גדולה 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
```

הסבר:

כמו בתרגיל הקודם, עיקר הבעיה פה הוא לנתח את הבייטוי הרגולרי. במקרה זהה:

[A-Z] הטווח הוא A-Z (כלומר אות גדולה).
[0-9] הטווח הוא 0-9 (כלומר מספר).

הቢיטוי המשולב משמעו מחרוזת טקסט שיש בה אות גדולה אחת לפחות ומימיד אחריה מספר אחד. אחרי שמנסחים את הבייטוי הרגולרי, יוצרים אותו בג'אווהסקריפט בין //

ומנכיסים אותו לקבועщин regularExpression. קבוע זה, מהרגע שיש בו בייטוי רגולרי, יש את מתודת test שמקבלת כארגומנט מחרוזת טקסט, במקרה הזה את הסיסמה שהוזנה, 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 כתבם:

```
console.log(myVar);
```

ולא טורחים להגדיר קודם קודם את `myVar`, מקבלים שגיאה כזו:

`Uncaught ReferenceError: myVar is not defined`

אם משתמשים בדפדפן, רואים את השגיאה בקונסולה באופן הבא:

`▶ Uncaught ReferenceError: myVar is not defined`

הודעות השגיאה מכילה מידע גם על השגיאה וגם על המיקום שלה. עד עכשו התיחסתי אל שגיאות כאלה בלתי רצוי, אבל הרבה פעמים כן רוצים שגיאות. שגיאות והטיפול בהן הם חלק מכל מערכת פונקציה חשובה. אם למשל כתבם פונקציה שבודקת סיסמה וממשתמש McNis סיסמה בעברית בלבד, זו שגיאה שחייבים לתפוס ולטפל בה. אם כתבם פונקציה שקוראת קובץ אבל הקובץ לא שם, חייבים ליצור שגיאה עבור מי שכותב את הפונקציה.

יצירת שגיאה היא ממש פשוטה. שנייה היא אובייקט שנוצר מפונקציה בנאית ויצרים אותו כך:

```
const myError = new Error('Oy Vey');
throw myError;
```

ראשית יוצרים את השגיאה עם הטקסט המוחד לה. במקרה זהה "Oy". אחרי שיש אובייקט שגיאה אפשר בכל שלב של הסקריפט לזרוק אותו. ברגע שיש שגיאה, כל קוד שירוץ אחריה יפסיק לroz מידית, ממש כמו שגיאת הרצה מהסוג המוכר (זההוב?).
כלומר הקוד הזה:

```
function testMe() {
    const myError = new Error('Oy Vey');
    throw myError;
}
testMe();
console.log('Hello'); // Will never run
```

ההדפסה של Hello בקונסולה לא תרצו לעולם, אלא אם כן מדובר במקרה נדיר שבו חייבים להפעיל מערכת. בדרך כלל יוצרים שגיאה כחלק מהניסיונו לנהל משהו. שגיאה שמנפילה את כל הסкриיפט לא ממש תעזר כאן. בגלל זה לרוב זורקים את השגיאות לתוך блוק שנקרא **.try-catch**.

בלוק של try-catch הוא בעצם דרך לזרוק שגיאות ולטפל בהן בלי השבתה של הסкриיפט כלו. בגודל הוא מורכב מהמילה השמורה try שאחריה סדריים מסולסים, וביהם הקוד רץ. אם לא תהיה שגיאה – טוב ויפה. אם תהיה שגיאה, קוד השגיאה ייכנס לבлок catch, שבו יש את ארגומנט השגיאה:

```
function testMe() {
  const myError = new Error('Oy Vey');
  throw myError;
}

try {
  testMe();
} catch (error) {
  console.log(error);
}
console.log('Hello');
```

```
function testMe() {
  const myError = new Error('Oy Vey');
  throw myError;
}

try {
  testMe();
} catch (error) {
  console.log(error);
}
console.log('Hello');
```

**ה-*catch* מקבל
ארוגומנט את
השגיאה שנזרקת**

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 שמוחזק ל--try רץ כרגיל והסקריפט לא קופס.

במקרה הזה, testMe תמיד זורקת שגיאה. בחרתי להדפיס את השגיאה בבלוק ה-*catch*. ההדפסה של ה-Hello, מחוץ ל-try-catch, תודפס כרגיל.

הפלט שיוצג בקונסולה

```
Error: Oy Vey
at testMe (VM552 pen.js:2)
at VM552 pen.js:7
Hello
```

כל קוד שעתופ ב-try ויזרוק שגיאה, אפילו שגיאת קומפילציה, לא יפריע למהלך התקין של הקוד ויהיה אפשר לנוהל את דרך הטיפול בשגיאה באמצעות ה-catch. האם תדפיסו הودעה למשתמש? האם תנסה להריץ מחדש את הפונקציה? הכל אפשרי.

בדרכ נכל מקובל שמודולים זורקים שגיאות וסומכים על מי שמאפייע אותם שיטפל בהן לפי מיטב הבנותו. 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 מורכב שנבנה בעזרת פונקציה בנית, מהסוג שכבר למדנו עליו. במקרה זהה, בתוך האובייקט עצמו בודקים את הסיסמה באמצעות ביטוי רגולרי. אם הסיסמה לא מכילה אות גדולה, זורקים שגיאה מסוג אחד. אם הסיסמה קצרה מדי, זורקים שגיאה מסוג אחר. כל זריקה של הסיסמה עוצרת את הסקריפט בתחום האובייקט.

אם מישמים את האובייקט כמו שצරיך, צריך לעטוף אותו ב-try וב-catch:

```
try {
    new passwordTester('12345678');
} catch (error) {
    console.log(error);
}
```

בתוך ה-catch אפשר לנוהל את השגיאה, ככלmor להדפיס אותה למשתמש, למשל. מקבלים את האובייקט של השגיאה, והטקסט מופיע בתחום אובייקט השגיאה תחת השדה message. אפשר להדפיס את הטקסט של השגיאה, כמובן, או לעשות בו מה שרוצים.

כמעט לכל מודול חיצוני או אפילו פנימי בג'אווהסקריפט יש מדיניות שגיאות שאפשר לתפוס ולנהל. לא את כל השגיאות אפשר לנוהל בצורה ממש טובה, והבחירה מה לעשות היא של המתכנן; לעיתים מעדיפים להשיבת את כל הסקריפט, לעיתים מציגים הודעה

שגיאה, לפעמים מתקנים את הקלט ומריצים שוב – מה שיש בתוך ה-`catch` הוא באחריותכם.

זו גם הסיבה שבגללה זה לא נחשב להברקה גדולה לעטוף את **כל** הקוד ב-`try-catch`. כנה רק מונעים הדפסת שגיאות, אבל מפספסים את הנוקודות החשובות של ניהול שגיאות – ניהול השגיאה הוא חשוב אם יודעים מאייפה היא מגיעה ומה לעשות אליה. אם הסקריפט מורכב, בשלבعلיוון(`try-catch`) אי-אפשר לעשות כלום עם השגיאות וקשה לדעת מהיכן הן הגיעו. לפיכך, מומלץ שתעתטו את חלקו הקוד שלכם `try-catch` רק אם יש לכם תוכנית מסודרת בנוגע למה צריך לעשות במקרה של קריישה או במקרים שאתם צופים שבהם יעופו שגיאות.

finally

כטוספת ל-`try` יש את `finally`. זה בלוק קוד נוסף יכולם להוסיף לביטוי ה-`:catch` והוא תמיד יירוץ 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 תרצו את הקוד הזה, תראו ש-`Try again!` מודפס בקונסולה, גם אם תשנו את הסיסמה למשהו שעובר. בדרך כלל משתמשים ב-`finally` על מנת לבצע עבודות ניקיון. למשל אם מבצעים טעינה של משאב, לפני ה-`try-catch` תהיה הצגה של איקון טעינה וב-`finally` תהיה מחיקה שלו.

תרגיל:

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 להסביר מילימ. 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 השורות האלה הן לב **התרגיל**:

```
const numberError = new Error('What?!? no number?? :(');
throw numberError;
```

כאן זורקים את השגיאה שיש בפונקציה. אפשר לראות שהשגיאה הזו תודפס גם אם אין כאן כלל שגיאת קומפיולציה. ג'אווהסקריפט בהחלט יכולת לחבר מחרוזת טקסט ומספר (היא פשוט תמיר את המספר לטקסט), אבל במקרה זהה, הפונקציה לא מאפשרת זה ותזרוק שגיאה.

תרגיל:

נתון משתנה שהוא מספר. כתבו פונקציה שמקבלת אותו, ואם מדובר במספר שלילי היא זורקת שגיאה. עטפו את הפונקציה שכתבתם 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:

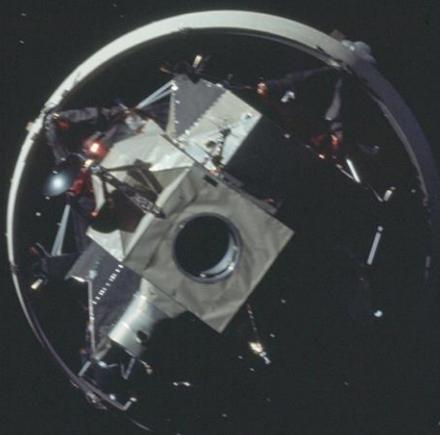
הfonקציה פשוטה למדי, היא מקבלת ארגומנט. אם הארגומנט הוא שלילי, זורקים שגיאה באמצעות:

```
const numberError = new Error('Negative Number');
throw numberError;
```

כדי לנצל את השגיאה, עוטפים את הקוד ב-try-catch. אם יש שגיאה, מאפסים את המשתנה. זה הכל. מה שיש בתחום `try` יזרוק את השגיאה אך לא יפריע למלאך התקין של שאר הסקריפט. מה שיש בתחום `catch` ינהל את השגיאה. במקרה זה דרך ניהול השגיאה היא לאפס את המשתנה.

פרק 16

מבני נתונים מסובב SET-1 MAP 1



מבנה נתונים מסוג Set-1 Map

בפרק על מערכים למדנו שאפשר לאחסן בהם נתונים. אבל יש בעיה – במערכות, המפתח הוא מספר ואי-אפשר לקבוע אותו. אפשר לאחסן מידע באובייקטים, אך לא נוח לעשות זאת כיון שקשה לעבוד עם איטרציות באובייקטים, קשה למחוק ערך לפי המפתח שלו וקשה לספור את מספר הערכים. ג'אווהסקריפט מספקת שני מבני נתונים מתוחכמים יותר לאחסן נתונים, וכן אפשר ליהנות מה יתרונות של אובייקטים אבל גם מהפונקציות של המערכים.

Map

מבנה הנתונים הראשון מאפשר להכניס ולאخזר ערכים בפורמט `key value` פשוט שבפשוטים. בנגדו לאובייקט, אפשר לעשות עליו לולאות פשוטות. אין ערכים כפולים. בגדול, הוא דומה לערך, אלא שבמקום מספרים יש שם מפתח וערך. על מנת ליצור `Map` צריכים להשתמש בפונקציה הבנאית `Map new`, שמחזירה אובייקט שיש לו את התכונות ואת המתודות של `Map`.

למשל, כך יוצרים `Map` בסיסי עם המפתחות `:phone`-`1` `city`, `username`

```
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, כאמור, כל דבר, לא רק מחרוזות טקסט כמו פה אלא גם מספרים, מערכים ואפילו `Map` אחר. בזמן היצירה אפשר ליצור את `Map` עם מפתחות. למשל:

```
let myMap = new Map([['userName', 'Ran'], ['city', 'Petah Tiqwa']]);
```

איך מzechרים את השמות? באמצעות методת `get` שיש בה ארגומנט של המפתח המתאים. ממש כן:

```
const phone = myMap.get('phone');
console.log(phone); // 6382020
```

אם רוצים לדעת כמה איברים יש במערך, משתמשים בתוכנת `size`. שימוש לב שבסונה מערכיים, כאן משתמשים ב-`size` ולא ב-`length`.

הינה דוגמה:

```
const size = myMap.size;
console.log(size); // 3
```

מחיקה נעשית באמצעות המתודה `delete`, שמקבלת כารוגומנט 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
```

שימוש לב שבניגוד למערך, שם אם מוחקים איבר האורך שלו נותר כשהיה, כאן האורך משתנה בהתאם לערכיהם.
על מנת למחוק את כל המפה, צריכים להשתמש במתודה `clear` שלא מקבלת שום ארגומנט:

```
console.log(myMap.size); // 3
myMap.clear();
console.log(myMap.size); // 0
```

אפשר להשתמש בולולאת `forEach` ממש פשוטה על מנת לעבור על כל ה-`Map`, ממש כמו מערך, אלא שכאן המפתחות הם לא מספריים:

```
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` אמרור להכיל כל מבנה נתונים שהמפתחות שלו הם לא מספריים, ויש המון אליו, מאובייקט של משתמש, דרך אובייקט של פועלה שהמשתמש עושה ועוד כל דבר בעצמו. היתרון האדריר של `Map` הוא שאפשר להיות בטוחים שיש רק

מפתח אחד מכל סוג. אין צורך ולא צריך לנהל את הדופליקציות. בנוסף על כן מרוחיקים יופי של איטרציות, ממש כמו במערכות.

Set

מדובר במבנה נתונים דומה ל-Map, אך מעט פשוט יותר, שיש בו ערכים בלבד ללא מפתחות. בניגוד למערכים, שבהם המפתחות הם מספרים, או לאובייקטים ול-Map, יש מפתחות שיכולים להיות מחרוזת טקסט רגילה, ב-Set יש רק ערכים - כלומר אין ערכים כפויים. אם מנסים להכניס ערך ל-Set והוא כבר קיים, הוא פשוט "ידرس". Set נוח מאוד לאחסון נתונים כמו תגיות, רשימת מצריכים וכל נתון שהוא שחייב בו הוא רק הערך שלו.

יצירת Set נעשית גם היא באמצעות פונקציה בנאית, כמו יצירת Map:

```
const tags = new Set();
```

מהרגע הזה יש Set שאפשר להכניס לתוכו ערכים. שימוש לב שהכנסתי את ה-Set לתוכו קבוע. מהנקודה הזאת איז-אפשר להכניס לקבוע משהו אחר, אבל מן הסתם אפשר להכניס ערכים לתוך ה-Set.

איך מכניסים ערכים? באמצעות המתודה add, שקיימת לכל משתנה מסוג Set. הארגומנט היחידי שיש בה הוא הערך:

```
tags.add('tag1');
tags.add('tag2');
tags.add('tag3');
console.log(tags); // Set(3) {"tag1", "tag2", "tag3"}
```

ל-Set יש בדיק אוטומטי איטרציות שיש למערכים ול-Map, כלומר אפשר לנחת להשתמש ב-Set.forEach. בפונקציית הקולבך של ה-foreach יש כמובן רק את הערך, כי אין מפתחות ב-Set:

```
tags.forEach((value) => {
  console.log(value); // tag1, tag2, tag3
});
```

נחמוד, נכון? אפשר לבדוק אם יש ערך מסוים ב-Set בעזרת Method `has`, שמחזירה `true` או `false`:

```
tags.has('tag1'); // true  
tags.has('Not here'); // false
```

כמו ב-Map, גם כאן מוחקם ערכיהם באמצעות `delete` ובודקים את גודל ה-Set בעזרת תכונת `theSize`. 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 Method `clear`. כן, גם כאן יש דמיון ל-Map:

```
tags.clear();  
console.log(tags.size); // 0
```

כאמור, Set שימושי מאד לאחסנת ערכים שאין צורך במקומות שלהם, כגון רכיבי מתכוון, תגיוט, שמות וכו'. Set יכול להכיל כל נתון, ולא רק מחרוזות טקסט.

תרגיל:

צרו `Map` שיכיל עיר, רחוב, מספר בית ומיקוד.

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`. שימוש לב שהשתמשתי במילה השמורה `new` על מנת להפעיל את הפונקציה הבנאית `Map`. מהרגע שיש `Map` בתוך הקבוע, אי-אפשר לשנות את הסוג שלו, אבל כמו אובייקט שנכנס לתוך קבוע – אפשר להוסיף ל-`Map` ולהוריד ממנו.

וזה בדיק מה שעושים – ברגע שהקבע `address` הוגדר כ-`Map`, אפשר להשתמש במתודה `set` על מנת להכניס לתוכו מפתחות וערכים. הדפסה של האובייקט ב קונסולה תראה את ה-`Map` במלואו.

תרגיל:

צרו `Set` של רשימת 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 באמצעות פונקציה בנית מסוג `Set`. מכניסים את ה-`Set` החדש לתוך קבוע. זה לא אומר שא-י-אפשר להכניס דברים ל-`Set` או למחוק דברים מתוכו, אבל אי-אפשר לשנות את `names` לסוג אחר. קל ונחמד.

משתמשים בMETHOD `add`, שקיימת בכל `Set` להכנסת ערכים בלבד. אתם זוכרים שב-`Set` יש אך ורק ערכים ולא מפתחות? זו הייחודה של `Set` לעומת `Map`. הצגת מספר הערכים נעשית באמצעות תכונת `size`. זה הכל.

פרק 17

תכנות אסינכרוני - הולבוקים



תכנות אסינכרוני – קולבקים

כדי להבין מה זה תכנות אסינכרוני, כדאי להבין מה זו אסינכרוניות. כשחושבים על תוכנה, בעצם חושבים על תוכנה סינכרונית. למשל התוכנה הבאה:

1. לטען את מערך A.
2. לטען את מערך B.
3. למין את מערך A.
4. למין את מערך B.

מה קורה אם שלב 1 ושלב 2 (טעינת המרכיבים) הם ארוכים? בכל הדוגמאות, מנוע היג'אומסתקריפט פשוט חיכה. שלב 1 מתבצע, אחרי זה שלב 2, אחריו שלב 3 ואז שלב 4. תכנות אסינכרוני עובד קצת אחרת. בעצם כותבים את התוכנה לפי שלבים. הרוי אין היגיון לחכות לטיענת מערך B אם מערך A כבר נמצא ואפשר למין אותו בזמן שמערך B נטען.

תוכנה אסינכרונית נראה כך:

1. טוענים את מערך A. כשהוא נטען (אם אין פעולה אחרת שמتابצת) – ממיינים אותו.
2. טוענים את מערך B. כשהוא נטען (אם אין פעולה אחרת שמتابצת) – ממיינים אותו.

קל לראות שההlixir יעל בהרבה. טוענים את מערך A ואת מערך B. ברגע שהראשון נטען, ממיינים אותו. זה לוקח הרבה זמן. לא מבצעים כלום במקביל – אבל בזמן שמתתייחסים למשהו שיטען, יכולים לעשות דברים אחרים. כדאי לשים לב שזה שונה מפעולות דברים במקביל.

תכנות אסינכרוני הוא אחד מהפתרונות החזקים של הפלטפורמות השונות שMRI צו ג'אומסתקריפט (מדפסן ועד שרף), וג'אומסתקריפט תומכת באסינכרוניות באופן מושלם, בטיח ובטח בשימושים מודרניים. כשכתבם סקריפט, יש המון פעולות שMRI צו והتوزאה שלתן מגיעה מאוחר יותר. הנה נדים זאת ונשתמש שוב בדוגמה של תוכנה פשוטה של המרת מtekudeot. הסקריפט פשוט זה מקבל קלט מהמשתמש (למשל סוג המטבח וכמוות), ניגש לשירות חיצוני כלשהו באמצעות האינטרנט, על מנת לקבל את שער ההמרה, מכפיל את שער ההמרה בכמות המטבח ומציג את התוצאה למשתמש.

| סדר פעולה | מה קורה בפעולה |
|-----------|---|
| 1 | קיבלה קלט מהמשתמש – שם המטבח וכמותו |
| 2 | גישה לשרת חיצוני כדי לקבל את שער המריה |
| 3 | קיבלה הנתון מהשרת החיצוני, חישוב הסכום והציגו למשתמש |
| 4 | מוכנים לקבל נסף מהמשתמש |

פעולה 2 דורשת זמן. בעוד פעולה 1 ופעולה 3 הן מיידיות, עבור פעולה 2 צריכים לבצע קריאה לשרת מרוחק ולהוכיח עד שתגיעה תשובה. גם אם השרת המרוחק הוא מאד-מאוד מהיר והרשת של המשמש סופר-יעילה, עדין הפעולה הזאת תיקח כמה מילישניות לפחות, ואז יש בעיה. אי-אפשר לעבור לשלב 4. האם עוצרים את פועלות הסקריפט והמשתמש "נענעל"? או שמסמיכים את פועלות הסקריפט לדברים אחרים? למשל, המשמש אומר רוצה גם לבצע המירה לשער אחר או להפעיל סקריפט אחר.

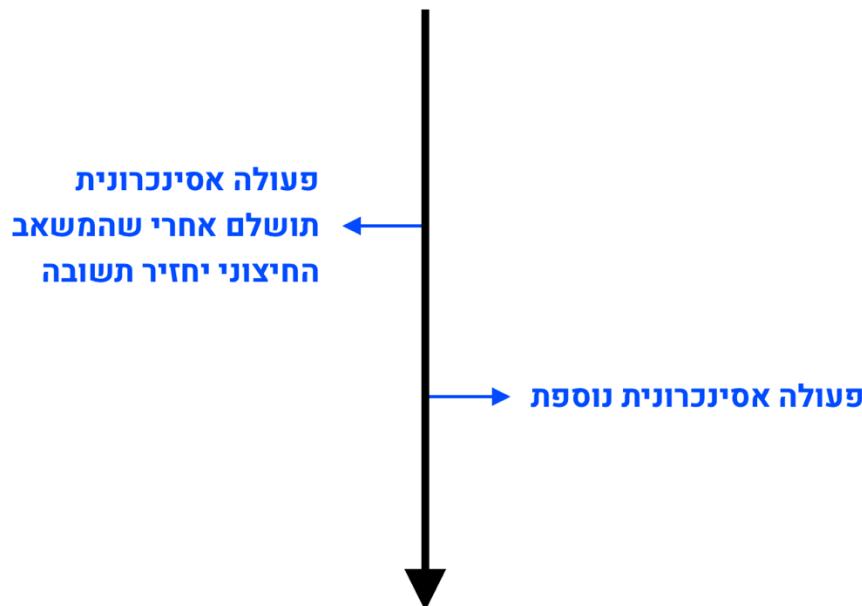
יש שתי אפשרויות:

האפשרות הסינכרונית – כל שלב נועל את השלב הבא. מ-1 עוברים ל-2. עד ש-2 לא מושלם לא עוברים ל-3 ובטח ובטח שלא עוברים ל-4. זו בעצם האפשרות הקלאסית שאותה למדנו עד עכשוו. בקוד הבא:

```
console.log('1');
console.log('2');
console.log('3');
console.log('4');
```

2 יודפס אך ורק אחרי 1-3 תמיד יודפס אחרי 2. הקוד רץ לפי סדר הכתיבה/קריאה. זה נקרא סדר סינכרוני. כל פעולה בג'אווהסקריפט היא סינכרונית וקוד תמיד ירוז מההתחלת עד הסוף.

האפשרות האсинכרונית היא מעט מורכבת יותר. אם שלב 2 חיוני רק לשלב 3, אפשר לומר לסקריפט, "שמע נא, מר בחור, שלח את הקריאה לשרת. ברגע שהשרת ייזור עם תשובה, תציג את הפלט למשתמש. אבל עד שזה יגיע, אל תחכה לי. רוץ להלא לשלב 4 ולשלבים שאחריו". ככלומר הקוד האсинכרוני לא חוסם ולא בולם את המשך הרצת הסקריפט. והוא כן חוסם את מה שצרכן.



קוד אסינכרוני מופיע סקריפטים מודרניים. באפליקציות או אתרים כיום יש המון פעולות כאלה. למשל, רישום לדיסק קשיח, קריאה למשאב חיצוני, הפעלת מצלמת וידיאו או מיקרופון. במקרים לנעול את המשטח, מאפשרים לו לעבור הלהה. בדיק שבסוגול דוקס, למשל, השמירה נעשית ברקע ואפשר לכתוב מסמך בלי הפרעה או "נעילה". בדיק כי שבפייסבוק, לדוגמה, אם רוצים ליום שיחת וידיאו, עד שהמשתמש השני מקבל את הבקשה אפשר לגלו בפייד או לעשות לijk. כל מה שנכתב לעיל נכתב בג'אווהסקריפט 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 נחזור ונסביר מה זה קולבק.

על מנת להמחיש את השיטה ניצור הדמיה של פונקציה שקורסית לשירות. הפונקציה זו תשתמש 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 עיקוב של מילישניה אחת. אפשר להעמיד פנים שהתשובה זו הגיעה משרת מרוחק. בפונקציה אמיתית, לא יהיה `setTimeout` אלא תהיה קריאה אחרת, אבל בכל מקרה הקריאה זו תיקח זמן. אפילו מילישניה אחת (שהה מעולга) היא עדין זמן.

אם מנסים לקרוא לפונקציה זו, מקבלים `undefined`. למה? כי הפונקציה עדין לא רצתה בכלל, אף על פי שמדובר בעיקוב של מילישניה אחת בלבד!

```
2 function getService() {
  setTimeout(() => {
    return 'Result from remote service'; ←
  }, 1);
}
1 const answer = getService();
3 console.log(answer); // undefined
```

לא מספיק לחזור

כלומר, כשהנמצאים בשלב 3, שלב 2 עדין לא הספיק להתרחש.

יש דרכים סינכרוניות לטפל בכך, אבל הנה נטפל בכך באופן אסינכרוני. אם רוצים שהשלב 3 יתרחש רק לאחר שלב 2 יושלם, הדרך לעשות זאת היא להכניס את שלב 3 לפונקציה ולגרום לפונקציה בשלב 2 לקרוא לה באופן אוטומטי. הדרך זו נקראת "קולבק".

איך גורמים לפונקציה `getService` להריץ פונקציה אחרת? מعتبرים לה ארגומנט שהוא בעצם פונקציה וקוראים לפונקציה זו עם מה שמניע מהשרת המרוחק:

```
function getService(cb) {
  setTimeout(() => {
    cb('Result from remote service');
  }, 1);
}
```

למדנו את זה לעומק בפרק על הפונקציות, אבל ארכיב שוב. ארגומנטים יכולים להיות פונקציות. בג'אווהסקריפט פונקציות הן אובייקטים. כמו שאפשר להעביר מחזרות טקסט, מספר, מערך וכל דבר אחר, אפשר להעביר פונקציה. ברגע שהיא בתוך משתנה, קוראים לה כמו פונקציה וגילה ומכירים לה Aiזה ארגומנט שרצים. במקרה זהה, הארגומנט הוא סתם מחזרות טקסט מומצאת. בקריאה לשרת הוא יכול להיות מחזרות טקסט ממשועית או אובייקט מייד.

הפונקציה מקבלת ארגומנט של פונקציה וקוראת לו אק ווק כשהפעולה מסתיימת, ובמקרה זהה, אחרי מילישניה אחת:

```
function getService(cb) {
  setTimeout(() => {
    cb('Result form remote service');
  }, 1);
}
```

ה콜בק מופעל רק בסוף הפעולה

עכשו מפעילים את הפונקציה. צריך רק לקרוא ל-`getService`, כשהארוגמנט הוא הפונקציה שרצים שתעבד מיד לאחר שהפעולה של `getService` תושלם. מعتبرים כargonmnt פונקציה אונימית מסווג חוץ שתדפיס את מה שמעבירים אליה, זה הכל. שימו לב שה-`cb` זה הארגומנט שאותו מعتبرים והוא אמור 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 האנונימית
שאנו מעבירים נקראת
בסיום הפעולה**

ככה עובד קוד אסינכרוני עם קולבקים. הפעלה מתחילה כשיוצאים ממחוזות הדוגמה אל מחוזות המוצאות. נניח שיש אפליקציה שבה הלוקח מכניס מספר שקלים ובודק כמה מנויות של חברה זרה הוא יכול לקנות. צריך לעשות את הפעולות הבאות:

1. לקבל את הקולט מהлокוח – מחיר שקלים וסוג מניה.
2. לראות מה שער הדולר באמצעות קריאה לשירות חיצוני ולבדק כמה דולרים שקלים האלו שווים.
3. לקבל את מחיר המניה באמצעות שירות חיצוני אחר ולראות כמה מנויות אפשר לקנות.
4. להציג את המידע ללקוח.

כלומר, אם הלקוח רוצה לרכוש ב-40 שקל מנויות של חברת אינטל, קוראים לשירות מסוים שיראה מה שער הדולר. נניח שהוא 4, ולлокוח יש 10 דולרים. אחרי כן קוראים לשירות אחר שיראה את מחיר המניה, נניח שהוא 5 Dolars. הפלט שיוצג ללקוח הוא 2 מנויות. 40

לחילק ל-4 לחילק ל-5, או בתרגיגל: 40:4:5

איך ממשים את זה? הינה השירותים שمدמים קריאה לשער הדולר וקריאה למחיר מניה:

```
function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}

function getStockPrice(stockSymbol, cb) {
  setTimeout(() => {
    cb(5);
  }, 1000);
}
```

השירות המזוייף הראשון `getCurrencyRate` דומה לזה שתרגלנו – הוא קורא לקולבק שמעבירים לו ומחזיר 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);
}

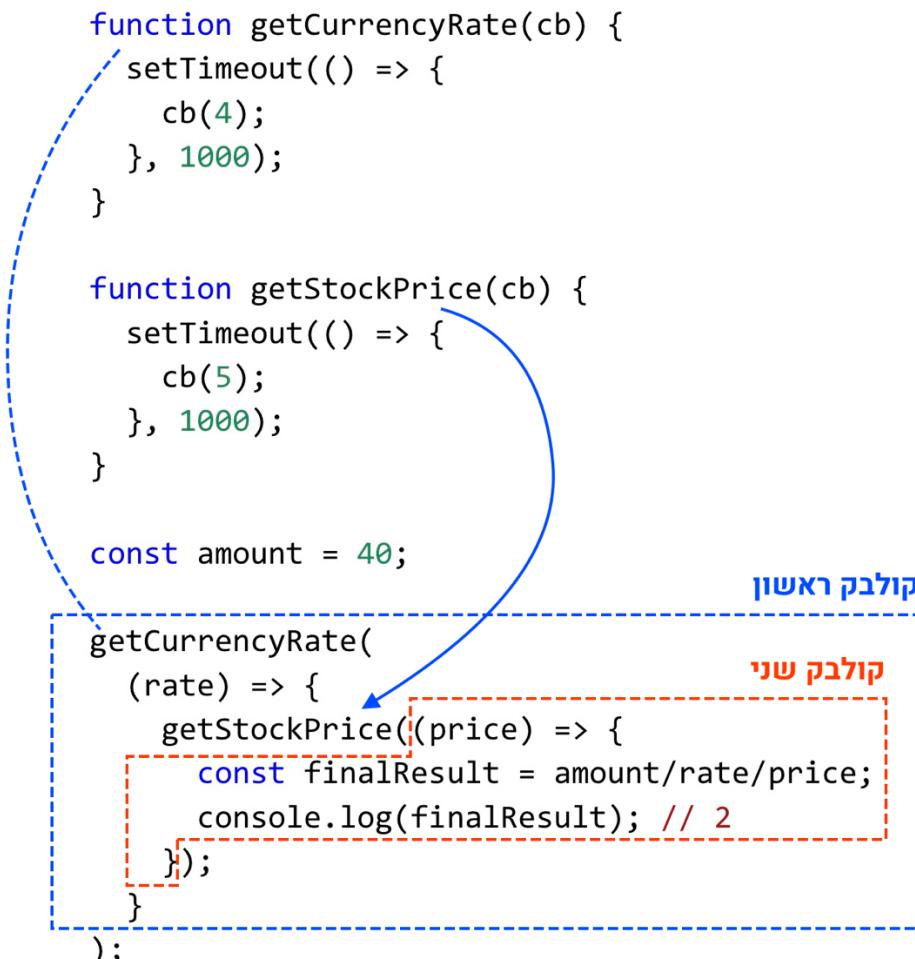
function getStockPrice(cb) {
  setTimeout(() => {
    cb(5);
  }, 1000);
}

const amount = 40;

getCurrencyRate(
  (rate) => {
    getStockPrice((price) => {
      const finalResult = amount / rate / price;
      console.log(finalResult); // 2
    });
  }
);
```

از מבצעים קריאה ל-`getCurrencyRate` כמספרים פונקציית חץ כארוגמנט. פונקציה החץ מקבל את שער הדולר מהשירות המזוייף ותקרה מיד לשירות שמחזיר את מחיר המניה. שער הדולר ומחיר המניה יסייעו לחשב כמה מנויות אפשר לרכוש. במקרה הזה – שתיים.

אפשר לראות שהמבנה הזה מסובך מדי. קולבק בתוך קולבק.



המצב הזה עוד יכול להסתיים ככל שהסקריפט מורכב יותר, וاز בהחלט אפשר לראות שלושה וארבעה קולבקים ואפילו יותר מקננים זה בתוך זה. מצב כזה נקרא **"גיהינום הקולבקים"** (callback hell). איך פותרים את זה? בעזרת promises, כפי שנלמד בפרק הבא.

תרגיל:

נתונה פונקציה `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 אונימית מסווג ח' שמקבלת ארגומנט. הארגומנט הזה יודפס. איך זה עובד בפועל? פונקציית `getCurrencyRate` מצפה לקבל ארגומנט שהוא קולבק. היא מדמה קריאה לשרת ומפעילה את הקולbak עם ארגומנט שהוא התשובה. הפונקציה האונימית שיצרתם מקבלת את התשובה ומחזירה אותה.

תרגיל:

צרו שירות מזויף המדמה קריאה לדיסק הקשיח וקריאתקובץ. השירות המזויף יקבל שם קובץ וקולבק. הוא תמיד יחזיר טקסט שבו כתוב "this is from filename", שם הקובץ יהיה השם שמוסבר לו. קראו לשירות המזויף והדפיסו את התוצאה בקונסולה.

פתרונות:

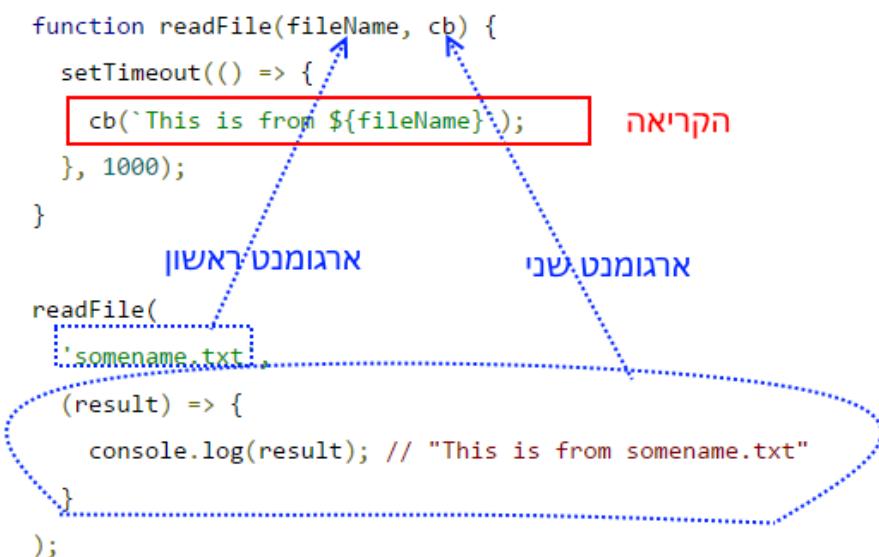
```
function readFile(fileName, cb) {
    setTimeout(() => {
        cb(`This is from ${fileName}`);
    }, 1000);
}

readFile(
    'somename.txt',
    (result) => {
        console.log(result); // "This is from somename.txt"
    }
);
```

הסבר:

השירות `readFile` הוא שירות מזויף מהסוג שכבר יצרנו. הוא מקבל שני ארגומנטים, שם הקובץ וקולבק. את הקולבק הוא מפעיל לאחר עיכוב של שנייה ומשתמש בשם הקובץ כדי להעביר אותו כארגומנט ל科尔בק.

עכשו רק צריך לקרוא לשירות ולהעביר אליו שם קובץ מזויף ופונקציית חץ אונומית שתופעל בסיום הפעולה. בפונקציית החץ האונומית רק מבצעים הדפסה של התוצאה:



Promises

כפי שראינו, קולבקים הם שיטה ייעילה מאוד לעבוד עם קוד אסינכרוני, אבל יש להם חסרונות, ובראשם הסרבול הגדול מאוד של הקוד או "גיהינום הקולבקים", וגם חסרונות אחרים – כמו למשל חוסר יכולת לתפוס שגיאות (את עניין ה-try-catch למדנו באחד הפרסקים הקודמים). כדי לנצל את השגיאות קיימים פורמייסטים (promises, "הבטחות" באנגלית).

העיקרון מאחוריהם הוא פשוט: פונקציה אסינכרונית מחזירה "הבטחה" שהיא יכולה להפר או לקיים. אם למשל יש פונקציה שקוראת לשירות חיצוני, 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 ההבטחה מתקיים ומה קורה אם לא.

איך יוצרים promises? הנה נשתמש בשירות המזמין מהפרק הקודם. השירות מחייב שנייה ואז מחזיר 4. הוא מדמה שירות שמחשב שעריו מטבעות וקורא לשירות חיצוני. הקוד הוא פשוט מאוד:

```
setTimeout(() => {
  return 4; // Return 4
}, 1000);
```

ובכן שזו יכולה להיות קריאה לשרת אחר, קריאה למAGER נתונים או לכל מקום שהוא. בסופו של דבר, מדובר בפקודה שמקבלת קולבק שפועל אחרי 1,000 מילישניות. למדנו על כך בפרק על setTimeout 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 מצליחה, ופונקציית reject, שלה קוראים אם רוצים שההבטחה תיכשל. ייצור ההבטחה נראה כך:

```
let myPromise = new Promise((resolve, reject) => {  
});
```

כאן ממש אפשר לראות את יצרת ה-Promise באמצעות פונקציה בנאית שמקבלת כารגוומנט פונקציית חץ אונונימית, שלה יש שני ארגומנטים שהם בעצם קולבקים.

כל מה שנשאר לעשות הוא להכניס את השירות המזוייף ולקרא ל-`resolve` שבתוכו. כאן רוצים למש את הבדיקה עם 4:

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(4);
    }, 1000);
  });
  return myPromise;
}
```

אפשר לראות שהפונקציה הבנאית מקבלת פונקציית חץ 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;
}
```

Annotations in Hebrew:

- פונקציית החץ מבילה שני ארגומנטים שם בעצם פונקציות
- פונקציית החץ שאודה אנו מעבירים לאובייקט הבדיקה

השירות המזוייף מחייב שנייה אחת (1,000 מילישניות) אז קורא ל-`resolve` ו מעביר את הארגומנט 4. הכל א Roz בתוך פונקציית `readFile`. הפונקציה מחזירה הבדיקה. אפשר לקרוא לפונקציה זו ו לקבל את הבדיקה.

מה עושים עם הבדיקה? אפשר בעצם לבוא ולומר, "תשמעי, הבדיקה, ברגע שתתמלאי, תריצי את הקוד הזה". איך עושים את זה? לכל הבדיקה יש מתודת `then` שמקבלת כARGINMENT ראשון פונקציה שמופעלת כשההבדיקה מתממשת:

```
readFile().then((result) => { console.log(result) });
```

הקוד הזה רץ בכל פעם שההבדיקה הנמצאת ב-`readFile` מתממשת. בואו נמחיש את זה באמצעות דוגמה נוספת: הפעם אFILEO בלי `setTimeout`: יוצרים פונקציה שמחזירה הבדיקה. שמה `readFile`, והוא לא מקבל שום ARGUMENT. מה שהוא

עשה הוא ליצור הבטחה. בתוך הבטחה, במקומות שונים לקרווא לשירות זה או אחר, פשוט עושים ומקיימים את הבטחה.

כשקוראים לפונקציה, משתמשים ב-`then` ומעבירים כארגומנט פונקציה שמאזינה להבטחה. ברגע שהוא תקין, היא תפעל. כך נראה הקוד:

```
function readFile() {
  let myPromise = new Promise(
    (resolve, reject) => {
      resolve('Promise resolved!')
    }
  );
  return myPromise;
}
readFile().then(
  (result) => {
    console.log(result);
  }
);
```

הشرطוט הבא ימחיש טוב יותר את העניין. בפונקציית `readFile` יוצרים את הבטחה. הבטחה מקבלת ארגומנט של פונקציה חז שבתוכו מקיימים אותה ומחזירים תגובה. בחלק השני קוראים לפונקציה:

```
function readFile() {
  let myPromise = new Promise(
    (resolve, reject) => {
      resolve('Promise resolved!')
    }
  );
  return myPromise;
}
```

```
function readFile() {
  let myPromise = new Promise(
    (resolve, reject) => {
      resolve('Promise resolved!')
    }
  );
  return myPromise;
}

readFile().then(
  (result) => {
    console.log(result);
  }
);
```

מה שחשוב להבין הוא שיש לשימוש בהבטחות שני חלקים – החלק הראשון הוא יצירת הפונקציה שבה יש את הבטחה וכטיבת הבטחה, כולל החלק שבו מבצעים `resolve` או `reject` את הבטחה. החלק השני הוא ההאזנה, באמצעות `then`, לפונקציה שמחזירה הבטחה.

כמו שאפשר לקיים הבטחה, אפשר גם להפר אותה. למשל אם השירות מחזיר שגיאה כי השירות לא נמצא, כי מסד הנתונים נפל או מכל סיבה אחרת. הפרת ההבטחה נעשית באמצעות המתוודה `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 ההבטחה אחריו שתי שניות, מפירים אותה בgesot. שימו לב שהכול נראה אותו דבר: בנית הפונקציה שעוטפת את ההבטחה והשימוש ב-`setTimeout`. השוני העיקרי כאן הוא השימוש ב-`reject`. וכך מפירים את ההבטחה. גם פה אפשר לשלוח ארגומנט שייתפס בפונקציה שකולטת את ה-`reject`.

הפונקציה שתופסת את הפרת ההבטחה נכנסת כארוגמנט השני של פונקציית `then`. הערה: בקוד מורכב מומלץ לעשות `reject` עם אובייקט שגיאה מסודר, אחרת לא מקבלים לוג שגיאות מסודר וזה עלול לבלב.

```

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!
);

```

הfonקציה הראונה תרוץ במקרה וההבטחה תקיים
הfonקציה השני תרוץ במקרה וההבטחה תופר

הfonקציה הראונה מתממשת רק כשההבטחה מקיימת. במקרה זה – זה לעולם לא יקרה.

הfonקציה השנייה היא חידוש פה: מעבירים אותה כארוגומנט השני של the-then והוא תפעל אך ורק כאשר ההבטחה תופר. במקרה זה היא מדפיסה את מה שהוא קיבלת בקונסולה בתוספת המילים Bad error. כיון שבഫוטה ההבטחה מחזיר את המילים "Bad error: ERROR! ERROR!", מקבלים בקונסולה את המשפט:

"Bad error: ERROR! ERROR!"

בשימוש בהבטחות צריך לזכור לטעות את כל המקרים, כדי שלא יקרה שהקוד פשוט יפסיק לroz מפני ששכחתי להשתמש ב-resolve/reject באחד ממשפטי התנאי.

שרשור הבטחות

אחד הפיצ'רים החזקים ביותר בהבטחות הוא יכולת לשרוך אותן – כאמור ה-`then` יכול להחזיר `promise` שיתפס על ידי `then` שימושו אליו. ראשית נשאלת השאלה – למה צריך את זה? התשובה פשוטה: פעמים רבות פונקציה לשרת, ואת התגובה שמקבלים ממנה שולחים לשרת אחר או שומרים במסד הנתונים או כל דבר אחר. שרשור הבטחות או פונקציות אסינכרוניות הוא נפוץ מאוד.

בדוגמה הבאה נראה איך עושים את זה. יש שתי פונקציות שממשות ההבטחה. אחת היא `writeNumber` שתמיד מקיימת את ההבטחה ומחזירה 100, והאחרת היא `getNumber` שבודקת את המספר. אם המספר הוא 100, היא מקיימת את ההבטחה ומחזירה OK. כאן אפשר לראות את התלות – הfonקציה `writeNumber` תלויות בפונקציית `getNumber` שתחזיר לה את המספר. הן מממשות כמו כל הבטחה שכבר ראינו, אבל מה מיוחד בשרשור הבטחות הוא מי שקורא לפונקציות הללו:

```
function getNumber() {
    let myPromise = new Promise((resolve, reject) => {
        resolve(100);
    });
    return myPromise;
}
function writeNumber(number) {
    let myPromise = new Promise((resolve, reject) => {
        if (number === 100) {
            resolve('OK');
        }
    });
    return myPromise;
}
getNumber()
    .then(
        (number) => { return writeNumber(number); }
    )
    .then(
        (result) => { console.log(result) }
    )
}
```

אפשר לראות שימושים את הקראיה ל-`getNumber` כרגע ומשתמשים ב-`then` כדי לתפוס את הקראיה הראשונה. מה מיוחד פה הוא שלוקחים את התוצאה של קיום ההבטחה הראשונה (שתיהי מן הסתם 100) ובמוקם להחזיר אותה לפונקציה, מבצעים קראיה של הפונקציה השנייה עם התוצאה (ש כאמור היא 100) ומוחזירים אותה זו ההבטחה

לכל דבר ועניין אפשר לתפוס אותה עם `then`, שם מדפיסים את התוצאה,ermen הסתום היא OK.

אם跣יש זאת בעזרת דוגמה נוספת:

```
function myPromise() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise resolved');
    });
    return promise;
}
myPromise().then((data) => {
    return data + ' 1 ';
})
    .then((data) => {
        return data + ' 2 ';
    })
    .then((data) => {
        console.log(data); // promise resolved 1 2
    });
}

function myPromise() {
    let promise = new Promise(function(resolve, reject) {
        resolve('promise resolved');
    });
    return promise;
}

myPromise().then((data) => {
    return data + ' 1 ';
})
    .then((data) => {
        return data + ' 2 ';
    })
    .then((data) => {
        console.log(data); //promise resolved 1 2
    });
}
```

אפשר לראות איך `then` מחזיר תמיד הבטחה, גם אם לא קוראים לפונקציה שמחזירה את הבטחה יותר מפעם אחת. ואין תופסים בשיטה זו הבטחה שנכשלה? בעזרת מתודת `catch` שמקבלת לתוכה את מה ששולחים ב-`reject`, בדוק כמו במתודה השנייה ב-`then` הרגיל שלמדנו עליון בסעיף הקודם. הנה דוגמה ל-`catch`:

```
function myPromise() {
  let promise = new Promise(function (resolve, reject) {
    reject('promise rejected!');
  });
  return promise;
}
myPromise().then((data) => {
  return data + ' 1 ';
})
  .then((data) => {
    return data + ' 2 ';
})
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.log(error); // promise rejected!
});
```

הדוגמה זו זהה לדוגמה הקודמת למעט שני דברים: נוסף catch שתווסף ומדפיס כל error, ואת ה-promise מוחזרים כ-reject. הוספה ה-catch מאפשרת לתפוס שינויים בכל רגע נתון כשרשרים .then.

קיבוץ 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 לרשור לה `then` שיטף במצב שבו כולן עוברות או שאחת מהן לפחות נכשלה, אף על פי שסדר השילחה וההפעלה אינו מוגבט.

כלומר אם יש שלוש הבטחות, אפשר להאזין לקיום של שלושתן במת אחת. אם כולן עוברות, הפונקציה הראשונה ב-`then` מופעלת. אם אחת נכשלה, הפונקציה השנייה מופעלת, ממש כמו בהازנה להבטחה בודדת, ואין זה משנה אם שאר הבטחות קיימו או לא:

```
function myPromiseA() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise # 1 resolved');
    });
    return promise;
}
function myPromiseB() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise # 2 resolved');
    });
    return promise;
}
function myPromiseC() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise # 3 resolved');
    });
    return promise;
}
Promise.all([myPromiseA(), myPromiseB(), myPromiseC()]).then(
    (results) => { console.log(results) }
    // ["promise # 1 resolved", "promise # 2 resolved", "promise # 3 resolved"]
)
```

כאן נוצרו שלוש פונקציות שמחזירות הבטחה פשוטה. יוצרים מערך של שלוש הבטחות שלahan. שימושו לב שבמערך קוראים לפונקציות כדי לקבל את הבטחות שלahan. המערך הוא של הבטחות ולא של פונקציות. את המערך מכנים עם `Promise.all` ואפשר להשתמש

עכשו ב-`then`. כיוון שיש כמה הבטחות וכמה תוצאות לקיום ההבטחות, הכל נכנס למערך של `results`. זה השוני היחיד.

קידוז הבטחות מתקדם

הבעיה ב-`all` היא שאם הבטחה אחת נכשלה, כל הבטחות נכשלות. ה-`Promise.all` מחזיר שנייה.

```
function myPromiseA() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 1 resolved');
  });
  return promise;
}

function myPromiseB() {
  let promise = new Promise(function (resolve, reject) {
    resolve('promise # 2 resolved');
  });
  return promise;
}

function myPromiseC() {
  let promise = new Promise(function (resolve, reject) {
    reject('promise # 3 resolved');
  });
  return promise;
}

Promise.all([myPromiseA(), myPromiseB(), myPromiseC()]).then(
  (results) => { console.log(results) } // Uncaught (in promise)
promise # 3 resolved
```

אם אני רוצה שלא יהיה כשלון, גם אם הבטחה אחת, שתיים או יותר לא מתקימות, אני משתמש בפקודת `allSettled` שזהה לחלוון `all` אך בניגוד לו, היא מחזירה דיווח מלא על כל ההבטחות – בין אם יש כשלון ובין אם ההבטחה מתקימת. ניתן לראות את זה בклות אם תיקחו את הדוגמה הקודמת ותחליפו בין `all`-`allSettled`:

```

function myPromiseA() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise # 1 resolved');
    });
    return promise;
}
function myPromiseB() {
    let promise = new Promise(function (resolve, reject) {
        resolve('promise # 2 resolved');
    });
    return promise;
}
function myPromiseC() {
    let promise = new Promise(function (resolve, reject) {
        reject('promise # 3 resolved');
    });
    return promise;
}
Promise.allSettled([myPromiseA(), myPromiseB(), myPromiseC()]).then(
    (results) => { console.log(results) }
    /*[{"status": "fulfilled", value: "promise # 1 resolved"},  

     {"status": "fulfilled", value: "promise # 2 resolved"},  

     {"status": "rejected", reason: "promise # 3 resolved"}, */  

)

```

ישנן עוד פקודות של קיבוץ הבטחות. `any` היא פקודה שמחזירה מבחן שורת ההבטחות את ההבטחה הראשונה שמתקיים ואלה בלבד. `race` היא פקודה המחזירה את ההבטחה הראשונה שמתקיים או נכשלת.

תרגיל:

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) });
```

הסביר:

החלק הראשון הוא ייצרת ה-setTimeout, שמקבל שני ארגומנטים. הראשון הוא מה שיש לעשות אחריו פרק הזמן שהוגדר לו והשני הוא פרק הזמן במלשנות. 2,000 מילישניות. הן שתי שניות.

החלק השני הוא ייצרת פונקציה שמחזירה הבטחה. יוצרים פונקציה בשם secondsPromise. בתוכה יש הבטחה בשם secondsPromise.waitTwoSeconds.

החלק האחרון והקשה ביותר הוא ליזוק תוכן בתוך הבטחה. ההבטחה נוצרת באמצעות פונקציה בנאיות של Promise. היא מקבלת ארגומנט אחד שהוא פונקציית קולבק עם שני ארגומנטים, resolve ו-reject. בתוך הקולבק מוכניסים את ה-setTimeout וקובעים שההבטחה תקיים ברגע שתשתי השניות יעברו. הבדיקה פשוטה יותר. קוראים לפונקציה ואז מואזינים עם then. היא מקבלת ארגומנט אחד, פונקציה אחת שפועלת ברגע שההבטחה מתקינה. כל מה שצריך לעשות הוא להאזין למה שמחזירים ב-result ולחדרים אותו בקונסולה:

```
function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Two seconds passed!');
    }, 2000);
  });
  return secondsPromise;
}
waitTwoSeconds().then((result) => { console.log(result) });
```


תרגיל:

צרו פונקציה `checkIfNumberOK` שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזיר הבטחה מקוימת עם מחוזת הטקסט `Number OK`. אם המספר קטן מ-10 או שווה לו, תחזיר הבטחה מופרת עם מחוזת 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
);
```

הסבר:

הfonקציה `checkIfNumberOK` היא פונקציה רגילה המקבלת מספר כארגומנט. בתוך הפונקציה יוצרים הבטחה כאובייקט `Promise` עם הפונקציה הבנאית. הפונקציה מקבלת ארגומנט שהוא פונקציה אנונימית מסוג חצ, ובה יש שני ארגומנטים, `resolve`-ו-`reject`. משתמשים בשניהם. מפעילים משפט תנאי פשוט ובודקים: אם המספר גדול מ-10, ההבטחה תקווים באמצעות `resolve`, ואם המספר קטן מ-10 היא תופר באמצעות `reject` - כמפורט עליון.

בדיקות יהיו פשוטות למדי – הריצה של הפונקציה עם המספר 11 ותפישת הצלחה או השגיאה באמצעות `then`. כיוון שה-`checkIfNumberOK` מחזירה הבטחה, אפשר להשתמש ב-`then` שמקבל שני ארגומנטים, ארגומנט של הפהה וארוגמנט של הצלחה. מן הסתם, כשמעבירים 11, ההבטחה מקוימת והfonקציה הראונה שמעבירים ב-`then` תופעל. כשמעבירים מספר קטן יותר מ-10, ההבטחה מופרת והfonקציה השנייה שמעבירים ב-`then` תופעל.

פונקציית `async`

השיטה השלישית לתכנון אסינכרוני לא שונה כל כך מהשיטה של הבטחה. למען האמת מדובר בהרבהה שלה. פונקציית `async` מאפשרת לטפל בהבטחות ללא `then` או ליתר דיוק מאפשרת לנתח קוד אסינכרוני בצורה סינכרונית. אם יש שירות שמחזיר `promise`, אפשר לבחור אם לעבוד עם `then` כמו בפרק הקודם או בפונקציה אסינכרונית עם המילה `async`. השמורה `.asycn`.

לשם הדגמה משתמש בדוגמה של הקריאה המזוייפת שהשתמשנו בה קודם. הקריאה 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 מזוייפת של 4 אחרי שנייה אחת. שימוש לב שהיא מחזירה `promise`. אם היינו רוצים להתחבר אליה, היינו צריכים לקרוא לה וזו להשתמש ב-`then` כדי לתפוס את ההצלחה. אבל אפשר לעשות את זה עם `async` באופן פשוט. ראשית מגדירים פונקציה אסינכרונית, כלומר הפונקציה הזו מרצה קוד שיכול לזרע בזמן אחר. ההגדרה נעשית באמצעות המילה השמורה `async`. בתוך הפונקציה זו אפשר לקרוא לפונקציות המחזירות הבטחה כפי שרוצים. פונקציות המחזירות הבטחה יקבלו `await` ליד הקריאה שלהן, והסקריפט בתוך ה-`async` יעצור ויתמן להן. קוד מוחוץ לפונקציה ירוז כרגע. כך מרווחים אסינכרניות אבל עם קוד ברור יותר. למשל:

```
async function main() {
    let result = await readFile();
    console.log('result' + result);
}
main();
```

בדוגמה יוצרים פונקציה שמוגדרת כאסינכרונית. שימוש לב למילה השמורה `async` לפני `function main()`. בתוך הפונקציה זו, שיש בה `async`, אפשר להשתמש במילה `await` שקוראת לפונקציה המחזירה `promise`. שימוש לב ש-`readFile` לא שונה

מפונקציה אחרת שמחזירה הבטחה, בדיק כפי שלמדנו בפרק הקודם. פונקציות `async` פשוט נועדו לנהל 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 هذا לא חוזרת
הפונקציה לא ממשיכה

לא יritz עד שה await
לא יושלם

כפי שב-`then` יש שני ארגומנטים – אחד שמתקיים כאשר ההבטחה מקוימת ואחד שמתקיים כאשר ההבטחה נכשלה – גם ב-`async` אפשר לתפוס את הכישלון. איך? באמצעות `try-catch` רגיל, שעליו למדנו בפרק על טיפול בשגיאות. בדוגמה הבאה יוצרים שירות ש תמיד נכשל ותופס את השגיאות:

```
function readFile() {
  let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('error :(');
    }, 1000);
  });
  return myPromise;
}

async function main() {
  try {
    let result = await readFile();
  } catch (error) {
    console.log('An error occurred: ' + error); // An error occurred:
    error :(
  }
}
main();
```

שירות `readFile` הוא שירות שתמיד נכשל. הוא מנסה שנייה ועושה `reject`. היישום שלו הוא בדיקת כמו כל שירות אחר שמחזיר הבטחה ומקיים אותה או לא מקיים אותה. מה שמשמעותו הוא שה-`asynchronous` שנראה `main` זהה לדוגמה הקודמת, למעט ה-`try-catch`. ה-`try-catch` יקרה כאשר הבטחה תקיים (ובדוגמה זו לעולם לא) וה-`catch` יתקיים אם הבטחה לא תקיים (בדוגמה זו תמיד כי `throw` עושים `reject`). מה שקרה הוא שמודפסת שגיאה בקונסולה. אפשר כמובן לקבוע כל התנאות אחרות.

מובן שאין בעיה לקבוע כמה `await` בתוך פונקציית `async` אחת. מה שצריך לזכור, וגם לתרגל, הוא שבסוףו של דבר מדובר בצייפוי של סוכר על גבי ה-`promises`. במקום להשתמש ב-`then` שעלול להיות קצת מסורבל, משתמשים ב-`async`. אלו שתי דרכים שונות לעבוד עם שירות זהה שמחזיר הבטחה.

תרגיל:

בדוק כמו בפרק הקודם, צרו פונקציה `checkIfNumberOK` שמקבלת מספר ומחזירה `Number`. אם המספר גדול מ-10, תחזיר הבטחה מקיימת עם מחוזת הטקסט `is OK`. אם המספר קטן מ-10, תחזיר הבטחה מופרת עם מחוזת 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 מצליח 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 ומחייב הבטחה. אם המספר גדול מ-10, הוא מקיים את ההבטחה. אם לא, הוא מפרק אותה.

מה שמעניין הוא איך שקוראים לשירות: יוצרים פונקציית `async` בשם `checkMyNumber` שמקבלת מספר. בתוכה יש `try-catch`. בתוך ה-`try` קוראים לשירות ולא שוכחים להשתמש במילה השמורה `await` כדי להראות שמדובר בשירות אסינכרוני שיש ללחכות לו. אם ההבטחה מקיימת, מה שיש בתוך ה-`try` רץ. אם לא, מה שיש בתוך ה-

רץ. עכשו כל מה שנותר הוא לקרוא לפונקציה `checkMyNumber` עם המספר המתאים.

תרגיל:

נתונים שלושה שירותים שונים המחזירים הבטחות:

```
function myPromiseA(number) {
    let myPromise = new Promise((resolve, reject) => {
        resolve('Promise A success!');
    });
    return myPromise;
}
function myPromiseB(number) {
    let myPromise = new Promise((resolve, reject) => {
        resolve('Promise B success!');
    });
    return myPromise;
}
function myPromiseC(number) {
    let myPromise = new Promise((resolve, reject) => {
        resolve('Promise C success!');
    });
    return myPromise;
}
```

כתבו 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 לסמך שהיא אסינכרונית משתמשים במילה השמורה `async`. ברגע שמשתמשים במילה זו, אפשר להשתמש ב-`await` לפני כל פונקציה שמחזירה הבטחה. שימוש זה יבטיח שהשורה הבאה בפונקציה תחכה. כך למשל:

```
results[1] = await myPromiseB();
```

ת្រוץ רק אחרי שההבטחה של myPromiseA תتمלא.

```
results[0] = await myPromiseA();
```

נותר רק לקרוא לשירותים השונים לפי הסדר ולהדפיס את התוצאה. שימוש לב שזרימת הקוד מחוץ ל-`callMyPromises`.

פרק 18

AJAX



AJAX

ההגדרה היבשה קובעת ש-AJAX הוא ראשי תיבות של Asynchronous JavaScript and XML. בעבר זה גם היה נכון. בגדול מדובר בשם כולל לדרך לתקשר עם שרתוי אינטרנט באמצעות ג'אווסקריפט ופרוטוקול HTTP. זה נשמע מעט מצחיק, כי ג'אווסקריפט היא שפה שחיה בראשת. אבל כשוחבים על כך לעומק, ג'אווסקריפט לא יהיה ברשות. ג'אווסקריפט היא שפה שחיה בדף ונטענת מיד כשהדף נטען. אם התקנתם את סביבת הבדיקה שהסבירתי עלייה באחד הפרקים הראשונים, אתם יודעים שבכל פעם שמבצעים טעינת דף, קוד הג'אווסקריפט רץ זהה. הקוד רץ עם המידע שמכניסים לו. הוא יכול לחת את המידע מה-DOM או ממשתנים שמגדירים לו, אבל מקור המידע אחד – האתר שהדף טוען.

עם AJAX אפשר לגשת לכל אתר שהוא ולקחת ממנו מידע. לצורך העניין אפשר אפילו להיכנס לאתר חדש, לשאוב את ה-HTML שלו ואז, באמצעות ביתוי רגולרי (שעליו למדנו בפרק המוקדש לביטויים כאלה), לחת את הכתובת ולהציג אותה למשתמש. מה שדף או, נכון יותר, המשתמש בדף יוכל לעשות, גם ג'אווסקריפט יוכל לעשות. אבל בדרך כלל לא משתמשים ב-AJAX כדי לפנות לאתרים המיועדים לבני אדם, אלא כדי לפנות לנתחות אינטרנט שמחזירות אובייקטי JSON, שקל לעבוד איתם בג'אווסקריפט. אם תפעילו את הדף ותנסו להיכנס לנתחה זו, גם אתם תראו את אובייקט ה-JSON. כאמור, AJAX הוא בדיקת דף שמשגר נתונים, אבל עושים את זה באמצעות קוד.

הבה נדגים. יש בראשת שירותים רבים המספקים מידע, למשל אתר המספק נתונים מג אויר או שערים של מטבחות חוץ. אלו אתרים המספקים API, זהה לראשי התיבות של Application Programming Interface, ובעצם המשמעות היא שמי שימצא בהם תועלות הם בעיקר סקריפטים של AJAX. אפשר למצוא רשימות של API וכך בראשת בחיפוש אחד האתרים המרכזיים רשיימה של שירותים כאלה נמצא בכתבota public API data sites הבאה:

<https://github.com/toddmotto/public-apis>

אפשר למצוא שם עשרות אתרים המספקים API. כאן נתרgal בשירות שנקרא "בדיקות צ'אק נוריס". השירות אינו דורש היזמות, רישום או CAB ראש ובעצם כל מה שצריך לעשות הוא לשלוח בקשה אל:

<https://api.chucknorris.io/jokes/random>

כדי לקבל בדיחה רנדומלית של צ'אק נוריס.
קל לבדוק אותו! פשטן היכנסו לקישור באמצעות דפדפן רגיל. תוכלו לראות שאתם מקבלים ממש JSON עם מידע. ייתכן שהעיצוב יטעה אתכם (בכroom התוצאה מוצגת כמחרוזת טקסט) אך זה נראה כך:

```
{
  "category": null,
  "icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
  "id": "uDjkIgHpQhe7kv8xoyJm6g",
  "url": "http://api.chucknorris.io/jokes/uDjkIgHpQhe7kv8xoyJm6g",
  "value": "Chuck Norris is currently suing NBC, claiming Law and Order are trademarked names for his left and right legs."
}
```

ובן שבעור משתמש רגיל זה הקצת בעיתוי, אבל לסקרייפט שמשגר בקשה AJAX זה מעולה. יוצרים בקשה AJAX אל האתר הזה ומציגים את המידע.
נשאלת השאלה "איזה?" בג'אווה סקריפט המודרנית יותר מקובל להשתמש ב-fetch, פונקציה גloballית שקל להשתמש בה וקיימת בסביבת הדפדפן. הפונקציה זו מחזירה promise שהוא התוצאה. מהתוצאה בוררים את סוג הנתון שרוצים ומהציגים אותו, ואיתו עושים מה שרוצים, ובמקרה של שירות הבדיקות – מדפיסים את הבדיקה.

השימוש ב-fetch:

```
fetch('https://api.chucknorris.io/jokes/random')
```

פשוט מאד, נכון? הפונקציה זו מחזירה promise עם התגובה מהשרת. ב-promise ניתן להשתמש בפරקים הקודמים. עכשו ציריך רק להחליט מה עושים אם ה-promise מתקיים. במקרה זה אנחנו רוצים את ה-JSON, סוג המידע שה-API עובד אליו. זו הסיבה שאנו משתמשים במתודת `json()` הגדינה לאובייקט התגובה `response`. יש סוגים מתודות שונים לשוגרי מידע שונים שמתקבלים מה-`response`. אם למשל ה-API היה מחזיר תמונה, היה נדרש BLOB שהוא סוג מידע אחר שלא נמצא בו :

```
fetch('https://api.chucknorris.io/jokes/random')
  .then((response) => {
    return response.json();
  })
```

פשוט, נכון? עכשיו אפשר לשרר 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 בתכונות אסינכרוני בכלל ובתכונות מבוסס `promises`, לא צריכה להיות לכם בעיה בהבנת הסינטקס הזה: `fetch` מחזירה promise עם סוג המידע, ויש לבחור את הסוג ולהחזיר אותו, אז לעשות בו מה שרצים.

מתודות של HTTP וארגומנטים נוספים

השימוש שהראיתי עד כה הוא שימוש בסיסי בפורמט GET. פרוטוקול הרשות לא נלמד בספר זה, אך אפשר לציין בקצרה שבפרוטוקול אינטרנט יש כמה סוגי בקשות. הבקשה הנפוצה היא בקשה GET, לקבל נתונים. זו הבקשה שמבצעים כאשר ננסים לאתר אינטרנט למשל. מתודות אחרות הן למשל POST – שבה משתמשים כדי לשלוח נתונים (למשל כאשר מלאים טופס באינטרנט, הטופס נשלח בבקשת POST), PUT (לעדכן נתונים) או DELETE (למחיקת נתונים).

אם רוצים לשלוח בקשה POST, אין קל מזה. צריך להעביר ארגומנט נוסף ל-`fetch`, שבו מגדרים את הנתונים ששולחים כשהמתודה מבצעת את הקריאה לשרת. במקרה הזה body method. זה מה שנשלח לשרת. מה שמתקבל הוא `response` ואני מחליטים את `then` ב-`response` באמצעות המתודה `json` שמחזירה הבטחה שאوهاן תופסים השלישי והאחרון.

```
fetch('https://jsonplaceholder.typicode.com/posts/',
{
  method: 'POST',
  body: { title: 'MyTitle' }
})
.then((response) => {
  return response.json();
})
.then((jsonObject) => {
  console.log(jsonObject); // { id: 101 }
```

});

בדוגמה זו שלוחים בקשה POST עם אובייקט אל הכתובת של השירות: GET, אלא שכאן נדרש להביר ארגומנט שני שמספרט את סוג הבקשת וכמוון את המידע. באותו אובייקט אפשר גם להגיד headers למשל. על מנת להבין את סוג הבקשות יש צורך במידע על HTTP שאינו נלמד בספר זה, אבל הפרקטיקה היא די פשוטה. כאמור, fetch הוא פשוט מאוד להבנה אם מכירים היטב תכונות אסינכרוני promise.

תרגיל:

נתון ה-API שכתובתו היא – [הדף](http://jsonplaceholder.typicode.com/posts/1)
את הכתובת המתבקשת מהגישה לכותבת זה.

```
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 שמתקיים אובייקט JSON מסווג זהה:

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem eveniet architecto"
}
```

כעת רוצים את ה-`title`.

ראשית יוצרים fetch אל הכתובת. זה קל ופשוט. הfonקציה `fetch` ממחישה promise שאותו תופסים עם `.then`. ה-`promise` מוחזר עם ארגומנט התגובה וצריך תמיד להחזיר את מה שרצוים מההתגובה זו, לענייננו – JSON. מוחזרים את:

`response.json`

כיוון שאפשר לשרר `promise`, צריך להשתמש בעוד `then` על מנת להציג בו את מה שרצוים, לענייננו – הכתובת `title`, שיש להדפסה.

תרגיל:

שלחו בקשה מסוג DELETE אל <http://jsonplaceholder.typicode.com/posts/1> והדפיסו "Deletion successful" במקרה של הצלחה.

פתרונות:

```
fetch('https://jsonplaceholder.typicode.com/posts/1',
{
  method: 'DELETE'
})
.then((response) => {
  return response.json();
})
.then((jsonObject) => {
  document.write('Delete successful');
});
```

הסבר:

כפי שציינו קודם יש כמה סוגים בקשות. מי שאחראי לפעול לפי הבקשות הוא השרת ובעל השירות, או המתכונת שכותב את השירות שנמצא על השירות, אומר לנו מה לשלוח. בתרגול הזה אנו נדרשים לשלוח בקשה מסוג DELETE. מה השירות יעשה איתה? זה תלוי בו (במקרה הזה, שירות הדוגמה לא יעשה דבר עם DELETE) אבל בתרגול אנו צריכים לדעת לשלוח בקשות שונות וזה המהות שלו.

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 זהה מכיל רק תכונה אחת: `method`, שהערך שלה הוא DELETE. מפה ממשיכים בדיקות כמו ב-`fetch` רגיל. מקבלים את התגובה ומעבירים את ה-JSON המתתקבל ממנה להלאה. אם מקבלים תגובה מסוימת כלשהו, סימן שהמ剔קה הצליחה.

ES6 Classes

מחלקות, קלאסים, או `classes` באנגלית, הן דרך מצוינת לארגן קוד בג'אווהסקריפט. יש לא מעט שפות תכנות 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 כאן, כולל איך הן נראהות כאובייקט רגיל.

מחלקה היא אובייקט שיש לו מפתחות. חלק מהמפתחות הם סוגים מיידע פרימיטיביים וחלקם פונקציות (ואז קוראים להן מתודות של המחלקה). הנה נבנה קלאס פשוט כדי להדגים איך זה עובד:

```
class Book {
  constructor(title) {
    this.title = title;
    this.isCurrentlyReading = false;
  }
  start() { // Public method.
    this.isCurrentlyReading = true;
    return this.isCurrentlyReading;
  }
  stop() { // Public property.
    this.isCurrentlyReading = false;
  }
}
const theShining = new Book('The Shining'); // Instance of the object.
console.log(theShining.title); // Return "The Shining".
const result = theShining.start();
console.log(result); // Return true.
```

אם זה מזכיר לכם פונקציה בנאית מהפרק על האובייקטים, אתם צודקים. זה בדוק אותו דבר והפונקציונליות היא אותה פונקציונליות, רק שהכל ארוזיפה יותר. ב-`constructor` יש את הפונקציה הבנאית עצמה, זו שרצה כאשר מרים את `new`. בתוך המחלקה מגדירים גם את המתודות.

זה הכל. בדוגמה אפשר לראות איך יוצרים מהטבנית, שהיא פונקציה בנאית בעצם (שים 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 מהמילימ השמורות `class` או `constructor` – זה בדוק מה שלמדנו קודם.

המחלקה המפורטת לעיל בעצם נראה כך מתחורי 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 טוביה ו נעימה יותר לעין.

בפרק על האובייקטים דיברתי על `get` ועל `set`, ובטע תשמחו לדעת שאפשר להגדיר אותם גם פה. אלא מה? צריך להיזהר מאד מהפניות מעגליות. הנה דוגמה שמتبוססת על הדוגמה הקודמת. במקרה זה יש שmbbia את התוכנה 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, וכן מוצגת דרך פשוטה יותר ונעימה יותר לעין לממש את מה שאתם כבר יודעים. אין פה משהו חדש. זו הסיבה שמייקמת את הפרק הזה בסוף הספר. מחלוקת היא לא וודו אלא דרך שונה שונה ונעימה לכתיבה פונקציה בנאית, ואת הפונקציה הבנאית שיצרת אובייקט עם תכונות ומתחות אתם מכירם.

אם יש לנו פונקציות בנאיות שהזורות על עצמן, אפשר ליצור פונקציה בנאית שמורישה לפונקציות הבנאיות את כל התכונות שלהן. אם למדתם תכונות מונחה עצמים, אז זה דומה – אבל רק במידה, כיוון שבג'אווהסקריפט היא לא שפה מונחת עצמאים.

הבה נמחיש זאת באמצעות דוגמה מהחיים. יש שני סוגי לקוחות – רגילים ופרימיום. לשני סוגי הלקוחות יש תכונות דומות: לשניהם יש שמות וכתובות ומethod set и get, אבל לקוחות פרימיום יש גם תכונת "הטבות". איך ממשים זהה דבר? אפשר ליצור שתי מחלקות באופן זהה:

```
class RegularClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class PremiumClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe",
address: "Tel Aviv"}
const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address:
"Petah Tiqwa"}
console.log(premium.getBenefits()); // 'A Lots of premium stuff';
```

מגדירים שתי מחלקות, PremiumClient ו RegularClient (בתחתיות מודגם שימוש בכל אחד מהן). שתי המחלקות זהות, למעט המתוודה getBenefits שנמצאת רק במחלקה .PremiumClient

באופן עקורי נאין בעיה עם הקוד הזה והוא עובד. אבל מה כוֹן הבעיה? שכפול קוד. יש לנו קוד שחוור על עצמו, וזו בעיה. למה? כי אם יבקשו מכם שינויים במבנה הלקוב (למשל להוסיף תכונה נוספת), תצטרכו לשנות את המבנה פגמיים. ואם יש יותר משני סוגים ללקוחות, תצטרכו לעשות שינויים במיקומות נוספים. שכפול קוד הוא ממשו שנדאי במידת האפשר להימנע ממנו.

בדוק בשבייל זה קיים ה-`extend`, שמאפשר ליצור מחלקה-על ולרשת ממנה את כל התכונות. במקרה הזה – ליצור מחלקה `client` כללית, ש-`RegularClient` ו-`PremiumClient` יעתיקו ממנה הכל. במחלקה `PremiumClient` צריך ליצור את `getBenefits`. כך זה נראה:

```
class Client {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class RegularClient extends Client {}
class PremiumClient extends Client {
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe",
address: "Tel Aviv"}
const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address:
"Petah Tiqwa"}
console.log(premium.getBenefits()); // 'A Lots of premium stuff';
```

אפשר לראות שהקוד נראה הרבה הרבה אלגנטי ושיין חזרות. בעצם, `extends` לוקח `Client` ומuplicatos אותו. ברגע שעושים `extends`, כל המתודות והתכונות של

נכונות ל-`PremiumClient` ו-`RegularClient` באופן אוטומטי. אפשר להוסיף להן מתודות חדשות.

זה אולי נראה לכם כמו קסם, אבל זה לא. זה בדיק מה שלמדנו בפרק על אובייקטים – רק באריזה חדשה. אם אתם מתקשים בתרגילים, כדאי לחזור שוב לפרק על האובייקטים.

מתודות ותכונות פרטיות

בג'אווסקריפט ניתן גם להגדיר שמתודות או תכונות יכולות להיות פרטיות (`private`) – כלומר שאף אחד מחוץ למתודות בקלאס יוכל לקרוא להן או להשתמש בהן. הגדרת המתודות פרטיות נעשות באמצעות הסימן `#` ואז שם המתודה או התכונה. מהרגע שהגדרנו כך מתודה הוא תכונה, לא ניתן לשנות או לקרוא לה מחוץ לקלאס.

בדוגמה הבאה, אנו מגדירים מתודה בשם `privateMethod` באמצעות הסימן `#` לפני שמה. השם שלו, שבו אנו יכולים להשתמש בקלאס בלבד, כולל את `#`. מתודות אחרות בקלאס יכולות להשתמש במתודה זו, אבל אם ננסה להשתמש בה מחוץ לקלאס, נקבל הודעת שגיאה.

```
class MyClass {
  #privateMethod() {
    console.log("This is a private method.");
  }

  publicMethod() {
    console.log("This is a public method.");
    this.#privateMethod();
  }
}

const myObj = new MyClass();
```

השימוש במתודות או מושגים פרטיים נועד למנוע שימוש לרעה בקלאסים. במקרים מורכבות יש לנו לעיתים קלאסים ואובייקטים שנוצרו מהם שינוי של התכונות שלהם עלול ליצור בעיות. למשל מספר מזהה של אובייקט מסוים.

תרגיל:

כתבו מחלוקת של מכונית המכבלת בפונקציה במבנה 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 opelwhite1200. צרו למחלוקת "מכונית" פונקציה המחזירה את מספר הזיהוי.

פתרונות:

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

הסבר:

באמצעות המילה השמורה Class יוצרים מחלוקת בשם Car. בפונקציה הבנייתית מקבלים את שלושת הארגומנטים הנדרשים ונותר modelNumber. שימו לב ל-*this* – ברגע שימושם בו, *this* זמין לכל המחלוקת. יוצרים מתודת *get* מסודרת כדי שתחזיר אותו.

למטה משמשים במחלוקת עם הנתונים. אפשר ליצור עכשו איזו מכונית שאתם רוצים. לשם השוויה – הסתכלו על התרגיל הראשון בפרק על *new* ו-*this*. זה אותו תרגיל בדיק והתשובה דומה מאד לתשובה זו, אלא שכאן השתמשנו בקלאס. כאמור, המילה *klass* והטכנית האלגנטית יותר מבלבולות מתכנתים שמסתכלים על כל העניין של המחלוקת ועל וודו, אבל ברגע שambilנים שזה ציפוי מעלה שהוא שכבר היה קיים בשפה – הכל נראה פשוט יותר.

תרגיל:

כתבו קלאס בשם Rectangle שיש לה שתי תכונות פרטיות: רוחב וגובה. בפונקציה הבנאית יש שני ארגומנטים: רוחב וגובה והם מאוכלסים בקלאס. לקלאס יש שתי מתודות – שניתן להשתמש בהן: getArea המחזירה את שטח הריבוע (רוחב כפול גובה) ו- getPerimeter המחזירה את היקף הריבוע (רוחב כפול גובה כפול 2).

נסו לקרוא לאחת התכונות הפרטיות על מנת לקבל שגיאה.

פתרונות:

```
class Rectangle {
    #length;
    #width;

    constructor(length, width) {
        this.#length = length;
        this.#width = width;
    }

    getArea() {
        return this.#length * this.#width;
    }

    getPerimeter() {
        return 2 * (this.#length + this.#width);
    }
}

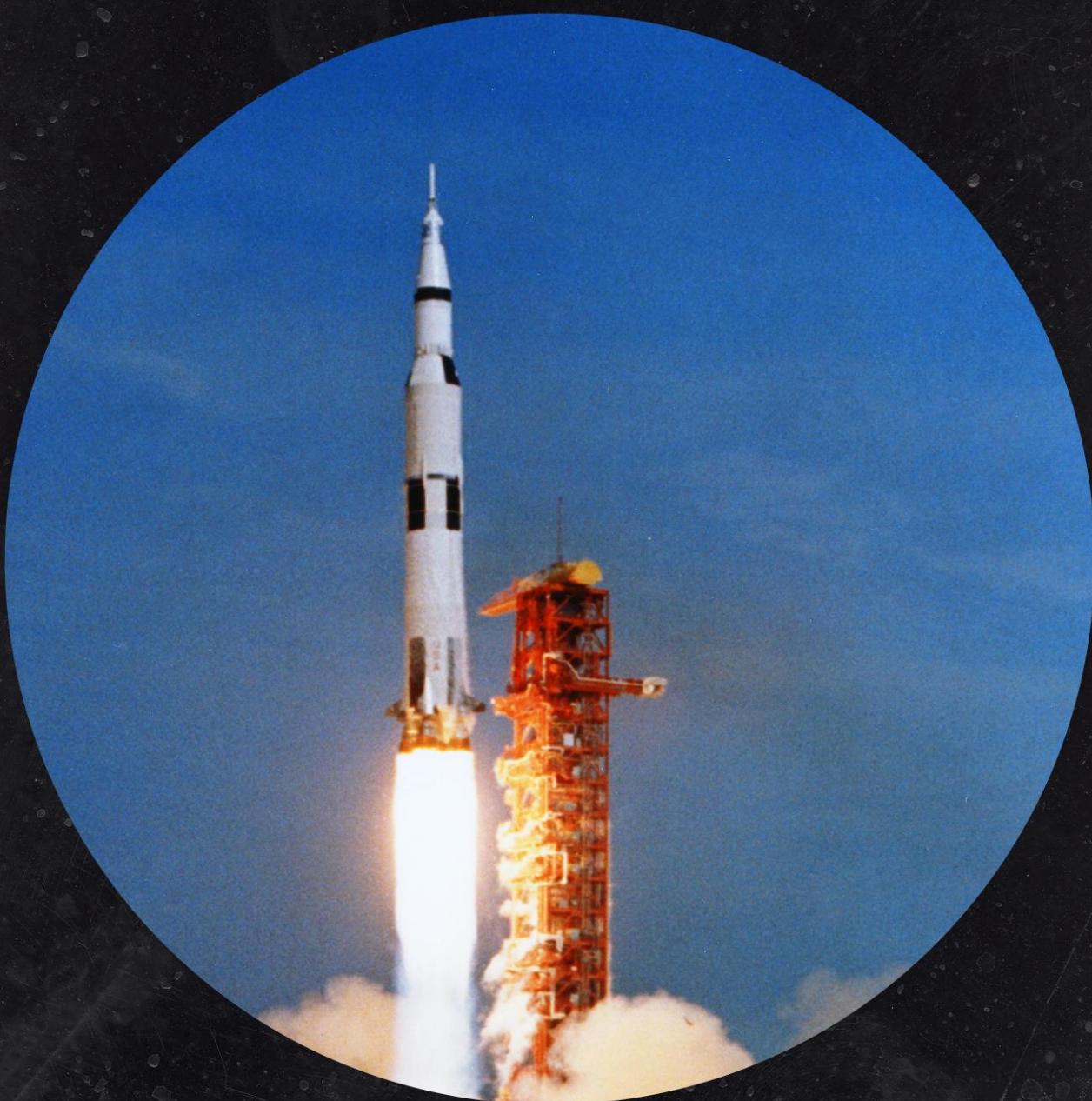
let rectangle = new Rectangle(5, 6);
console.log(rectangle.getArea()); // 30
console.log(rectangle.getPerimeter()); // 22
console.log(rectangle.#length) // Private field '#length' must be declared in an enclosing class
```

הסבר:

עיקר התרגיל הוא יצירת שתי תכונות פרטיות. אורך וגובה. אותן אני מצין כפרטיות באמצעות קריית השם שלהם עם # בהתחלה. מעבר לכך, אני יוצר פונקציה בנאית שמבצעיה אותן ויוצר שתי מתודות פומביות שמבצעות את החישוב שאני צריך.

פרק 19

נמה ענשין?



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 אובייקטיבים ועל לולאות ושינויים מערכיים ואין לאrgan את המידע; העמקנו להאה בקוד באמצעות לימוד משמעותי של אובייקטיבים מבניים ואין ג'אוהסקרייפט מתנהלת בסביבת דף-דף; ובפרקם האחרון למדנו איך עובדים בקוד אסינכרוני. זה לא מעט ללמידה, במיוחד אם לא רואיתם שפת תכנות קודם לכך.

אם עברתם על כל פרקי הספר בסדרם וכן על הנספחים, יש לכם ידע תיאורטי מבוסס על עקרונות ג'אוהסקרייפט והתחביר שלו. אתם אמורים לדעת לנכתב קוד פשוט, לנתח קוד מורכב יותר ולפתח בעיות. על אף שהבסיס שלכם יציב וטוב, עדין נדרש תרגול על מנת לחזק את הידע שלכם בשפה ולהעшир אותו. אף אחד לא הופך למתכנת בעקבות קריית ספר, טוב ככל שהוא.

ניסيون מושי טוב ניתן למצוא בנספח על [Selo](#), שנכתב על ידי מור גלעד, מתכנתה בחברת [Wi.Wi](#). בנספח זהה, אפשר ללמוד איך בונים רכיב באתר [Wi.Wi](#). רכיב שניית להשתמש בו בכל אתר ו אף למכור אותו ללקוחות.

אם אתם רוצים להמשיך ולהרחיב את הידע התיאורטי שלכם, אתם מוזמנים להמשיך בספריו ההדרינה הנוספים שכתבתי. היתי מתחילה בקריאה הספר "למוד Node.js בעברית" שילמד אתכם ג'אוהסקרייפט מצד שרת. ומיד לאחריו בספר "למוד ריאקט בעברית" שילמד אתכם בניית ממשקים בג'אוהסקרייפט ואז הספר "פיתוח ווב מושי בעברית" שמרת איך יוצרים פרויקטים שלמים, מעלים אותם לרשות וממנפים את הידע לניסيون ממשי וכייזד משלבים אותם בקורות החיים שלכם. הספרים האלה יותר מעשיים ויסיעו לכם למנף את הידע החדש שלכם בתחום מעשי ממש. אם אתם לא יכולים לרכוש את הספרים האלה, המדריכים החינמיים שכתבתי ומופיעים באתר שלי: [internet-israel.com](#) יעזרו לכם. כדאי לעבור על הספרים או על המדריכים עד שתדעו את התיאוריה טוב מספיק.

אבל מהתיאוריה צריכים לעשות למשה ולכתיבת קוד שעובד ממשו, וזה הצעד הכני קשה. אין בדיק עוברים ממצב שבו אני יודע (תיאורטי) לנכתב קוד ג'אווהסקריפט למצב שבו אני כותב ממשו שעבוד ושאנשים משתמשים בו?

הדרך הכני טובה היא לבנות ולכתוב. השאלה היא מה. פה שום ספר או מדריך כבר לא יעזרו לכם וזה תלוי בהם ובאריכים שלכם. ג'אווהסקריפט היא השפה הפופולרית ביותר בעולם, ומשתמשים בה בהרבה מאוד מקומות: באתר אינטרנט, באפליקציות סקוטופ, בטלפון, בשרתים ועוד. אחרי שלמדתם את אחד מהנושאים האלה או את כולם, כדאי להתנסות בקוד ממשי.

המקום הטוב ביותר להתחילה הוא באמצעות תרומות קוד לפרויקט קוד פתוח שאפשר למצוא בפייסבוק. עבודה במסגרת פרויקט קוד פתוח זה יכולה להעשיר ולהעמיק מאוד את הידע שלכם. בחנו פרויקט מבוסס ג'אווהסקריפט ונסו לתרום לקוד שלו. באמצעות פתרון באג שמתועד בפרויקט, באמצעות תוסף לדוקומנטציה, לבדיקות אוטומטיות או אפילו באמצעות הוספת תכונה מסוימת. נסו להתקין קומפוננטת ריאקט על אתר של חבר, לבנות תוסף קטן ל-[jQuery](#) עבור האתר של העמותה החביבה עליהם, לתרום לדוקומנטציה של קומפוננטה שאתם אוהבים במיוחד, להגיע להאקטון ולהצטרף לצוות קיים או אפילו רק לשבת ולהסתכל על מתכנתים אחרים. אפשר אף רצוי להתנדב בעמותה ולהתנסות בעבודה בפרויקט ממשי שאנשים ישתמשו בו. יש הרבה דרכים – איך דרך תבחרו? זה תלוי בהם.

יש לא מעט מיטאפים ופגישות של מתכנתים שכדי הגיעו אליהם. במפגשים האלה פוגשים מתכנתים שעובדים בחברות מגוונות, והאוירה נעימה מאוד. אפשר ורצוי לשאול את האנשים האלה איך מתקדמים הלאה בלימוד. יש גם קבוצות פיסבוק בנושא, שהאנשים הנחמדים והמקצועיים בהן יכולים ליעץ לגבי המשך הדרך.

באטר הזה יש רשימה של מקורות ושל קבוצות שאנו משתמשים בהן, אני ממליץ לכם להסתכל בה ולבדק את הקבוצות ואת המקורות השונים:

<https://github.com/barzik/web-dev-il-resources>

כן או אחרת, חשוב מאוד להמשיך לכתוב בשפה. כאמור, שפת תכנות אינה שונה מהוותית לשפה מדובר; אם לא תשימוש בה, תשכח אותה. אפשר להשתמש בג'אווהסקריפט במגוון עצום של פרויקטים – רק צריך לבחור פרויקט ולהמשיך לתרגם. להשתלב בתחום ההיבט זה לא בשםים וזה אפשרי. אם אני הצלחתי – כל אחד יכול.

נספח: למד טייפסקרייפט בעברית

טייפסקרייפט היא השפה ש מבוססת על ג'אוהסקריפט, מרחיבת אותה, מוסיפה לה תכונות ולבסוף מומרת לג'אוהסקריפט. טייפסקרייפט נחשבת מאוד פופולרית ובהרבה חברות מתקדמות כבר עובדים אליה והיא נחשבת כידע בסיסי ונדרש לעובדה בסביבה מודרנית. טייפסקרייפט נצירה ונתמכת על ידי חברת מיקרוסופט והלימוד של טייפסקרייפט, למי שכבר יודע ג'אוהסקריפט, הוא לא מסובך מדי. טייפסקרייפט היא שפת תכנות שהיא ג'אוהסקריפט עם טיפים (Types) – קלומר סוגי מידע. כל הקוד שנכתב בג'אוהסקריפט הוא בעצם טייפסקרייפט אך ניתן לנתח עוד אינדיקציות על סוג המידע השונים.

על מנת להבין למה בכלל צריך את טייפסקרייפט, אני משתמש בדוגמה הזו. הנה וניה שיש לנו פונקציה הממירת מספר לימי השבוע. משהו בסגנון זהה:

```
function weekDay(val) {
  switch (val) {
    case 1:
      return "Sunday";
    case 2:
      return "Monday";
    case 3:
      return "Tuesday";
    case 4:
      return "Wednesday";
    case 5:
      return "Thursday";
    case 6:
      return "Friday";
    default:
      return "Saturday";
  }
}
```

מה קורה אם אני מעביר נתון שאינו מספר? למשל מחרוזת טקסט שהיא "5", או false למשל? אני קיבל תשובה נכונה.

```
console.log(weekDay(true)); // Saturday
console.log(weekDay('5')); // Saturday
console.log(weekDay(false)); // Saturday
```

מדוע זה קורה? כמובן שג'אווהסקריפט היא שפה דינמית ולכן אין בדיקה על הקלט שמתקבל לתוכה הפונקציה וסוג המידע שלו. אני יכול לבדוק באמצעות משפט תנאי אם אכן הנתון שאני מzin לפונקציה הוא בפורמט המתאים. הנה דוגמה:

```
if (isNaN(val) || val > 7 || val < 0) {
    throw "Invalid input";
}
```

אם אני אזין קלט לא מתאים, אני מקבל שגיאה בזמן ריצה. שגיאה כזו, למורת שהיא מקובלת בשלב הלימוד, היא בעייתית כאשר אנו מרצים קוד באתר או אפליקציה אמיתיים. חישבו על זה – אתר שמציג את התאריך אבל פתאום לא יציג אותו. לא נעים ומוגע לתקלות. הרו כל מתכנת יצרך לבחון היטב כל פונקציה שהוא משתמש בה כדי להבין מה אפשר להזין בה.

האם אנו יכולים להשתמש בכלי קסום שיבדק איזה סוג מידע הפונקציה יכולה לקבל? שגם יציג בפני המתכנת שאפשר להכניס לפונקציית `weekDay` מספר אחד עד שבע בלבד וגם יתריע בפניו אם הוא מכניס קלט שגוי למערכת? ויעשה זאת עבור כל פונקציה?

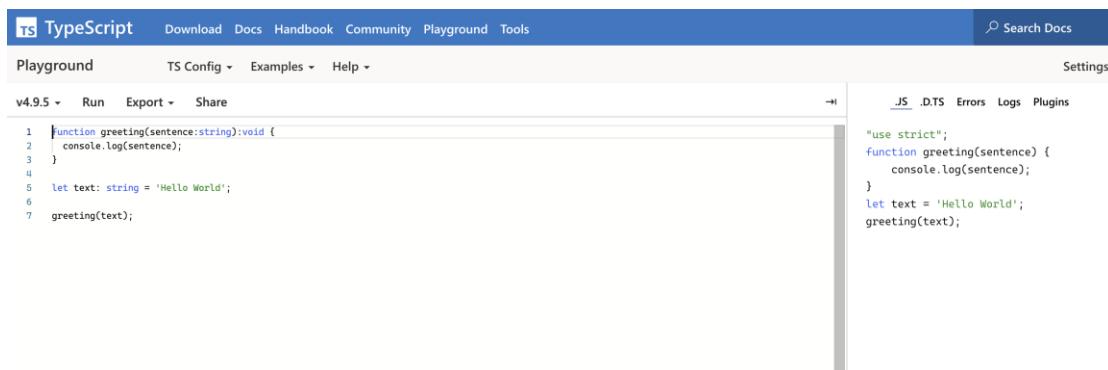
התשובה היא כן. זה אחד היתרונות בשימוש בטיפסקריפט. שהיא בעצם ג'אווהסקריפט אבל עם טיפים. המונח המקבוצי נקרא `superset` – כלומר שפה שיושבת על שפה אחרת, מכילה את כל מה שיש באותה שפה אבל תכונות נוספות. טיפסקריפט היא `superset` של ג'אווהסקריפט.

דפדףים לא יודעים להריץ טיפסקריפט. זו הסיבה שאנו צריכים תוכנה מיוחדת שתמיר את קוד הטיפסקריפט שלנו לקוד ג'אווהסקריפט שדףן, או מחשב, יוכל להריץ אותו. התוכנה המיוחדת זו נקראת **קומפיילר** (Compiler). היא ממירה קבצי טיפסקריפט לג'אווהסקריפט. קומפיילר הוא מונח בסיסי במדעי המחשב שלא נ逋וק בו בפירוש בספר זה אבל אני ממליץ ללכט ולקרא על המושג והשימוש בו כדי להרחב את הידע.

קובצי טיפסקריפט הם עם סימנת `ts`. בניגוד לקובצי ג'אווהסקריפט שלהם מוקדשת סימנת `js`. אם תבחנו פרויקטים קיימים בטיפסקריפט, תראו שהקובצים שלהם מסתיימים בסימנת `ts`. אנו נסביר על כך בהמשך אבל כרגע, על מנת שלא להסתבך עם סביבת

העובדת, אנו נלמד ונשתמש בקומפיילר אונלייני שנמצא באתר הרשמי של שפת טייפסקרייפט בכתובת זו:
<https://www.typescriptlang.org/play>

אם תכנסו לאתר, תוכלו לראות שהוא מורכב משני חלקים:



The screenshot shows the TypeScript playground interface. On the left, the TypeScript code is displayed:

```
v4.9.5
1 function greeting(sentence:string):void {
2   console.log(sentence);
3 }
4
5 let text: string = 'Hello World';
6
7 greeting(text);
```

On the right, the generated JavaScript code is shown:

```
"use strict";
function greeting(sentence) {
  console.log(sentence);
}
let text = 'Hello World';
greeting(text);
```

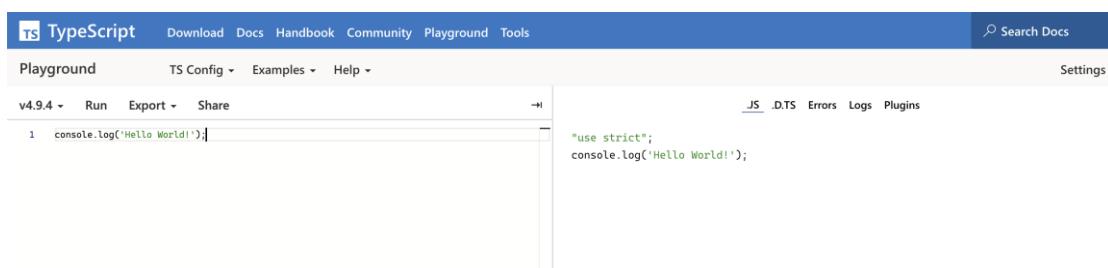
הشمאליו מכיל קוד טייפסקרייפט והימני כולל את הקוד המקורי בג'אווהסקריפט. כאשר אנו מקlidים את טייפסקרייפט בצד שמאל, אנו מקבלים את הקוד בג'אווהסקריפט בצד ימין.

הבה נתאמן. נכנס לאתר ונקליד בצד שמאל את הקוד זהה:

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

אנו נראה שהפלט שלנו הוא זהה:

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



The screenshot shows the TypeScript playground interface. On the left, the TypeScript code is displayed:

```
v4.9.4
1 console.log('Hello World!');
```

On the right, the generated JavaScript code is shown:

```
"use strict";
console.log('Hello World!');
```

זה יכול להראות מוזר, איך `console.log` שאותו אנו מכירים בג'אווהסקריפט, הוא קוד תקין בטייפסקרייפט? התשובה היא פשוטה: טייפסקרייפט היא הרחבה של ג'אווהסקריפט

עם סוגי מידע. זה אומר שכל קוד בג'אווהסקריפט שאוותה אתם מכירם ואוהבים בשלב זה, הוא קוד טייפסקריפט כשר למהדרין. כאמור טייפסקריפט היא ג'אווהסקריפט עם תוספות. ובדוק כיצד שהמברגר הוא המבורגר גם עם תוספות כמו ملفון חמוץ, עגבניה ובצל – כך גם טייפסקריפט.

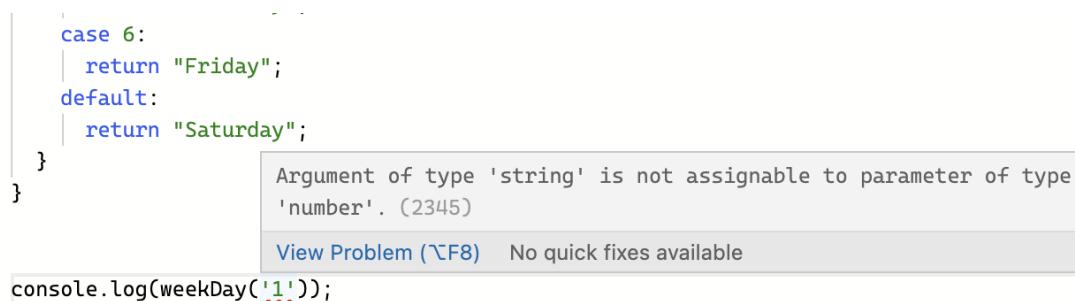
הבה וננסה להכניס תוספת אחת. התוספת זו תציין שהfonקציה `weekDay` מקבלת מספר בלבד. אני אראה ראשית את התוספת ואז איך היא עובדת:

```
function weekDay(val: number) {
  switch (val) {
    case 1:
      return "Sunday";
    case 2:
      return "Monday";
    case 3:
      return "Tuesday";
    case 4:
      return "Wednesday";
    case 5:
      return "Thursday";
    case 6:
      return "Friday";
    default:
      return "Saturday";
  }
}
```

התוספת היא `val: number` שמופיעה בארגומנט. אני מצין שהערך של הקלט המצופה הוא מספר. אם אני אנסה להכניס את הפקודה הבאה:

```
console.log(weekDay('1'));
```

אני מקבל שנייה מיידות:



השגיאה גם תופיע בפלט השגיאות:



Errors in code

Argument of type 'string' is not assignable to parameter of type 'number'.

למה זה חשוב? כי השגיאה זו מופיעה ברגע כתיבת הקוד. כרגע אנחנו עדין בשלב ה"שלום עולם" הפשט וכותבים בעורק הטקסט של טייפסקריפט. אבל חישבו על פרויקט ממש מעוטרי יותר, בן عشرות אלפי או אפילו מיליון שורות. אתם מתכוונים הנדרשים להשתמש בפונקציה כלשהי ויכולים לקבל חוויה מיידי על הקלט שלכם והשימוש שלכם בפונקציה מבלי לבדוק אותה. זה חוסך זמן ותקלות מרבות. זו הסיבה שחברות גדולות מאוד אוהבות טייפסקריפט וטייפסקריפט הפכה לפופולרית מאוד – בזכות היכולת להגדיר קלט ופלט. קלט או output באנגלית הוא הגדרה למידע שאנו מכנים במיוחד תוכנה – למשל פונקציה. פלט, או output באנגלית הוא הגדרה למידע שאנו מקבלים מיחידת תוכנה (כמו פונקציה). עם טייפסקריפט אנו יכולים לוודא או לבדוק את סוג המידע של הקלט ושל הפלט ואנו נלמד על טייפסקריפט באמצעות עורך הטקסט הפשט שיש באתר וرك לאחר מכן להמיר את טייפסקריפט לג'אווהסקריפט גם בסביבה אמיתית – אבל הערך האמתי של טייפסקריפט הוא בפרויקטים גדולים.

קביעת סוג הקלט של פונקציה

אנו קובעים את סוג הקלט שפונקציה צריכה לקבל באמצעות נקודותים וסוג המידע. יש לנו כמה סוגי מידע פרימיטיביים:

| סוג המידע | שם סוג המידע בטייפסקריפט |
|-----------------------|--------------------------|
| מספר | number |
| טקסט | string |
| (False או True) כלומר | boolean |

| | |
|---------|--------|
| any | כל ערך |
| unknown | כל ערך |

נשתמש בדוגמה נוספת לוודא שהבנו. נניח שיש לנו את הפונקציה זו:

```
function BoolChecker(value) {
  if (value) {
    console.log("value is true");
  } else {
    console.log("value is false");
  }
}

BoolChecker(true); // value is true
BoolChecker(false); // value is false
BoolChecker(); // value is false ⚡
BoolChecker("false"); // value is true ⚡
```

אפשר לראות שאנחנו מקבלים תוצאות לא צפויות בכלל שאין לנו בדיקה של הקלט של הפונקציה. אבל עם טייפסקריפט אנו יכולים למנוע הופעת תוצאות לא צפויות אם אנו מפרט מה אנחנו מצפה לקבל. איך? בארגומנט של הפונקציה יש להוסיף נקודותים ואז את סוג המידע. במקרה זהה `boolean`.

```
function BoolChecker(value: boolean) {
  if (value) {
    console.log("value is true");
  } else {
    console.log("value is false");
  }
}

BoolChecker(true); // value is true
BoolChecker(false); // value is false
BoolChecker(); // value is false???
BoolChecker("false"); // value is true???
```

אם תעתייקו את התרגיל אל הקומפיילר החי של טייפסקריפט ותנסה להריץ את הקוד באמצעות Run, תגלו שאתם מקבלים מיד שגיאה. גם כאשר אתם מרכיבים את הקוד עם Run וגם ממש בכתיבה שלו. אני ממליץ לכם לנסות את קטע הקוד שלעיל לآخر וצפו בשגיאה. צירפתי צילום מסך להמחשה:

The screenshot shows the TypeScript playground interface. In the code editor, there is a function named `BoolChecker` which logs 'value is true' if the argument is truthy and 'value is false' if it is falsy. Below the function, three calls are made: `BoolChecker(true)`, `BoolChecker(false)`, and `BoolChecker()`. The last call is underlined with a red squiggle, indicating a type error. In the bottom right corner of the playground, there is a panel titled 'Errors in code' with two error messages: 'Expected 1 arguments, but got 0.' and 'Argument of type 'string' is not assignable to parameter of type 'boolean''. There are also tabs for .D.TS, JS, Errors (with a count of 2), Logs, and Plugins.

```

1 function BoolChecker(value: boolean) {
2   if(value) {
3     console.log('value is true')
4   } else {
5     console.log('value is false')
6   }
7 }
8
9 BoolChecker(true); // value is true
10 BoolChecker(false); // value is false
11 BoolChecker(); // value is false???
12 BoolChecker('false'); // value is true???

```

כפי שכתבתי קודם לכן, זה נראה מוזר ואFILEו מיותר – אבל אל תשחחו שאנו בשלב הלמידה. חישבו על עצמכם בעתיד, כאשר אתם תכתבו קוד למערכת גדולה. האם יש לכם פנאי לבדוק בכל פונקציה ופונקציה מה הקלט המתאים? עם טייפסקרייפט יש לכם חיוי מיידי שחווסף צרות ובעיות.

אם יש לנו מספר ארגומנטים, אנו יכולים לקבוע את הסוג של כל אחד מהם עם נקודותיים וסוג המידע. למשל:

```

function multiply(arg1: number, arg2: number) {
  return arg1 * arg2;
}

```

אם יש לנו ארגומנט אפשרי – קלומר זה שניתן להשתמש בו או לא ניתן להשתמש בו? במקרה הזה נסמן ארגומנט כאפשרי באמצעות סימן שאלה לפני הנקודותיים. הנה דוגמה:

```

function VATCaclulator (sum: number, otherVATNumber?:number) {
  const vat = otherVATNumber || 1.17;
  return sum * vat;
}

```

בדוגמה זו הפונקציה מוסיפה מע"מ לסכום. המע"מ הוא בסכום בירית מחדר של 17% אך ניתן להעביר שיעור מע"מ אחר. כיוון שלא חובה להעביר אותו, הוא מסומן בסימן שאלה. אך כן יש הגדרה של סוג המידע. כך שגם אני אעביר בטעות ארגומנט שהוא לא מספר, אני אקבל שגיאת טייפסקרייפט ואדע לתקן.

"סוג מידע" נוסף שיש הוא עהמ. מדובר בעצם בסוג מידע שכולל את הכל, כל סוגי המידע.

זה כמובן נוגד את המטרת של טייפסקריפט ויש מקומות שאוסרים על שימוש ב-any. אם אני משתמש ב-any, עדיף כבר שלא להשתמש בו כאשר הוא עומד בפני עצמו אך יש לו כמה שימושים לגיטימיים יחד עם מערכיים ואובייקטיבים.

הגדרת סוג מידע של משתנה

כפי שנition להגדיר סוג מידע בקלט של פונקציה, כך ניתן להגדיר סוג מידע במשתנה. כלומר להגדיר משתנה יקבל סוג מידע אחד ספציפי ואי אפשר יהה להכניס לתוכו סוג מידע אחר. זה שימושי במיוחד למשתנים גלובליים או משתנים בפונקציות מאוד מאוד ארכוכות וכמובן לקלאסים. עושים את זה בדיק כמו בקלט של פונקציה. נקודתיים ואז סוג המידע.

הנה דוגמה קלאסית – הגדרתי מחיר גלובלי של 100 (דבר שעושים לעיתים בחניות ורטואליות על מנת לחשב גובה הנהה וכו'). הגדרתי אותו כמספר בטוייפסקריפט. אם אני בטעות אוסיף לו מחוזות טקסט, דבר שעלול ממש לשבש את התפקוד, טייפסקריפט ימנע מהני. בעוד שם הייתה עושה את זה בג'אוּהַסְקְּרִיפְט בלבד, ללא טייפסקריפט, זה היה עובד והיהתי מקבל ערך שגוי. אם אני אנסה להכניס משתנה מחוזות טקסט, גם כן אקבל שגיאה.

```
let globalPrice: number = 100;

// Later ...

globalPrice += '200'; // Type 'string' is not assignable to type 'number'.

globalPrice = '100'; // Type 'string' is not assignable to type 'number'.
```

השגיאות האלו, המוצגות מייד בעורק הקוד שלנו, מסיעים לנו מאוד כאשר אנו כתבים קוד בעולם האמיתי.

הגדרת סוג מידע של פלט של פונקציה

בדיק נמו קלט ומשתנה, אנו יכולים לקבוע סוג מידע של פלט של פונקציה. אנו עושים את זה על מנת לוודא שהפלט של הפונקציה יהיה בסוג מידע שאנו רוצים. זה חשוב כדי לוודא שבאמת הפונקציה המחשבת את המתייר עם מע"מ תחזיר לנו מספר ולא מחרוזת טקסט, שהפונקציה ששולחת בקשה AJAX לשירות מסוים מחזירה לנו מחרוזת טקסט. כאשר אנו מגדירים את סוג המידע שהפונקציה מחזירה לנו, אנו יכולים להמנע מטעויות של לקבל מחרוזת טקסט מפונקציה של חישוב המע"מ ואז לקבל שגיאה של NaN (שניהם שמתקבלת כאשר עושים פעולות מתמטיות על מחרוזת טקסט) כאשר אנו מנסים להוריד מהמחיר של המוצר הנחה מסוימת.

קבעת סוג הפלט נעשית גם היא באמצעות נקודותיים וסוג המידע, אך לאחר הסוגרים של הקלט בהגדרת הפונקציה. הנה דוגמה:

```
function addNumbers(x: number, y: number): number {
    return x + y;
}

let sumOfNumbers: number = addNumbers(10, 20);

console.log(sumOfNumbers); // 30
```

כאן אנו מגדירים שפונקציית addNumbers, מקבלת שני ארגומנטים, x ו-y מחזירה מספר תקין.

במידה והפונקציה לאמחזירה דבר, ניתן להגיד `void` כסוג המידע שהוא מחזירה. זה סוג מידע שאומר שהפונקציה פשוט לא כולל הצהרת `return`.

```
function printMessage(message: string): void {
    console.log(message);
}

printMessage("Hello, world!"); // Hello, world!
```

גם פה אפשר לראות את היתרונות העצום של טייפסקייפ. תחשבו למשל שהפונקציה הזאת היא פונקציה שימושים בה לאורך כל המערכת שלכם. אחרי כמה שינויים מתכנת צעריך בצוות שלכם מקבל משימה של להוסיף את התו "שׁ" לכל המחיר. הוא משנה את הפונקציה באופן הבא:

```
function addNumbers(x: number, y: number): number {
  return x + y + 'x'; // Type 'string' is not assignable to type 'number'.
}

let sumOfNumbers: number = addNumbers(10, 20);

console.log(); // 30
```

הוא יקבל שגיאה נमובן. אם הוא ישנה את חתימת הפונקציה ואת הקלט שלה ל-`string` – הוא יקבל שגיאה ממוקמות אחרים בקוד – כמו משתנים שהותzáה של הפונקציה נכנסת אליויהם ואמורים להיות מספרים. ככה אנחנו מגינים על הקוד שלנו מפני טעויות או באגים עתידיים.

```
function addNumbers(x: number, y: number): string {
  return x + y + 'x';
}

let sumOfNumbers: number = addNumbers(10, 20); // Type 'string' is not assignable to type 'number'.

console.log(); // 30
```

למדנו עד כה על הגדרת סוג מיידע בסיסיים כקלט של פונקציה, פלט של פונקציה ולמשתנה. אני מקווה שהצליחתי להסביר גם למה זה חשוב. כרגע התרגול שלכם הוא באמצעות עורך הטקסט של טייפסකרייפט ולא על פרויקט אמיתי – אבל בשעונים על פרויקט אמיתי מבינים את הצורך והכוח של טייפסකרייפט.-cut נלמד על סוג מיידע מורכבים יותר מאשר מספר או טקסט.

מערכות

דנו בקלט של סוג מיידע פרימיטיביים – אך יש לנו גם סוג המיידע אחרים. למשל מערכים. אם יש לנו פונקציה שמקבלת מערך – איך ניתן להציגו על כן? למשל פונקציה שמבצעת סכימה של ערכי האיברים בתחום מערך. אני רוצה לוודא שאני מעביר לתוכה מערך של מספרים. אם במערך יהיה אפלו איבר אחד שאינו מספר, הסכימה של המספרים עלולה להיכשל. במקרה הזה אני מצין שהקלט המבוקש הוא מערך של מספרים באמצעות `[] number`:

הנה הדוגמה שמבヒירה לנו היטב כיצד זה נראה בפועל:

```
function sumArray(numbers: number[]) {
  let sum = 0;
  for (const num of numbers) {
    sum += num;
  }
  return sum;
}

const numbers = [1, 2, 3, 4, 5];
const sum = sumArray(numbers);
console.log(sum); // prints 15
```

אפשר כמובן מערך של מחרוזות טקסט בלבד:

```
function toUppercase(strings: string[]) {
  const result = [];
  for (const str of strings) {
    result.push(str.toUpperCase());
  }
  return result;
}

const strings = ['hello', 'world', '!'];
const result = toUppercase(strings);
console.log(result); // prints ['HELLO', 'WORLD', '!']
```

אני יכול לבחור מערך בלבד באמצעות `any`, אבל כמובן שהוא מקרה שבו אנחנו רוצים להשתמש במרקם הטרוגני שכולל כמה סוגים שונים. אנו יכולים לציין מערך עם מספר איברים ספציפיים. למשל 4 איברים שהם מספר. עם פירוט של סוג המידע לכל איבר במרקם. כאן למשל אני מגדיר במפורש מערך של ארבע מספרים:

```
function average(numbers: [number, number, number, number]): number {
  let sum = 0;
  for (const num of numbers) {
    sum += num;
  }
  return sum / numbers.length;
}
```

ישנה דרך אחרת להגדיר מערכים בקלות שנדון עליה בתת החלק הנקרא "גנריות" בהמשך.

אובייקט עם סוג מידע פרימיטיביים

אנו יכולים להגיד אובייקט מסווג מידע (קלט, פלט או עברו משתנה) בדיק נמו מערך. כל מה שאנו צריכים לעשות זה להגיד את חתימת האובייקט ולהכניס אותו אחרי הנקודות במקום שבו היינו מגדירים את סוג המידע הפרימיטיבי.

כך למשל, אם יש לי פונקציה המתקבלת כקלט אובייקט של משתמש שיש בו name שהוא מחוץ טקסט ו-age שהוא מספר, ההגדרה שלו תהיה `{name: string, age: number}` וההגדרה הזאת תבוא מיד אחר הנקודות בדיק נמו הגדרה של מערך או של סוג מידע פשוט יותר.

הנה דוגמה:

```
function greetPerson(person: {name: string, age: number}): string {
  return `Hello, my name is ${person.name} and I am ${person.age} years old.`;
}

let john: {name: string, age: number} = {
  name: "John",
  age: 30,
};

console.log(greetPerson(john)); // Hello, my name is John and I am 30 years old.
```

אפשר לראות שההסתברן. אם למשל יש לי אובייקט רחב יותר שאחט מתוכנותיו היא פונקציה או מערך או חיללה וחס מערך של אובייקטים. במיוחד כשצריך אותם יותר מפעם אחת. במקרה הזה יש לטיפוסקריפט דרך להגיד באופן יותר פשוט אובייקטים ואנו נסביר על זה יותר מאוחר בחלק על אינטראפיס בטיפוסקריפט.

סוג מידע: פונקציה

לעתים אנו נדרשים להגיד סוג מידע שהוא פונקציה. למדנו בספר שיש פונקציות שהקלט שלהם הוא פונקציה בעצמו. אנו רואים את זה בקובוקים למשל. אבל לא חסרות דוגמאות שבהן הקלט, או הפלט של הפונקציה או השמה של המשתנה היא פונקציה. ניתן להגיד סוג מידע כפונקציה ספציפית עם "חתימה". מה זאת אומרת חתימה? זה מונח

מקצועי (באנגלית: Signature) שבאמצעותו מגדירים איך פונקציה נראה. מבחינת הקלט ו מבחינת הפלט. לא מה שקרה בתוכה כמובן, כי זה פחות רלוונטי אליו. החתימה היא הקלט והפלט בלבד. וזה גם מה שהוא רוצה שהקומפילר של טייפסקייפ יבודק. אם לפונקציה האсинכורונית שלי אני מעביר קולבק שאמור לקבל מספר ולהחזיר מספר, זה מה שחשוב לי – כי אני רוצה לוודא שכל מי שימוש בקולבק הזה יעביר ויקבל מספר.

איך החתימה נראה? בגודל סגריים, שמכילים את הארגומנטים ואת מה שחוור. הנה ונבחן חתימה של פונקציה.

(x: number) => void
במקרה הזה החתימה אומרת שהפונקציה מקבלת ארגומנט שהוא מספר ומחזירה שום דבר. זה סוג המידע עצמו – פונקציה שמצויה לחתימה זו.

הנה דוגמה לשימוש בסוג חתימה כזה:

```
function doSomething(callback: (x: number) => void): void {
  console.log("Doing something...");
  callback(10);
}

doSomething((x: number) => {
  console.log(`Received callback with value: ${x}`);
});
```

פונקציית doSomething אמורה לקבל קלט בשם הוא callback. מה סוג הקלט זהה? פונקציה. איזו פונקציה? נזו מקבלת מספר ומחזירה שום דבר. וזה בדיקת הפונקציה שאני מעביר:

```
(x: number) => {
  console.log(`Received callback with value: ${x}`);
}
```

הבה ונדגים עם חתימה אחרת.

```
function performAction(callback: (message: string) => void): void {
  console.log("Doing something...");
  callback("Hello, world!");
}

performAction((message: string) => {
  console.log(`Received callback with message: ${message}`);
});
```

הfonקציה `performAction` מקבלת ארגומנט שהוא קולבק. קלומר הקלט שלה הוא קולבק. איזה סוג של קולבק מוגדר בחתימה:

```
(message: string) => void
```

קלומר הקולבק לא מחזיר דבר ומקבל מחרוזת טקסט כארוגמנט. זה אומר שאם אני משתמש ב-`onAction`, אני חייב להעביר לה ארגומנט פונקציה שהקלט שלה הוא מחרוזת טקסט ולא מחזירה דבר. וזה בדיק מה שאני עושה בקוד.

אני יודע שהיה עלול להיות מבלבל, אז אמחיש עם חתימת פונקציה נוספת. הפעם, הפונקציה האסינכרונית אמורה לקבל פונקציה אחרת, אותו קולבק. אנחנו רוצים שהkolbek יהיה פונקציה כMOVן שמקבלת מחרוזת טקסט כארוגמנט ומחזירה טקסט. זו החתימה שלה:

```
(message: string) => string
```

אני עבור את החתימה כסוג המידע של הארגומנט של `getResult`. זה מה שאני רוצה שיעבור כסוג המידע. למשל, פונקציית `getResult` אמורה לקבל רק ארגומנט בשם זה `callback`. והוא חייב להיות פונקציה מסווג מאוד ספציפי – צו שהארוגמנט שלה הוא טקסט והיא מחזירה טקסט. ממשו צזה:

```
(message: string) => {
  console.log(`Received callback with message: ${message}`);
  return `Message received: ${message}`;
}
```

הfonקציה צזו מציית לחולוטן לחתימה. היא מקבל מחרוזת טקסט ומחזירה מחרוזת טקסט. ייתכן שהיא עשויה עוד דברים בפנים, אבל זה לא מעניין אותנו. סוג המידע בעצם של פונקציה צזו נראה ככה:

```
(message: string) => string
```

אם הפונקציה שלי יכולה לקבל רק פונקציה עם סוג המידע זהה, אני רק צריך לומר לה את זה – קלומר במקומם להעביר סוג מידע של מחוזות טקסט, מספר או אובייקט, אני פשוט מעביר סוג מידע של חתימה. בדוק באותו אופן. אחרי הנΚודתיים להעביר את החתימה.

כך זה יראה:

```
function getResult(callback: (message: string) => string): string {
  console.log("Doing something...");
  return callback("Hello, world!");
}

let result: string = getResult((message: string) => {
  console.log(`Received callback with message: ${message}`);
  return `Message received: ${message}`;
});

console.log(result); // Message received: Hello, world!
```

כמה סוגי מידע אפשריים

לעתים אנו יכולים לקבל כמה סוגי מידע. כמו למשל פונקציה שיכולה לקבל מחוזות טקסט וגם מערך. אנו יכולים לציין כמה סוגי מידע באמצעות התו " | " (שמו באנגלית: Union). כך למשל אם כתוב `string | number` בסוג המידע, זאת אומרת שאני יכול לקבל או את זה, או את זה. הנה דוגמה של פונקציה שיכולה לקבל מחוזות טקסט או מערך. היא יודעת לטפל בכל אחד מסוגי המידע יפה מאוד אבל לא יודעת לעבוד עם מספרים או עם אובייקטים.

```
function getFullName(name: string | string[]): string {
  if (typeof name === "string") {
    return name;
  } else {
    return name.join(" ");
  }
}

console.log(getFullName("Moshe Cohen")); // Moshe Cohen
console.log(getFullName(["Moshe", "Cohen"])); // Moshe Cohen
```

סוג המידע הוא `string | string` או מחרוזת טקסט או מערך שמורכב מטקסטים.

אם אנו משתמשים ב-חסון, אנחנו חייבים לוודא שבפונקציה שלנו יש טיפול בכל סוג מידע נפרד והוא יכולה להתמודד עם סוג המידע השני לה. אחרת נקבל שגיאה.

```
function weekDay(value: number | string) {
  let inputValue;
  if (typeof inputValue == "string") {
    inputValue = parseInt(value);
  } else {
    inputValue = Math.floor(value);
  }
  switch (inputValue) {
    case 1:
      return "Sunday";
    case 2:
      return "Monday";
    case 3:
      return "Tuesday";
    case 4:
      return "Wednesday";
    case 5:
      return "Thursday";
    case 6:
      return "Friday";
    default:
      return "Saturday";
  }
}
```

החסון יכול לשמש אותנו גם אם אנו רוצים לעשות רשימת ערכים שאפשר להשתמש בהם. הנה לדוגמה גודל חולצה שיכולה להיות `small`, `medium` או `large`. אך הכנסה של ערך אחר תגרור שגיאה:

```
let shirtSize: 'small' | 'medium' | 'large';

shirtSize = 'small';
shirtSize = 'medium';
shirtSize = 'large';
shirtSize = 'big'; // Type "bug" is not assignable to type "small" | "medium" | "large".
```

בחורתי במשתנה שמקבל סוג מידע אבל כמובן אפשר להשתמש בסוג המידע הזה בפונקציה – בין אם בקלט או בפלט.

Type\ interface

לא מעט פעמים סוג המידע שהוא מגדירים, בין אם מדובר בסוג מידע של אובייקט ובין אם מדובר בסוג מידע של חתימת פונקציה (או אפילו שנייהם) – הם ארוכים וגם חוזרים על עצם בכל מיני מקומות. אני יכול להגיד את סוג המידע `C-type`, `C-interface` ועוד להשתמש בהם כמה פעמים.

אני מגדיר את סוג המידע שאני רוצה באמצעות המילה `shemora type`, את שם סוג המידע ואז את מה שיש בו. הנה דוגמה לمثال של סוג מידע מסווג `User`. המילה `type`, השם של סוג המידע שהחלטתי ששמו הוא `User` ואז סוגרים מסולסלים כמו אובייקט ואת סוג המידע שיש בו.

```
type User = {
  id: number;
  name: string;
  email: string;
  phone: string;
  address: string;
};
```

זה מאפשר לי להשתמש ב-`User` ממש כמו סוג מידע פרימיטיבי בכמה וכמה מקומות שארצה. ממש כמו `string` או `number`. למשל:

```
function showUser(user:User) {
  // Do something with user
}
```

השתמשתי בסוג המידע `User`. זה נכון בדיק אל:

```
function showUser(user: {
  id: number;
  name: string;
  email: string;
  phone: string;
  address: string;
}) {
  // Do something with user
}
```

אבל כאמור שימוש ב-Type יותר נוח. במקרה אם יש לי (ובדרך כלל יש) סוג מידע שחוזרים על עצמם.

נדגים עם דוגמה נוספת. למשל טיפול בזמן. יש לנו 12 חודשים בשנה, לכל אחד מהם יש שם. אנו יכולים להשתמש ב-`חסוענ` כדי להגדיר סוג מידע של חודשים וגם סוג מידע של הקלט של החודשים (1 עד 12).

```
type Month = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12;

type MonthName =
| "January"
| "February"
| "March"
| "April"
| "May"
| "June"
| "July"
| "August"
| "September"
| "October"
| "November"
| "December";
```

הגדרתי סוג מידע שנקרא `Month` שיכול להיות מספר מ-1 עד 12 ומספר זה בלבד. הגדרתי סוג מידע שנקרא `MonthName` שיכול להיות אחד משמות החודשים. אני יכול להשתמש בסוגי המידע בכל מקום ומקום שבו יש לי חודש מסווני ובכל מקום ומקום שבו יש חודש עם שמות.

```

function getMonthName(month: Month): MonthName {
  switch (month) {
    case 1:
      return "January";
    case 2:
      return "February";
    case 3:
      return "March";
    case 4:
      return "April";
    case 5:
      return "May";
    case 6:
      return "June";
    case 7:
      return "July";
    case 8:
      return "August";
    case 9:
      return "September";
    case 10:
      return "October";
    case 11:
      return "November";
    case 12:
      return "December";
    default:
      throw new Error(`Invalid month: ${month}`);
  }
}

```

כאן אני מגדיר שה משתנה שהפונקציה יכולה לקבל הוא מספר מ-1 עד 12 בלבד. ומה שהוא יכול להחזיר זה רק שם חודש. אני יכול למחזר נमובן את סוג המידע הזה בכל מקום.

במקום `type` אני יכול להשתמש ב-`interface`. שניהם דומים מאוד, למעט העובדה שאנו יכולים לשנות את `interface` לאחר שקבעתי אותו ואת `type` לא. בתחילת הדרך של טייפסקורפט הווה רק `interface`. את `type` הוסיפו לשפה מאוחר יותר.

קאסטיינג

לעתים אני צריך להסביר או להגיד לטייפסקורפט סוג מידע מסוים כאשר טייפסקורפט אינה יכולה לדעת בבירור איזה סוג מידע יש לה. זה קורה בעיקר כאשר אנו משתמשים בספריות צד שלישי או בפקודות שאין חלק מג'אוوهסקריפט עצמה (כמו למשל פונקציות הקוראות לאלמנטים ב-DOM).

על מנת להגדיר או להמיר סוג מידע, אנו משתמשים בקאסטיניג (באנגלית: Casting) באמצעות המילה השמורה `as`.

אנו נדנים באמצעות הקוד הבא, שבו הוצב משתנה בשם `myVar` שהגדרתי אותו בסוג מידע שיכول להיות מספר או מחרוזת טקסט. אולי בגלל שבתחילת הקוד אני לא יודע לבדוק מה הוא ומה יכנס לתוכו. בהמשך הקוד, אני מנסה להכניס אותו למשנה אחרת שהוא חייב להיות מספר. על מנת לוודא את זה, אני משתמש ב-`as`. אם המשתנה הוא מחרוזת טקסט (כמו במקרה הזה), אני מקבל הודעה שגיאיה.

```
let myVar: string | number = "100";
let parsedValue = myVar as number;
const result = parsedValue * 2;
console.log(result);
```

דוגמה נוספת ויתר מציאותית היא שימוש בפונקציית DOM על מנת לקרוא לאלמנט HTML מסווג `querySelector`. טיפוסקריפט לא יכולה לדעת שמדובר באלמנט ש תמיד יש לו ערך. אני חייב לומר לה שמדובר באלמנט מסווג `HTMLInputElement`. ואת זה אני עושים עם `.as`.

```
let input =
document.querySelector("input[type='text']") as HTMLInputElement;
console.log(input.value);
```

בכל פעם שיש `as`, אני מגדיר עבור טיפוסקריפט את סוג המידע, גם אם טיפוסקריפט לא בטוחה בנוגע לסוג המידע המתkeletal, או שיתכן שהוא סוג מידע כלשהו.

גנריות

חלק שימושי מאד בטיפוסקריפט הוא גנריות (Generics). מדובר במונח המתאר פלט של פונקציה התלו依 בקלט שלה והוא מצב מאד אופייני בפונקציות וקוד של העולם האמתי. מעבר לכך, אני יכול לבנות מבנים דרך גנריות וזה מכפיל כוח שימושי. על מנת להבין את הכאןי אדגים באמצעות פונקציה המחזירה את האיבר הראשון במערך שהוא מקבלת. מה סוג המידע של הפלט שלה? התשובה היא – תלוי במרק. אם זה מערך

שהאיברים בו הם מספרים, סוג המידע של הפלט הוא מספר. אם זה מערך שהאיברים בו הם מחרוזות טקסט, סוג המידע של הפלט הוא טקסט.

יש מקרים שבה יותר מורכב – למשל, פונקציה שמקבלת קלט מספרי בלבד. אבל אם המספר גדול ממהה, היא מחזירה מחרוזת טקסט – למשל "המון נספ", אם המספר קטן מאוד, היא מחזירה מחרוזת טקסט אחרת – למשל "אין נספ". איך אני אוכל להגיד את סוג הפלט שלו? התשובה היא: גנריות. סוג המידע של הפלט הוא סוג המידע של הקלט.

הבה ונדגים עם הפונקציה הכى פשוטה שישי – פונקציה שמחזירה את מה ששולחים אליה. אם אני אשלח אליה טקסט, אני מצפה לקבל טקסט בחזרה. כאמור סוג הקלט יכול להיות טקסט וסוג הפלט במקרה של טקסט הוא טקסט. אותו הדבר עם מספר. כך נראה:

הפונקציה בג'אווהסקריפט נקי:

```
function returnFunction(value) {
    return value;
}
```

מה סוג המידע שאפשר להעביר לפונקציה כזו? ככלומר הקלט שלו? זה יכול להיות Any. אבל מה סוג המידע שאנו יכולים להחזיר? גם any? כמובן כן ובאמת הקוד הזה ולידי:
לגמר:

```
function returnFunction(value: any): any {
    return value;
}
```

אבל בעוד זהorcheshet נחשב טייפסקריפט, אם השתמש ב-any אנו נאבד את המהות של טייפסקריפט. אני אדגים באמצעות הקוד הזה:

```
function returnFunction(value: any): any {
    return value;
}

let a = returnFunction('hello');
a *= 1;
console.log(a); // Nan
```

אם תרצו אותו, אנחנו מקבל שגיאת Not a number: NaN. זה לא מה שאנו רוצים שיקירה והמטרה של טייפסקריפט היא למנוע בדיק דברים כאלה.

از מה שאני צריך לומר לティיפסקרייפט זה: "אם אני מעביר קלט מסווג מחרוזת טקסט לפונקציה, אז הפלט שלה הוא מחרוזת טקסט". איך אני עושה את זה? באמצעות הסינטקס של גנריות שנכלל סוגרים משולשים <>. ראשית, אני צריך לבחור שם לסוג המידע שהוא אותו אני הולך לקבל מהמשתמש. מקובל לסמן את זה כ-T. אחרי שבחרתי את השם, אני אכתוב אותו בתוך סוגרים משולשים לפני הגדרת הארגומנטים של הפונקציה. מהנקודה הזאת אני יכול להשתמש בו כסוג מידע.

זה עלול להיות קשה להבין, אז הנה הדוגמה שתסביר:

```
function returnFunction<T>(value: T): T {
    return value;
}
```

ראשית אני מגדיר את <T> לפניו רשימת הארגומנטים. השם T הוא לבחירותי ואני יכול לבחור באיזה שם שבא לי. מהרגע הזה, אני יכול להשתמש ב-T כסוג מידע בקלט. במקרה הזה אני מגדיר שמה שמכניסים לי כ-value והוא סוג המידע שנכנס ל-T. אני אפגש את T שוב בהגדרת הפלט. פה אני יודע שמה שאני מכניס זה מה שאני מקבל.

ועכשיו אני מקבל שגיאה אם אני אנסה להתייחס אל מחרוזת הטקסט כאלו מספר, כי אני לא משתמש ב-any והסבירתי לティיפסקרייפט איזה ערך אני מצפה לקבל. אני מקבל

```
function returnFunction<T>(value: T): T {
    return value;
}

let a = returnFunction('hello');
a *= 1; // The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.
console.log(a);
```

נדגים שוב עם פונקציה המחזיר את האיבר הראשון במערך עם איבר יחיד:

```
function returnFunction<T>(value: [T]): T {
    return value[0];
}
```

זה בהחלט עלול להביחיל, אבל אם נבין שזה בדיק, אבל בדיקות כמו:

```
function returnFunction(value: [string] ): string {
  return value[0];
}

:1
function returnFunction(value: [number] ): number {
  return value[0];
}
:1
function returnFunction(value: [boolean] ): boolean {
  return value[0];
}
```

ובעצם כל סוג מידע שהוא – הכל ביחיד והכל תלוי במה שניי מכניס. אני מכניס מידע מסווג טקסט? הפונקציה תחזיר פלט של טקסט. אני מכניס מידע מסווג מספר? הפונקציה תחזיר פלט של מספר. זה מה שניי עושה עם גנריות. פשוט צריך להבין את הסינטקס. ראשית אני מגדיר את סוג המידע הגנריים וקורא להם בשם. במקרה זהה T. כל סוג המידע הגנריים מופיעים בין שם הפונקציה לסוגרים המכילים את המספר.

```
function returnFunction<T>(value: [T] ): T {
  return value[0];
}
```

אחרי כן, אני משתמש ב [] כדי לסמן מערך. בדוק כמו [string] שכפי שŁמדנו הוא מערך שיש בו איבר אחד של מחזות טקסט טקסט ו [number] שהוא מערך שיש בו איבר אחד של מספר. כך [T] הוא מערך שבו איבר עם סוג המידע T. מה סוג המידע זהה יהיה? תלוי במשתמש! תלוי במה שניי מכניס! במקרה הזה טקסט.

```
function returnFunction<T>(value: [T] ): T {
  return value[0];
}
```

השלב האחרון הוא מה הפונקציה ממחזירה וזה עצם הגנריות. מה היא ממחזירה? מה שהכניסתי – T! וכך בעצם אני מגדיר שהפונקציה ממחזירה את סוג המידע שיש במרקן.

הבה נדגים עם פונקציה נוספת. הפעם צזו הפונקציה את המערך:

```
function reverse<T>(items: T[]): T[] {
```

```
return items.reverse();
}
```

גם פה הגרסאות עלולה לבלבול, אבל ננסה להבין זאת. ראשית, אני מגדיר את סוג המידע הגרוריים שיש לי. במקרה זהו סוג מידע גנרי אחד שלו אני קורא T. הארגומנט של הפונקציה הוא items והוא מסוג מערך שיכול להיות או ריק, או מסוג המידע T. מה הפלט של הפונקציה? או מערך ריק או מערך מסוג המידע T, זה שאנחנונו.

כלומר ברגע ש-T מתאפס, אנחנו יכולים להשתמש בו כסוג פלט.

המשתמש יכול להציג על T ולא לנחש אותו מהפלט. זה שימושי כאשר אנחנו לא יודעים תמיד איזה פלט אנחנו יכולים לקבל לפי הקלט ובקלטים לא אחידים. למשל, נניח ויש לי פונקציה המוסיפה 1 לאיבר האחרון של מערך שאינו לא יודע מה יש בו. הוא יכול להיות מעורב עם מחרוזות טקסט ועם מספרים. מה הוא מחזיר? תלוי באיבר האחרון. אבל כן אפשר להוסיף 1 גם למחרוזות טקסט וגם למספר. אם אני אנסה לנתחו אותו באופן הזה:

```
function addOne<T>(items: any[]): T {
  return items[items.length - 1] + 1;
}

let a = addOne([1, 2, 3, 'moshe', 4]);

a = a * 1;
```

אני מקבל שגיאה מסוג `ts` because the left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type. על השורה האחרונה. מדוע? כי לא הגדרתי את T והוא יכול להיות כל דבר. אני חייב להגיד לו את הסוג. איך? עם סוגרים משולשים כך:

```
function addOne<T>(items: any[]): T {
  return items[items.length - 1] + 1;
}

let a = addOne<number>([1, 2, 3, 4]);

a = a * 1;

console.log(a);
```

אני אומר בעצם לティיפסקרייפט שפונקציה `addOne` מחזירה מספר. אז השגיאה נעלמת.
אני יכול לומר לティיפסקרייפט שהפונקציה אמורה להחזיר מחרוזת טקסט באופן זהה:

```
function addOne<T>(items: any[]): T {
    return items[items.length - 1] + 1;
}

let a = addOne<string>([1, 2, 3, 4]);

a = a * 1; // Type 'number' is not assignable to type 'string'.
```

ואז אני מקבל שגיאה כאשר אני אנסה לנכפול עם * - כיוון שלא ניתן להפעיל את זה על טקסט.

גנריות נותנת לי כוח ממשמעותי מאד בהגדלה ובשימוש בפונקציות שונות. אולי בהתחלה יהיה לכם קשה למשגנריות בפונקציות מסוימות, אבל בקדוד קיים לא מעט פעמים תדרשו להקליד את הסוגים המשולשים ולהגדיר איזה סוג מידע אתם צריכים, אז זה קריטי להבין בדיקות מה זו גנריות.

אנו יכולים להשתמש כMOVBN BIOTER מארגומנט אחד. למשל בפונקציה שבה אני מקבל שני משתנים ומחזיר מערך המורכב משנייהם, אני יכול להגיד שהערך יהיה מערך של שני איברים עם סוג המידע של המשתנים.

```
function makePair<T, U>(first: T, second: U): [T, U] {
    return [first, second];
}

const pair = makePair(1, true)

console.log(pair);
```

כאן בחרתי להגיד שני סוגי מידע, T ו-U. הראשון קיבל אוטומטית את סוג המידע שלו מהארוגומנט הראשון, `first`. השני קיבל אוטומטית את סוג המידע שלו מהארוגומנט השני, U. איזה סוג מידע הפונקציה תחזיר? מערך עם שני איברים. סוג המידע של הראשון הוא T וסוג המידע של השני הוא U.

הנה דוגמה ממש מהחיים האמיתיים. פונקציה המחזירה אלמנט HTML והוא גנרית. מדוע?
כי אנחנו בוחרים מה הפונקציה מחזירה.

```
function getElementById<T>(id: string): T {
  let element = document.getElementById(id);
  if (element === null) {
    throw new Error(`Element with id '${id}' not found.`);
  }
  return element as T;
}
```

ברצוננו, אנו נקבע שהפונקציה יכולה להחזיר סוג מידע של אלמנט HTML מתאים או סוג מידע של אלמנט HTML מסווג וידאו שלו יש תכונות מיוחדות. אם הגדרנו שהוא מחזיר אלמנט וידאו

```
let element = getElementById<HTMLElement>('someHTMLElementId');
let myVideo = getElementById<HTMLVideoElement>('someDivElementId');
```

אם ננסה להשתמש במתודה מיוחדת לוידאו על `element`, שאמור להיות מסווג אלמנט HTML רגיל, אנו נקבל שגיאה – קלומר טייפסקרייפט יגן علينا. קיבל את השגיאה ברמת הטייפסקרייפט ולא בזמן ריצה.

```
myVideo.cancelVideoFrameCallback(10);
element.cancelVideoFrameCallback(10); // Property 'cancelVideoFrameCallback' does not exist on type 'HTMLElement'.
```

כאמור, זה נראה מעט מרתקע ויש לתרגל את זה. אבל כשתגינו לחיים האמיתיים ולכתיבת קוד בסביבת קוד אמיתי, נראה שתctraco להגדיר מה סוג המידע שהוא שואתו אתם רוצים לקבל בכלל פונקציה שתקרוו לה. רואים את זה היט בעבודה עם ספרית ריאקט למשל.

גנריות אפשרות לנו להגדיר גם מערך. אם אנו למשל רוצים להגדיר מערך של מספרים, אנו יכולים להשתמש בסינטקס: `[]`. למשל:

```
let a: string[];
a[0] = 'dddd';
a[1] = 1; // Type 'number' is not assignable to type 'string'.
```

אך אני יכול להשתמש גם בסינטקס של גנריות עם המילה השמורה `Array` ולהכניס את סוג המידע שאנו רוצים. זה דרך שונה להביע את אותו הדבר.

```
let a: Array<string>;
a[0] = 'dddd';
a[1] = 1; // Type 'number' is not assignable to type 'string'.
```

טייפסקייפ עם קלאסים

מן הסתם אפשר להגיד סוג מידע גם בתחום קלאסים. כלומר בתור תכונות או מethodים אין שום בעיה להגיד סוג מידע בדיק כmo בכל פונקציה או משתנה שיש מחוץ לקלאס. הנה למשל דוגמה של קלאס שיש בו תכונות ומетодה אחת שבה יש הגדרת טייפסקייפ:

```
class User {
  userId: number;
  userName: string;

  constructor(userId: number, userName: string) {
    this.userName = userName;
    this.userId = userId;
  }

  getUserStatus(): string {
    // Some Call or Business logic
    return 'admin';
  }
}
```

METHODS, PROPERTIES, AND FIELDS

אנו יכולים להגיד בטייפסקייפ מethodים מוגנות ופרטיות. מה זאת אומרת? מethodות פרטיות הן מethodות שבן אנו לא יכולים להשתמש באופן ישיר אלא ורק מwithin הקלאס עצמו. אם אני אנסה להשתמש בהן מחוץ לקלאס אני מקבל שגיאת טייפסקייפ. מmethod הפרטית מוגדרת באמצעות המילה `private`. מרגע שהגדרתי מmethod של קלאס כ-`private`, אני יכול להשתמש בה רק בתחום הקלאס. אני אדגים:

```

class BasicUser {
  userId: number;
  userName: string;
  constructor(userId: number, userName: string) {
    this.userName = userName;
    this.userId = userId;
  }

  private getUserStatus(): string {
    // Some Call or Business logic
    return "admin";
  }
}

let userA = new BasicUser(1, "Ran");
let userAStatus = userA.getUserStatus(); // Property 'getUserStatus' is private and only accessible within
// class 'BasicUser'.

```

כפי שננו רואים בדוגמה, ניסיון להשתמש במתודה פרטית יניב לנו שגיאה.

ניתן להשתמש גם ב-`protected` על מנת לגרום רק לklassים שיורשים מklass אחר להשתמש במתודה זו. זאת אומרת שרק אובייקט שביצע `extend` לאובייקט אחר יוכל להשתמש בפונקציה אבל היא לא תהיה זמינה מחוץ לאובייקט.

השימוש במתודות פרטיות ומוגנות נקרא אנקסולציה ובאנגלית `encapsulation`. למה אנו זקוקים למתודות פרטיות? כי בקוד מבוסס אובייקטים (`Object oriented`) יש לעיתים דרישת צו, על מנת למנוע שימוש לרעה במתודות שלא רוצחים לחשוף החוצה. שימוש באנקקסולציה זו אחת הדריכים ליזור קוד יותר אינטיטי ושהה מאפשר לבצע הפרדה בין המימוש הפנימי של klass למה שקיים מחוץ לה.

כדי לשים לב שמדובר בטיפוסקריפט, קלומר אם אני אנסה להשתמש במתודה פרטית, השגיאות והازהרות יופיעו כאשר אני מקمل את הקוד מטיפוסקריפט לג'אוּהַסְקְּרִיפְט.

קריאה בלבד

אנו יכולים בטיפוסקריפט להגדיר תכונה בklass כקריאה בלבד. זה שימושי כאשר אנו רוצים שתכונה מסוימת לא תשתנה אחרי הגדרת klass. אנו משתמשים במילה השמורה

הטייפסKRIFTI הרגיל, אז שווה להתעכ卜 על זה מעט.

```
readonly prop1 : string;
```

הסינטקס לעיל אומר שהתכונה `prop` (שיכולה להיות מחורצת טקסט בלבד) ניתנת לקריאה בלבד. ככלומר ניסיון לשנות אותה אחרי שייצרנו אובייקט מתוך הקלאס באמצעות `new` יונשל. המקום היחיד שבו ניתן לעורך את הערך של משתנה קריאה בלבד הוא בתוך פונקציית הבנאי של האובייקט.

הנה דוגמה מעט יותר מעשית:

```
class User {
  readonly userId: number;
  userName: string;

  constructor(userId: number, userName: string) {
    this.userName = userName;
    this.userId = userId;
  }

  getUserStatus(): string {
    // Some Call or Business logic
    return 'admin';
  }
}

let userA = new User(1, 'Ran');
userA.userId = 5; // Cannot assign to 'userId' because it is a read-only property.
```

הדוגמה הזאת אומרת בעצם שם אני אנסה לשנות את ה-`userId` לאחר איתחול הקלאס, אני אכשל. זה שימושי במיוחד לא בדוגמאות פשוטות אלא במקרה שיש בו קלאסים כאלה. תחשבו למשל על מערכת שיוצרת אובייקט `user` ומכניסה אותו כאובייקט גלובלי או כאובייקט בסטיטי גלובלי כמו למשל בראיקט, ואם המילימ סטיטי גלובלי לא אומרת לכם כלום, כשתלמידו פרימורקים או ספריות מתקומות ריאקט או אנגולר, אתם תדעו מה זה. כרגע תחשבו למשל על מערכת שברגע החיבור יוצרת בסkop הגלובלי אובייקט `user` עם ה-ID הזה. לא מעט קוד שיוצר קריאות AJAX ומפוזר במסמך שלנו מסתמך על ה-ID `user` כדי לשלוח בקשות AJAX. למשל כשמשתמש מעדכן את פרטיו, מבצע רכישות

ועוד שפע של פעולות שאפשר לעשות במשחק. ואז קוד שמיישחו כותב כמה שנים אחרי זה ומשנה את `getPlayer` באמצעות_BT מומנט אובייקט הגלובלי יכול להביא לברדק רציני וזה דרך למנוע את זה שטייפסקריפט מספקת.

גם פה חשוב להסביר שלא מדובר בסינטקס של ג'אווהסקריפט. ההתראות והשנויות יהיו בטרנספילר של טייפסקריפט. אם תרגלו את הקוד הזה בעורך קוד שתומך בטייפסקריפט או בסביבה של טייפסקריפט, תראו את השנויות הללו. זו לא שגיאה שתראו בקונסולה הרגילה של ג'אווהסקריפט.

קלאס אבסטרקט

בתכנות מונחה עצמים (Object Oriented Programming) יש לנו פיצ'ר משמעוני שנקרא אבסטרקט (abstract). הוא קצר זר לג'אווהסקריפט אז אני אסביר עליו פה. בגדול, קלאס בג'אווהסקריפט הוא אובייקט לכל דבר ואפיו דנו לעילו בפרק על קלאסים זהה דבר שעלול לבבל מתכנתים שריגלים לעבוד בשפות שהן מונחת עצמים. אבסטרקט, שקיים בטייפסקריפט אך לא בג'אווהסקריפט מאפשר לנו ליצור קלאס "DMA", שלא עומד ברשות עצמו ואפשר רק לעשות ממנו `extend`, ככלומר לרשת ממנו. את הקלאס אנו מגדירים באמצעות המילה השמורה `abstract` לפני השם של הקלאס ואז יש לנו קלאס אבסטרקט. לא ניתן ליצור אובייקט מקלאס אבסטרקט.

כשאנו עושים `extend` לקלאס מקלאס אחר, אנו מקבלים את כל התכונות שלו. קלאס אבסטרקט בעצם הוא קלאס שאי אפשר לעשות לו `new`, אבל אפשר לעשות לו `extend` לקלאס אחר ולקבל את התכונות שלו. זה שימושי כאשר יש לנו `base class`.

אני אציג אדגנים באמצעות הקוד הבא:

```

abstract class User {
  userId: number;
  userName: string;
  constructor(userId: number, userName: string) {
    this.userName = userName;
    this.userId = userId;
  }

  getUserStatus(): string {
    // Some Call or Business logic
    return 'admin';
  }
}

class BasicUser extends User {

  getUserPoints(): number {
    // some logic
    return 100;
  }
}

let userA = new User(1, 'Ran'); // Cannot create an instance of an abstract class.
let userB = new BasicUser(1, 'Ran');
console.log(userB.getUserStatus()); // admin
console.log(userB.getUserPoints()); // 100

```

הקלאס User הוא קלאס אבסטרקטי, אם אני אנסה ליצור ממנו אובייקט אני אקבל שגיאת טייפסקריפט.

הוא שימושי כאשר אני רוצה ליצור מ-User קלאס. למשל BasicUser ועוד להוסיף לו תכונות או (במקרה זהה) מתודה. כשהאני עושה extend אני מקבל את כל המתודות והתכונות של הקלאס המקורי ויכול להוסיף עליה משלי.

קלאס אבסטרקטי הוא תבנית. צו שאני יכול לחת אותה ולהרחיב אותה אבל אני לא יכול ליצור ממנה יישירות קלאס. אם אתם עובדים או תעבדו במערכת מונחית אובייקטיבים שיש בה הרבה קלאסים שהם בסיס,סביר להניח שתראו את ה-abstract הזה לא מעט. אם תעבדו בסביבה או בחברה שהקוד שלו הוא מונחה עצמים.

סביבת עבודה אמיתית של טייפסקרייפט

חשוב להבהיר שתת פרק זה הוא לנכallow המעניינים לעבוד עם ג'אוּהַסְקְּרִיפְט באופן עצמאי ולא נסמכים על פוריימומורקים וספריות כגון ריאקט, Next.js, Node.js ודומיהם. לרוב הקוראים תת הפרק יהיה לא רלוונטי כיוון שסביבת העבודה של הפוריימומורקים והספריות כוללת את טייפסקרייפט ואין צורך להתקינה בנפרד.

סביבת העבודה היחידה שלמדנו עד כה היא סביבת העבודה שהאתר של טייפסקרייפט מספק. בעוד שהסביבה הזאת מצוינת ללמידה והתרנסות, היא לא רלוונטית לפרויקט אמיתי. בפרויקט אמיתי אנו צריכים להתקין תוכנה שתמוך את הטייפסקרייפט שאנו כתבבים לג'אוּהַסְקְּרִיפְט באופן מיידי. ככלומר שאנו יכולים לכתוב טייפסקרייפט ב- Visual Studio Code, לשמר, להכנס לדףדף, לטען מחדש ולראות את השינויים. על מנת שהזה יקרה, יש צורך בתוכנה שתבצע את הממרה הזאת מיידית. התוכנה הזאת מותקנת על המחשב ורצה כשאנו מפעילים אותה. המטרה שלה היא המירה של קובץ בסימות ts לקובץ בסימות dz כאשר הממרה עצמה היא גם יוצרת קובץ dz וגם המרה של הטייפסקרייפט לג'אוּהַסְקְּרִיפְט שהדף יודע לטפל בו.

ריאקט, אングולר וספריות נוספות מכילות את הממרה הזאת באופן אוטומטי בבניית הספרייה כיוון שאנחנו לומדים ג'אוּהַסְקְּרִיפְט וטייפסקרייפט נקי, אנו נלמד בחלק זה כיצד לבנות סביבת עבודה עם תוכנת tsc שמבצעת באופן אוטומטי את הממרה. אם אתם מתכוונים להמשיך לספר המלמד ריאקט, חלק זה מיותר עבורכם. ניתן לעשות אותו עבור ההתרנסות ולהלמیدה כמובן.

על מנת להתקין את סביבת העבודה, כדאי שנבין מה אנחנו עושים. אני אוצר קובץ בסימות ts. למשל ts script.ts בפרויקט שלי. המטרה שלי היא לבנות תהליך שבסיומו יוצר קובץ בסימות dz בתיקייה dist/ או בתיקיה אחרת שארצה.

כדי להתאמן, אנו ניצור תיקיה של פרויקט, למשל typescript-project.ts. אנו ניצור תיקית src ובתוכה קובץ ts index.ts. בקובץ נכניס את קוד הטייפסקרייפט הבא:

```
function greeting(sentence:string):void {
  console.log(sentence);
}
```

```
let text: string = 'Hello World';

greeting(text);
```

אפשר כמובן להכניס כל קוד שרוצים. אם אתם משתמשים בעורך הקוד הפופולרי ייז'ואל סטודיו קוד, תוכלו לראות את השגיאה שיש מיד. הנה צילום מסך של אותו קוד, אבל בו הכנסתי במשהו מיותר ולא טקסט, דבר שטייפסקריפט מתריעה בכך מיד ועורך הקוד מראה את זה.

```
src > TS script.ts > ...
1  func let text: string
2  co Type 'number' is not assignable to type 'string'. ts(2322)
3  }
4  View Problem (xF8) No quick fixes available
5  let text: string = 1;
6
7  greeting(text);
8
```

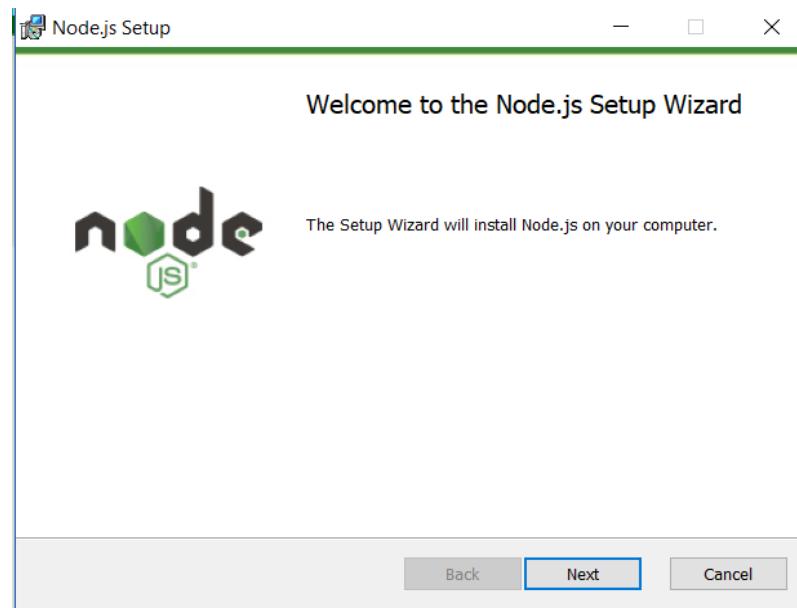
השלב הנוסף הוא ליצור קובץ index.html בתיקות dist ולודא שה-file script.js שלו מקשר אל script.js. אם נפתח את הדף שנמצא בקובץ זהה, אנו נקבל שגיאה, כיון שקובץ js לא מוכן. המטרה שלנו היא ליצור אותו באוטומטי.

התוכנה הממירה קבצי טייפסקריפט לג'אווהסקריפט נקראת תוכנת tsc. היא תוכנה שכותבה ב-node.js, סביבת ג'אווהסקריפט לצד השרת. Node.js אינה קשורה לדף השרת ומיצעת פעולות על שרתים ומחשבים ונitin להתקין אותה על המחשב האישי. על Node.js ניתן ללמידה בהרחבה בספר "לימוד node.js בעברית" וכן אנו רק נתקין אותה היא פשוטה לכל מערכת הפעלה – מחולנות ועד מק. את Node.js ניתן להתקין על ידי כניסה לאתר הזה:

<https://nodejs.org/en/download>

ובבחירה במערכת ההפעלה שלכם והפעלת הקובץ. כיון שלרוב הקוראים יש חלונות, אני אציג את תהליך ההורדה באמצעות חלונות. במק ובלינוקס המרכיבים דומים מאוד.

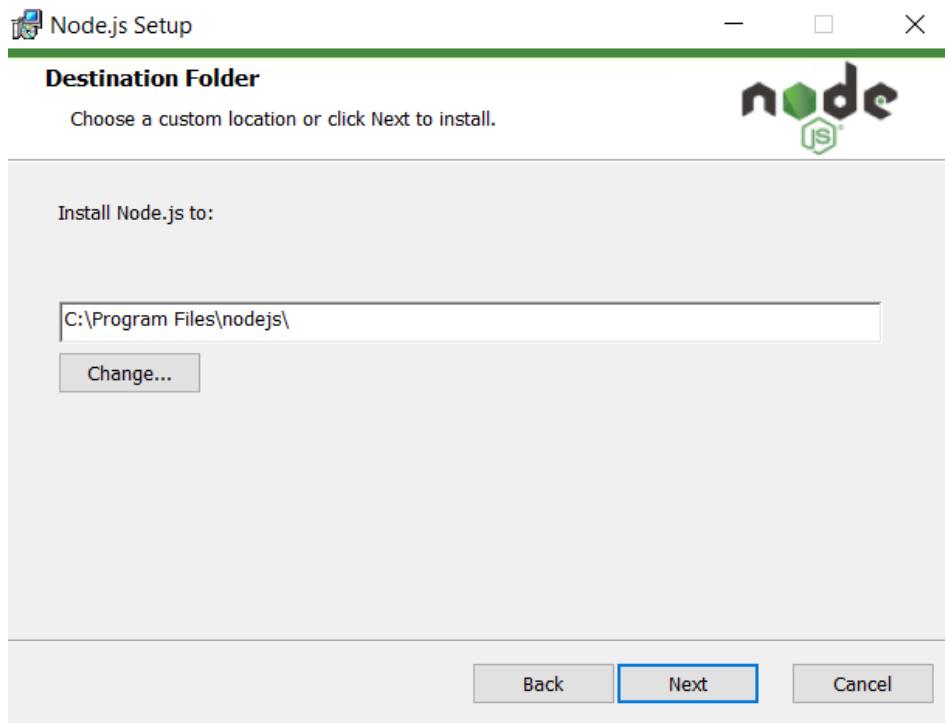
אחרי הפעלת התוכנה שירדה, אנו נקבל את אשי'ף ההתקנה:



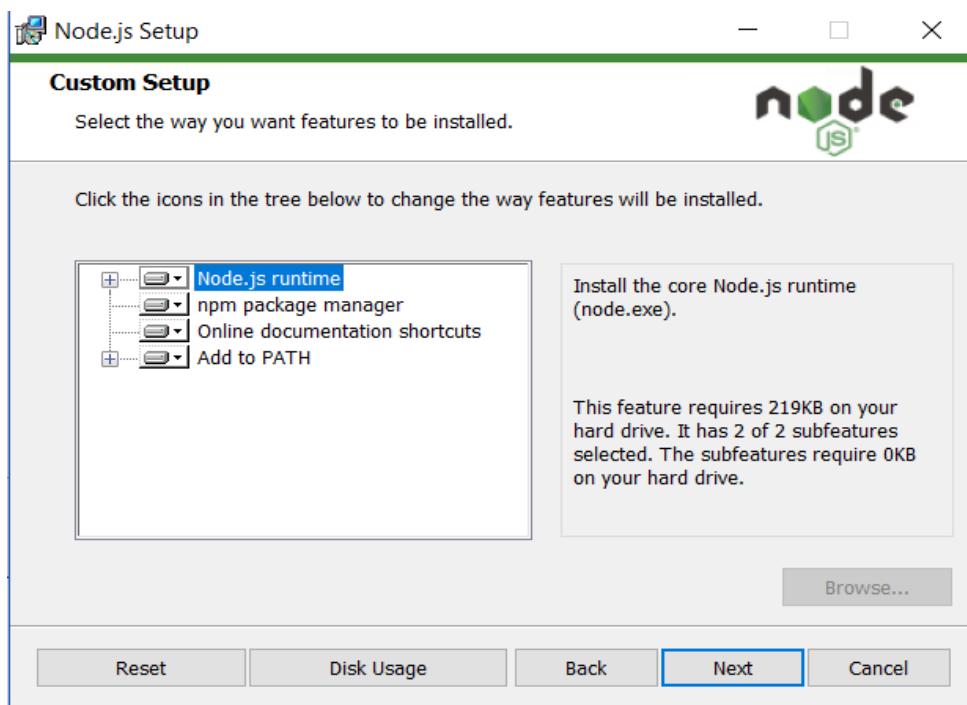
מקבלים את התנאים:



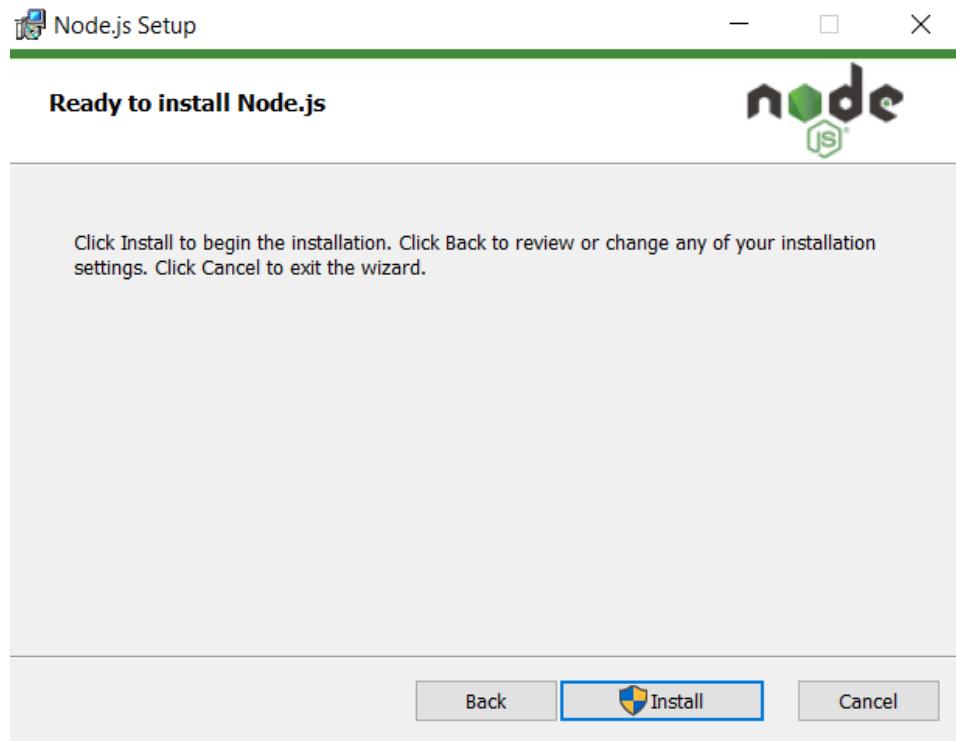
לוחצים על next:



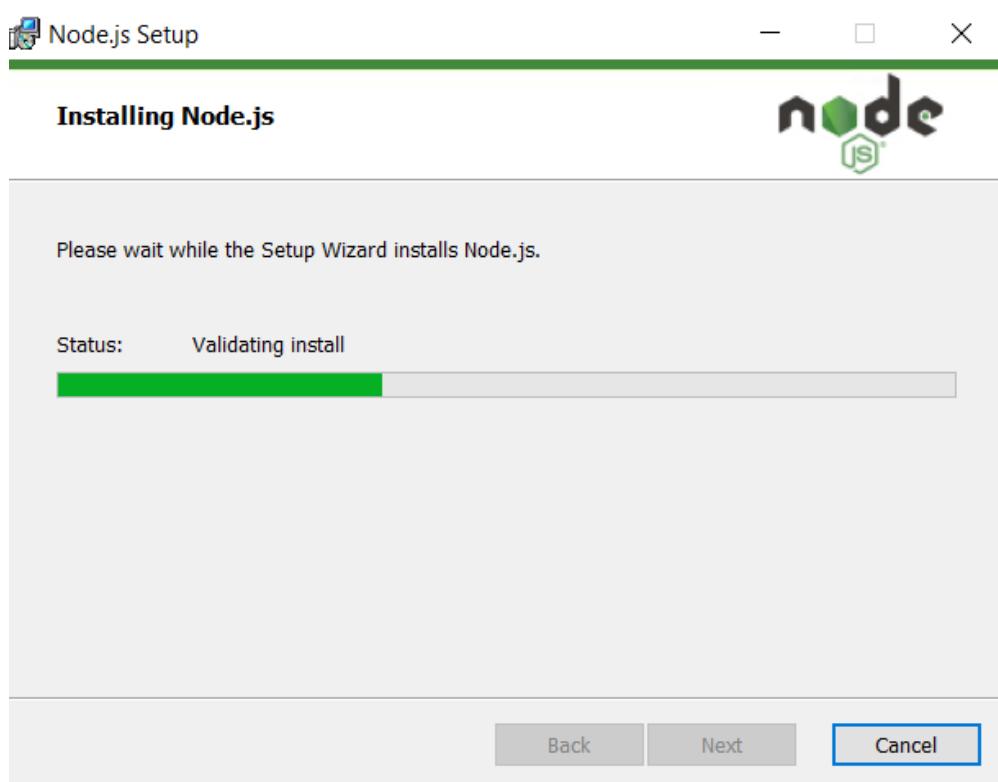
ישוב על next:



לחיצה על Install תתקין לבסוף את התוכנה:



כל מה שנותר הוא לחכות לסיום ההתקנה:

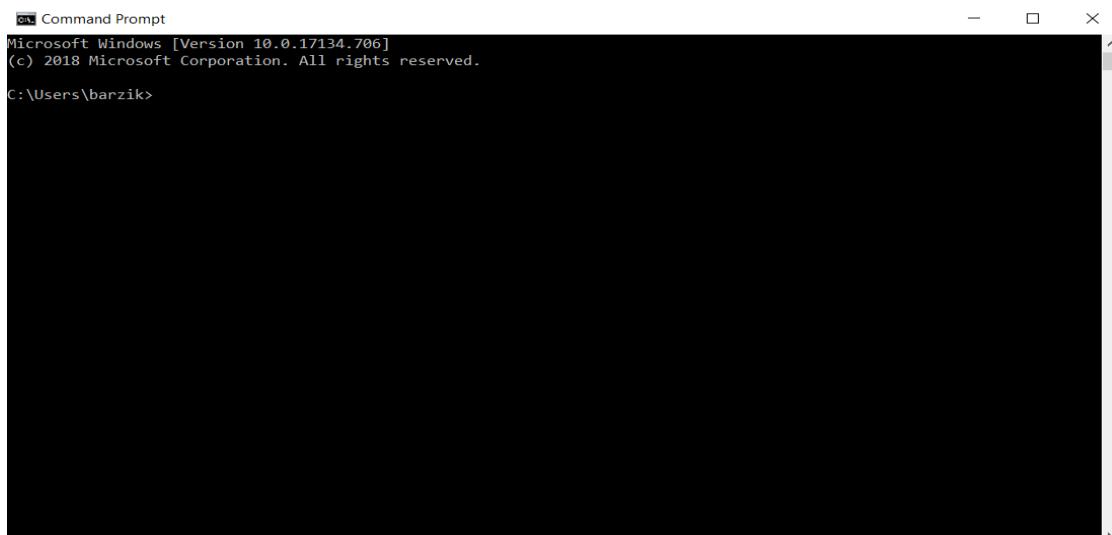


.זהו. אנו יכולים לעבוד עם Node.js

במידה זהה נראה לכם מהיר מדי או מוזר מדי, זה בסדר גמור – מדובר בהנחות בסיסיות וראשוניות להתקנה. אם איןכם רוצים למדוד על `js.Node` (למרות שהיא פופולרית וחשובה גם למי שרוצה למדוד ריאקט), זה מה שצריך לדעת.

אחרי שתוכנת `js.Node` מותקנת, אנו יכולים להתקין את `tsc`. אך לפני כן علينا להכיר לו רק באופן שטхи את הטרמינל. תוכנת `tsc` אינה תוכנה עם משק גרפי אלא תוכנה שעובדת בטרמינל. טרמינל, שנמצא גם בחלונות, גם במק גם בלינוקס היא משק שמאפשר לנו להזין פקודות טקסטואליות למחשב. אם אני רוצה למשל למחוק קובץ, במשק גרפי אני נכנס לסייר הקבצים, נכנס אל התקינה שבה הקובץ ממוקם ואז גורר אותו אל סל המחזור. בטרמינל אני עושה את אותן פעולות בדיק, אך באמצעות פקודות טקסטואליות.

בחלונות מגיעים לטרמינל באמצעות לחיצה על זכוכית המגדלת (בחלונות 11) והקלדה של `cmd` – ראש תיבות של `cmd`. נגיע לחלון שנראה כך:



פתחו את החלון הזה, הקלידו `notepad` ולחצו על `enter`. יפתחת ה-`Notepad` של חלונות. ברכותי! הפעלתם תוכנה באמצעות שורת הפקודה. הקלידו `calc` ולחזו על `enter`, תוכנת המחשבון תיפתח. זו עוד תוכנה שהפעלתם באמצעות משק הפקודה. המשק הזה, או הטרמינל בשפת העם, הוא סביבת העבודה של `Node.js`.

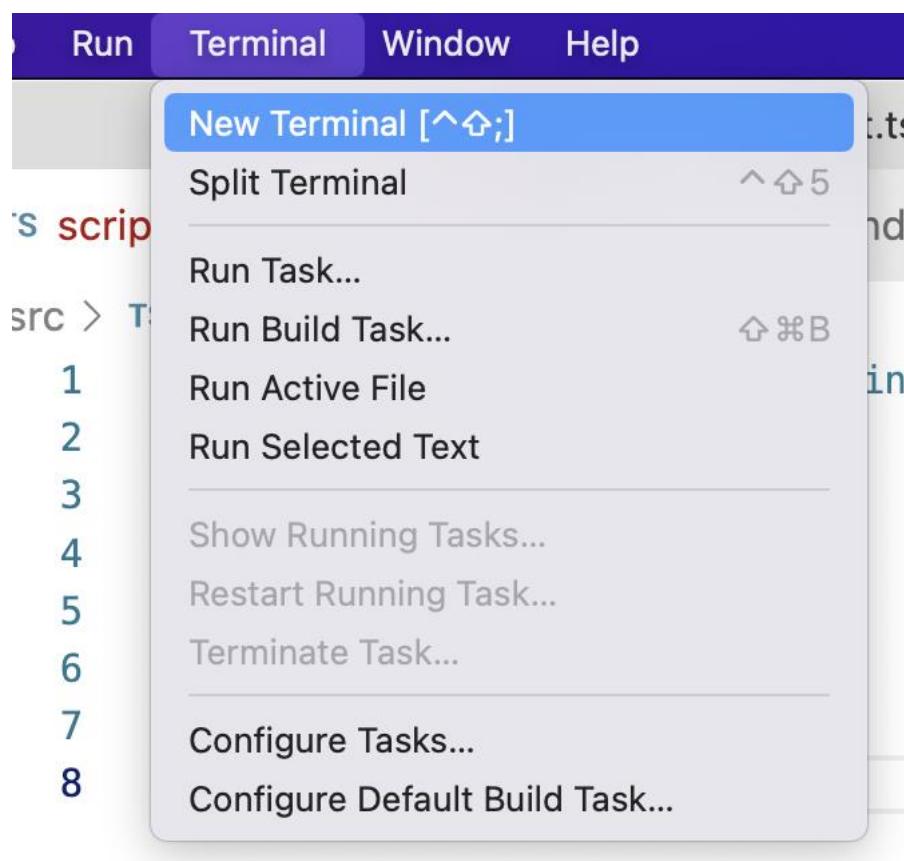
כיוון שהתקנו כבר את `js`, `Node.js` (אם נכנסתם לטרמינל לפני ההתקנה של `Node.js`, אנא סיגרו את הטרמינל ופתחו אותו מחדש), אנו יכולים להתקין את `tsc`. צריך לוודא שהוא נמצא באותו ספריה שבה יש את הפרויקט.

ההתקנה נעשית באמצעות הקלדת הפקודה:

```
npm install typescript --save-dev
```

אחרי כמה דקות, תוכלו להמשין.

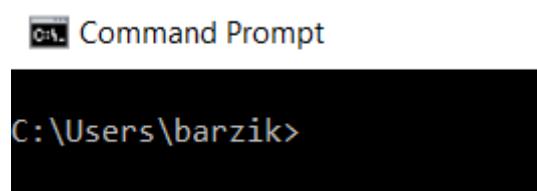
ניתן להכנס אל הטרמינל גם באמצעות ייז'ואל סטודיו קוד. בחלקו העליון של עורך הקוד כתוב `Terminal`. לחיצה עליו ובחירה באפשרות `New Terminal` תפתח חלון בתחתית העורך שמאפשר לנו להכנס לטרמינל.



יש את האפשרות הזו בכל עורך קוד שהוא ומאוד מקובל לעבוד עם הטרמינל דווקא מעורך הקוד ולא להכנס מה-`cmd` או מתוכנת ה-`terminal` במק או בלינוקס.

אחרי שהתקנו את תוכנת `tsv` במחשב, אנו יכולים לבצע את הממרה. אנו ננווט אל התיקייה.
אבל איך מנוטים בטרמינל?

הטרמינל תמיד נפתח בהקשר של תיקייה כלשהי. תמיד אנחנו "מצאים" בתוך תיקייה.
בדרכן הכל כשאני פותח טרמינל, הוא נפתח במקום של המשתמש שלי. כך למשל, אם
אני נכנס ל-`cmd` במחשב שלי – אני רואה את המיקום שלי:



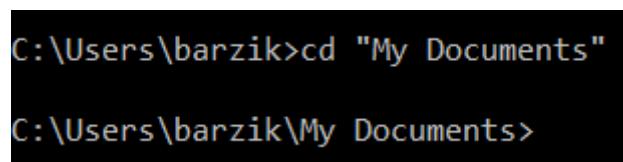
Command Prompt

```
C:\Users\barzik>
```

אפשר לראות שאין נמצא בכונן C, בתיקיית `Users` ובתת התיקייה `barzik`, שהוא שם המשתמש שלי בחלונות. אם אני אפתח את סייר הקבצים, אני אוכל לנוט לתיקייה זו.
הטרמינל הוא דרך פשוט נוספת לשוטט במחשב ולפעול בו – דרך שהיא לא גרפית, אבל
כל מה שאני יכול לעשות במכשיר הגרפי אני יכול לעשות בטרמינל.

כדי לראות את רשימת הקבצים בתיקייה, אני צריך להקליד `dir`. הקלדה של `dir` ואז `enter` תראה לי את רשימת הקבצים שיש בתיקייה שבה אני נמצא. רשימת הקבצים זו תהיה זהה לchlolutin לרשימה של הקבצים שאני רואה בסיר הקבצים כשהאני נכנס לאותו מיקום. אם אצור קובץ או תיקייה בסיר הקבצים וקליד שוב `dir` בטרמינל כשהאני באותו המיקום של סייר הקבצים, אוכל לראות את הקובץ או את התיקייה בטרמינל.

כדי להכנס לתיקייה מסוימת, אני צריך להקליד `cd` ואז את שם התיקייה ואז `enter`. אני יכול להשתמש במקש TAB על מנת לבצע השלמה אוטומטית. אם יש רוחה בשם התיקייה,
אני צריך להקיף אותו במירכאות. אם אני משתמש ב-TAB הוא יעשה זאת זה עבורי.



```
C:\Users\barzik>cd "My Documents"
C:\Users\barzik\My Documents>
```

אם אני רוצה לחזור לאחר מכן, אני אכתוב `cd ..` שתי הנקודות יעלו אותי לתיקיית האב. אם
אכתוב `cd ../../` אוכל לחזור לתיקיית האב של האב וכן הלאה.

```
C:\Users\barzik\My Documents>cd .../...
C:\Users>
```

| פקודה בLINQ/MK | פקודה בחלונות | פקודה |
|----------------|---------------|---------------------------------------|
| ls -al | dir | הציג את רשימת הקבצים והתיקיות בתיקייה |
| cd | cd | העבר לתיקייה אחרת |
| exit | exit | יציאה מהטרמינל |
| clear | cls | ניקוי המסך |

אם אנו פותחים את הטרמינל מעורך הקוד, הוא יפתח אוטומטית במקום של התיקייה של הפרויקט שלנו.

אנו נזודא שאנו בתיקייה הראשית של הפרויקט שלנו. אם נקליד dir (בחולנות) או al-ls (בקמ או LINQ/MK) אנו נראה את התיקיה dist ואת התיקיה src. אנו נקליד את הפקודה tsc init. הפקודה זו מתחילה את הפרויקט ליעודה עם טייפסקרייפט ויוצרת קובץ JSON בשם tsconfig.json שיש בו את הגדרות הפרויקט השונות. הקובץ הזה הוא חשוב ויש בו לא מעט הגדרות שימושיות על אופן הקימפוף של טייפסקרייפט בפרויקט. כך למשל, אנו יכולים לקבוע את הגרסה של ג'אוּהסְקָרִיפֶט שטייפסקרייפט יקומפל אליה. איזה קבצים וסימומות יכולות בתהיליך וכן הלאה. אנו רק נשנה הגדרות ספציפיות ומינימליות.

אנו צריכים לוודא שהגדרות יש:

```
"outDir": "./dist",
```

יכל:

```
"include": ["src"]
```

באופן הבא:

```
{
  "compilerOptions": {
    "outDir": "./dist",
  },
  "include": ["src"]
}
```

אם יש הגדרות נוספות, אז כמובן תשמרו עלייהן, אבל זו ההגדרה המינימלית שציריך. הקפידו שיש את שתי ההגדרות האלו שם. מה שהן אומרות זה שתיקית הקלט שלנו (כלומר התיקיה שבה יש את קבצי הטייפסקריפט, במקרה שלנו קובץ אחד בשם script.ts) היא התיקיה ששמה הוא src והתיקיה שאליה הגיע קבצי הג'אוوهסקריפט אחרי הקימפול היא תיקיות dist.

ועכשיו אנו נקליד בטרמינל את הפקודה:

```
tcs --watch
```

הפקודה הזו רצה וראשת היא תיצור לנו ב-dist את קובץ הג'אוوهסקריפט! אנו יכולים לבדוק את הקובץ באמצעות עורך הקוד שלנו ואם נפתח את קובץ ה-index.html, נוכל לראות אותו רץ. זה קובץ ג'אווהסקריפט שהוא מכירinos. כל עוד התוכנה הזו רצה, כל שינוי שהוא עשוי בקובץ הטייפסקריפט שנמצא בתיקית ה-src, יתורגם אוטומטית לקובץ ג'אווהסקריפט מבלי התערבות שלנו. אם סגרנו את הטרמינל או עורך הקוד, נצטרך להקליד את tcs --watch שוב.

זה נראה מסובך וזה אכן מסובך יחסית. רוב מתכנתיו הג'אווהסקריפט משתמשים בטייפסקריפט אינט מרים סביבה כזו אלא משתמשים בסביבה מוכנה – בין אם מדובר באנגולר, ריאקט או Vue בצד הלköח או באקספרס או סביבות אחרות ב-node.js. אם אתם רוצים למדוד טייפסקריפט בלבד על מנת להשתמש בו בריאקט למשל, לא צריך לעשות את זה כי סביבת ריאקט כבר תעשה את זה עבורכם.

סיכום

לסיכום, למדנו על טייפסקריפט וחשיבותה. למדנו על ההבדל בין טייפסקריפט לג'אווהסקריפט ועל כך שטייפסקריפט היא superset של ג'אווהסקרימפט – קלומר אותה שפה אך עם תוספות חשובות. למדנו על קלט ועל פלט ועל גנריות וכן טייפסקריפט יכולה לשיע לנו ליצור קוד עם פחות באגים. למדנו גם על בניית סביבת פיתוח אמיתית לטייפסקריפט.

טייפסקריפט הולכת ותופסת תאוצה משמעותית בעולם הפיתוח. בעתיד, הנספחזה ישולב בספר עצמו וייתכן שגם שם של הספר ישונה מהשם "לימוד ג'אווהסקריפט בעברית" אל השם "לימוד טייפסקריפט בעברית". המלצתה שלי היא להשתמש בטייפסקריפט בכל הפרויקטים שלכם. קטנים גדולים גם בתהילך הלימוד.

תרגילים

תרגיל

כתבו משתנה, הגדרו את הסוג שלו כטקסט, הכניסו לתוכו ערך והדפיסו אותו בקונסולה.

פתרון

```
let name: string = "Moshe";
console.log(name);
```

הסבר

השתמשתי ב-`string`: על מנת להגיד את המשתנה. השאר מתנהג כרגיל.

תרגיל

צרו פונקציה המתקבלת שני משתנים מסוג מספר ומחזירה את הסכימה שלהם. חשוב לפרט גם את סוג הארגומנטים וגם את סוג המידע של הפלט.

פתרון

```
function add(a: number, b: number): number {
    return a + b;
}
console.log(add(1, 2));
```

הסבר

יצרנו פונקציה שנקראת `add`. יש לה שני ארגומנטים. הראשון הוא `a` וציינתי את הסוג `number`: השני הוא `b` וציינתי את הסוג שלו גם כן. הפלט של הפונקציה מסומן כמספר עם `number`: אחרי הסוגרים שבהם אני מפרט את הארגומנטים.

תרגיל

צרו משתנה שיכול לקבל סוג מידע של מספר או מחרוזת טקסט.

פתרון

```
let value: number | string;
value = "hello";
value = 10;
console.log(value);
```

הסבר

השתמשתי בסימן | על מנת לומר או סוג מידע מסווג מחוץ טקסט או סוג מידע מסווג מספר.

תרגיל

צרו type בשם NameAndAge שמספרת סוג מידע של מערך שהאיבר הראשון שלו הוא מחוץ טקסט והאיבר השני הוא מספר.
הגדר משתנה מסוג המידע שלו הוא NameAndAge והכנס לתוכו מידע.

פתרון

```
type NameAndAge = [string, number];
let moshe: NameAndAge = ["Moshe", 66];
console.log(moshe[0]);
```

הסבר

השתמשתי במילה השמורה type והגדרתי סוג מידע מסויל בשם NameAndAge. מהו סוג המידע הזה? מדובר בערך עם שני איברים. הראשון הוא טקסט והשני הוא מספר. ציינתי זהה מערך באמצעות [] והכנסת האיברים שאני צריך.
אחרי שהגדרתי את הסוג, אני יכול להשתמש בו לבדוק כמו כל סוג מידע באמצעות נקודתיים ואז שם סוג המידע שהגדרתי.

תרגיל

צרו type של אובייקט בשם PersonType שיש לו את התכונות הבאות: name, age, address טקסט, שהוא מספר ו- address טקסט אבל התכונה הזו אופציונלית.
הכנס אובייקט למשתנה שסוג המידע שלו הוא PersonType.

פתרון

```
type PersonType = {
  name: string;
  age: number;
  address?: string;
};

let moshe: PersonType = {
  name: "Moshe",
  age: 66
};

console.log(moshe.name);
```

הסבר

אנו מגדירים סוג מידע של PersonType. אנו משתמשים בסימן שאלה "?" על מנת לציין שתכונת address היא אופציונלית.

תרגיל

צרו פונקציה בשם max. הפונקציה הזו מקבלת שני ערכים מסוימים אותו סוג (יכול להיות כל סוג טקסט, מספר, בוליאנו) ומחזירה את הגדל מביניהם. המשתמש צריך לפרט את סוג המידע.

רמז: צריך להשתמש בו בגרסאות שמאפשרת לשימוש לפרט את סוג המידע.

פתרון

```
function max<T>(a: T, b: T): T {
  return a > b ? a : b;
}

console.log(max<number>(5, 10));
```

אנו משתמשים ב-<T> כדי לאפשר למשתמש להגדיר את סוג המידע. T משמשת אותנו כסוג מידע לכל דבר ועניין. אנו מגדירים ש-T יהיה הקולט a והקלט B וגם הפלט. העבודה שיש לנו <T> בתחילת הפונקציה לא צריכה לבלב אותנו. זה בדוק כמו כל סוג מידע אחר. אפשר לדמיין ש-T הוא `string`, או `number`. כל עוד זה אותו דבר. כשאני משתמש בפונקציה, אני צריך לפרט מהו ה-T שלו. במקרה הזה `number`.

נספח: Best Practices

מחברים: שחר טל, רוני אורבן, דניז רוזג, נופר ברנס, חברת Really Good – בוטיק ReallyGood.co.il. End, מאז 2012.

מה זה Best Practices ולמה כדאי לישם אותן?

Best Practices הם כלליים, המלצות ומוסכמות שנעודו לשיעור תוכנות אינטואיטיביות, בין השאר באמצעות כתיבת קוד בצורה טובה ונכונה שמזהערת טעויות ובלבול.

תוכנות הוא פועלה חברתית. כשבושים על תוכנות, לרוב מדמיינים האקר בודד שיושב בחושך בקופצון ומclid במרץ בלי לחשוב שנייה, כאילו הקוד עף מתוכו. הסיטואציה הזאת היא נדירה, ובינינו, זה די,Nooch לשבת לתוכנת כהה. לרוב תוכנות נכתבות עם קולגות במקום העבודה או עם חברים או אפילו עם אנשים זרים באינטרנט – והיצירה שלhn דורשת קצת יותר סבלנות ושיקול דעת. מתכנתים אחרים יסתכלו על הקוד לפני שהוא יוכנס באוף סופי לפרויקט (אדרט על `Code Review` בהמשך), ומתכנתים אחרים יצטרכו להבין ולבוד אותו בעtid כשהירצו להוסיף, לשנות או לתקן את התוכנה.

לאורך הזמן מתכנתים עוזבים צוותים אחרים צריכים להתמודד עם קוד שהכתב שלו כבר לא בסביבה, וגם אתם בטח תרצו להשאיר אחריכם קוד שמתכנתים אחרים יכולו ואולי ישמשו לעבוד איתו. אולי בפרויקט שלכם שלכם, ככל שהזמן עבר גדל הסיכון שלום אחד תסתכלו על קטע קוד ותשאלו את עצמכם "מי לעזאזל כתב את זה?" והתשובה הטריגית-קומית תהיה – אתם בעצמכם. לכן חשוב לשמור על קוד מסודר ומארגון שנכתב בדרך מוסכמת. כך מוצאים באגים וטעויות, מקלים על כולם את קריאת והבנת הקוד במהירות וחוסכים רushi רקי ויקוחי סרך על שיטות – מחליטים פעם אחת על סגנון

ומדייניות ונמנעים מ"מלחמות עירכה" אינסופיוות שבהן כל אחד מסדר את הקוד איך שבאו והבא אחריו מבטל את השינויים.

אפשר להסכים על כללי בעל פה, אפשר בכתב, ויש כללי שадון בהם בהרחבה ממש כאן, שיכולים לעזור לאנוף לפחות חלק מהכללים באופן אוטומטי.

במיוחד בשפה גמישה כמו ג'אווהסקריפט, שמאפשרת לעשות המונם דברים יצירתיים ובתוכם המונם שיטויות, מומלץ לעקוב אחרי Best Practices מקובלים בתעשייה, שאוליל לא נכתבו בدم אבל בהחלט עלו ביעז ובד מעות. מעבר למניעת שגיאות ולשימוש מושך באפשרויות שהשפה מציעה, חלק מה-Best Practices הם העדפות סטיליסטיות של המתכננים בפרויקט, שהתקבעו ושהוצאות רוצחה לשמר – בשני הסוגים יש מקום לשיקול דעת בריא, ולא צריך לקחת הכלול כתורה מסוימת בלי להבין ולבחון מה מתאים לכם.

דוגמה להמלצה שמונעת טעויות היא תמיד להשתמש בהשוואה קפנית, כלומר ב-`==` או `!=` במקום ב-`=` המתירנית יותר, שמבצעת המرة לסוג המשתנים לפני השוואה (`coercion`).

לא מומלץ:

```
if (numberOfThings == possiblyNotANumber) { /* code */ }
```

מומלץ:

```
if (numberOfThings === possiblyNotANumber) { /* code */ }
```

דוגמה לכל סובייקטיבי יותר, שפושוט מבטא העדפה סגונית ועזר לשומר על אחידות, היא ריווח בין חלקיקי קוד שונים, למשל ריווח בין שם פונקציה לבין הסוגרים שגדירים את הארגומנטים:

אופציה א':

```
function jump(x) { }
```

אופציה ב':

```
function jump (x) { }
```

אפשריה ב', היא המועדפת علينا כי היא מאפשרת להבדיל בחיפוש בין הגדרת הפונקציה לкриאות אליה.

סת הכללים וההמלצות יכול להיות שונה מאוד בין פרויקטים שונים, ובעובדות צוות יש חשיבות בשם העקביות והשפיות להתגבר על העדפה האישית שלהם ולכבד את מה שכבר נקבע – מומלץ לדבר בפתחות ולבחון ביחיד צורך בשינויים.

שכל מפתח מקצועי צריך להכיר Best Practices

בחירה שמות

משתנה או פונקציה עם שם טוב יאפשרו לכם להבין מיד מה התפקיד שלהם. אם השם לא מצליח תצטרכו לראות איפה הגדירו את המשתנה ומה הוא מאחסן בתוכו או לקרוא את הפונקציה ולפענח מה ניסו לעשות בה. פתרון אפשרי הוא להוסיף הערה, אך עדיף פשוט לבחור שם טוב. שם טוב הוא קצר וקולע יותר מהערה, מעבר לכך שмотב לא לפזר הערות ליד כל מקום שבו משתמשים במשתנה או בפונקציה.

איך בוחרים שם טוב? שיטה מקובלת עבור פונקציות היא שיטת-ה-חונך-verb (פועל-שם עצם).

לדוגמה:

- `getStudents`
- `sortColorsByBrightness`
- `setLunchBreakTimer`

למשל הוא שם יותר טוב מ-`setTimer`, שעלול להיות כללי מדי, וכשנתקלים בו צריך להזכיר באיזה טיימר מדובר.

יש גם יוצאי דופן בולטים – שמות משתנים סטנדרטיים צריכים להכיר ואין צורך להסביר או לפרט אותם. דוגמה לשמות כאלה הם `x` ו-`y` בהקשר של פיקסלים או כיוונים על המסן, או `o` ו-`o` בשימוש הקלאסי שלהם בולאות. אבל חוץ מהם, כדאי לזכור שכמעט תמיד משתנה של אותן או שתים זה רעיון רע. אין שום יתרון בראשי תיבות מסוורים על פני מילים מובנות.

שם המשתנה הוא לא הדבר היחיד שעוזר לקרוא את הקוד. אפשר לכתוב אותו שם בכתמה דרכים בעזרת אותיות גדולות וקטנות. כל דרך כזו נקראת "קיס", ומיענדת לייצג סוג משתנה אחר.

אומנם השם זהה, אבל על כל אחד מהמשתנים האלה נוסף רוֹבֵד של משמעות שנרמז רק מבחןת הקיש:

camelCase

`lunchBreakTimer`

כל המילים מלבד הראשונה מתחילה באות גדולה; אין רווחים בין המילים.
קיים זה נמצא בשימוש עבור כמעט כל המשתנים והפונקציות בג'אוּהַסְקְּרִיפְט.

PascalCase / TitleCase

`LunchBreakTimer`

כל המילים מתחילה באות גדולה; אין רווחים בין המילים.
קיים זה נמצא בשימוש עבור קלאסים, וככה מקל את הבדיקה בין קלאס ל-`instance`.

SCREAMING_SNAKE_CASE / UPPER_CASE

`LUNCH_BREAK_TIMER`

כל האותיות גדולות; רווחים בין המילים הופכים לקו תחתון.
קיים זה נמצא בשימוש בעיקר קבועים, שבדרך כלל יוצבו במשתני `const`.

kebab-case

`lunch-break-timer`

כל האותיות קטנות; הרווחים בין המילים הופכים לקו מפריד.
נמצא בשימוש ב-HTML וב-CSS עבור שמות של `pi`, קלאסים ואלמנטים.
קיים זה לא נוח לשימוש בג'אוּהַסְקְּרִיפְט כי קו מפריד אינו תואקוי בשמות משתנים, וכך
שקשה עד בלתי אפשרי להשתמש בו ברוב המקרים.

snake_case

`lunch_break_timer`

לא בשימוש בג'אוּהַסְקְּרִיפְט בדרך כלל.

KISS

"*Keep It Simple, Stupid*" אמירה שקלעה בול כשנטבעה לפני יותר מחמשים שנה על ידי מהנדס המטוסים קל ג'ונסון בלוקהיד, וудין מתאימה כהנחה לתוכנים של 2020

והלאה. כתבו את הקוד בדרך הפשוטה והקצרה ביותר, כל עוד הקריאות נשמרות. למשל, הימנו מסינטקס איזוטרי שלא נמצא בשימוש רחב ומהתחכמת יתר בכלל.

DRY

לאוthon פונקציונליות בכמה מקומות אבל מהססים להפוך אותה לפונקציה מכיוון שמדובר במקרה פשוט וקצר או במקרה שדורש שינוי קטן בכל שימוש שלו.

אל תהססו. כשעוטפים את הקוד בפונקציה גם נתונים לו שם, שהופך את השימוש בו לקריאה יותר. ככל שימושים יותר קוד שלא לצורך, מגדילים את הסיכון שהמתכנת הבא יפספס חלק מהמקומות שבהם יש קוד כפול, והתחזקה התהפהן לקשה ורצופת תקלות.

זכור – הפך מ-WET הוא DRY, ראשית תיבות של **Write Everything Twice** (-;)

לא להמציא את הגלגל

למרות שתכנות זה כיף ומעניין, לא בכל דבר שבונים צריך להקים הכלול מאפס. וכך נראה ברור כיום שתשתמשו בספרייה כלשהי שתעזר לרנדר את הממשק, נניח Riotkit, ככה חשוב להשתמש גם בספריות ובכלים קטנים ומוקדים יותר כשלשה הגיוני. מהווצאת בקשوت לשרת עם axios, דרך ספריות לפועלות נפוצות על מערכים ואובייקטים כמו lodash ועד לאלפי הרכיבים המוכנים לרוב חלקי הממשק שתרצו לבנות – תהיו בטוחים שמיישהו כבר פיתח פתרון טוב כמעט לכל צורך. הבה נודה באמת – אתם לא הראשונים שכותבים Color Picker או Dropdown עם השלמה אוטומטית, וגם אלגוריתמים לערובות רשימות כבר נכתבו בעשורים שקדמו להצטרפות שלכם לתהום.

יש שלושה מקרים שבהם דוחק האגוי להיכנס לעובי הקורה ולכתוב בעצמכם משהו לא אלמנטרי:

1. אם אין פתרון מוכן שעונה על כל הדרישות שלכם (עדין זכרו שלעתים יש רכיבים מסוימים יותר בתחום המכלול שיוכלו לעזור לכם או שתוכלו ללמידה מהם, אז תמיד טוב להסתכל עליהם).

2. אם זה ממש התחום שלכם, מה שבשבילו קמתם בבוקר ואתם תרצו לקחת אותו כל כך רחוק שכנראה אף כדי מוכן לא יספיק לכם – למשל במסגרת העבודה ב-Really Good Anchnuo עובדים עם פלטפורמת וידיאו מובילה, אז כשהתבוננו לפתח נגן וידיאו חדש יצרנו אותו מאפס על בסיס תנית ה-video של HTML, בלי אף ספרייה חיצונית לניגון וידיאו.

3. אתם רוצים לתרגל או נבחנים על פיתוח עצמאי של משהו זהה. בהצלחה!

בשולי הדברים, שימו לב לרישיונות בקוד פתוח – רישיונות שונים מאפשרים שימוש בתנאים שונים, וולקם מגבלים שימוש מסחרי. אם אתם חלק מארגון גדול ולא בטוחים, כדאי לוודא שרישיון השימוש מקובל על הגורם המשפטי בחברה. בכלל 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 שפשטית עובדת עם הפקנציונליות הבסיסית. אחר כך חוזרים לעבות ולכ索ות את כל הדרישות, ומשפרים את הקוד כך שהוא קריא יותר, תמציתי ומובן ככל האפשר. אל תtemptו להקדים את המאוחר ולחטווא ב"Premature Optimization" – אופטימיזציה שומרה לסופ.

תיעוד

כבר למדתם מוקדם יותר בספר על הערות בקוד. שני השימושים הכי נפוצים בהערות הם:

1. כיבוי זמני של חתיכת קוד בלי לאבד אותה. נוח בכך כדי פיתוח ובדיקות, אבל שימושו

לב שהערות הן לא שיטה לשמר גרסאות של קוד – מיד אצלול לניהול גרסאות

נכון.

2. תיעוד – להסביר מה מצפים שקרה או למה פיתחתם משהו בצורה מסוימת. תיעוד

טוב מוסף מידע, ולא סתם חוזר על מה שהקוד עשה. אפילו כמה מילות הסבר

עדופות על כלום. הערה שכתובה טוב יכולה לחסוך לתוכנתך אחר הרבה כאבראש.

כשתם ניגשים לכתיבת התיעוד, נסו לחשב מה הדבר הכי מורכב או לא ברור

בקוד שלכם ושימו את הדגש שם. מתכוונים אחרים יודו לכם, ואפילו אתם תודו

לעצמכם כשתחזרו לעבוד על קוד ישן ומתווד היטב שלכם.

ניהול גרסאות

בגלל שקוד הוא בסופו של דבר קובצי טקסט שנערכים על המחשב שלכם, הפתרון הכי

פשוט (ונאיבי) לניהול גרסאות הוא יצירת עותקים של תיקיות בנקודות זמן מסוימות.

במצב זה – קל לאבד קוד וקשה לעבוד בצוות. המחשב יכול לשבוק חיים או שתצטרכו

לשחזר קוד שמחקתם וזה יהיה קשה עד בלתי אפשרי. לא רק זה – אם מישהו מהוצאות

ירצה לשנות קוד הוא יצרך לבורר אצל מי הקובץ החדש ביותר ולבקש ממנו שישלח לו

את הקובץ. באיזו שנה אנחנו?

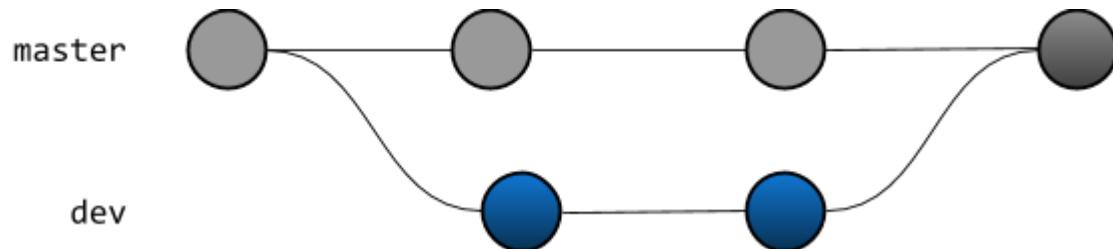
כדי להתגבר על כל הבעיה הללו, משתמשים בפתרון ניהול גרסאות (Version Control או

Source Control). הפתרונות הנפוצים ביותר כיום הם Git ו-SVN. לגיט יש כמה יתרונות

משמעותיים ובראשם מהירות, ביזור שמאפשר יכולת עבודה נוחה ללא חיבור לאינטרנט.

וקהילה עצומה – כמעט כל עולם הקוד הפתוח היום מתנהל על בסיס Git, לרובםgithub.

שימוש בניהול גרסאות מאפשר למפתחים רבים לכתב קוד בemma ענפים (Branches) שונים במקביל ולשתף אותו ביניהם. אפשר לראות איזה מפתח כתוב מה ומתי, אפשר לשלבEmma ענפים שונים אל תוך גרסה אחת שאויה משחררים, והכי חשוב – אפשר לחזור אחוריה במידת הצורך.



בקשת עזרה

כשנתקעים במהלך פיתוח – לבקש עזרה זו לא בושה. אם אחרי Debugging, בידוד הבעיה ובדיקה הלוגים (אם יש כאלה) עדין לא צלחتم את הבעיה, עוברים לגוגל. מנסים לחפש את השגיאה בצורה הכי נקייה שאפשר, בלי שמות קבועים או מספרי שורות מהקוד הספציפי. למרבה המזל, רוב הבעיה נפוצות 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 מכיריים את הבעיה הנפוצות והאתגרים שאתם עושים להיתקל בהם ויכולו להכוין אתכם ולהשוך לכם זמן יקר. שיתוף הבעיה עם חברי הצוות גם גורם 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 טכניות הוא StackOverflow, שמנגן דירוג התשובות בו עוזר לזהות מה מבין התשובותעובד ורלוונטי בעניין הקהילה, בוגוד לפורומים שגם הם רלוונטיים אך בחיפוש פתרונות בהם נדרש קריאה יותר ביקורתית. אין מה להתבזבז, השאלה שתשאלו תהיה נראה רלוונטית למפתחים אחרים בעtid שיתתקלו באותה בעיה ויחפשו תשובה.

בדרכן לפרסום שאלת ה策רנו להכין דוגמה מוקדמת וمبודדת של הבעיה (מכונה בדרך כלל **Reduced Test Case**), וגם זה תהליך מומלץ שעצם ההשערה בו עשוי להוביל אתכם לפתרון עוד לפני שתפרנסמו משהו.

ביקורת עמיתים – Code Review

סקורי הרצה נרבות ככלים, וזה יכול להיראות מאיים. האם באמת מקפידים על כלם בכל צוות פיתוח?

התשובה היא שאםنم לא מקפידים במאת האחזים, אבל מפתחים וארגוני מקצועיים יתעקשו לעמוד בכמה שיטות מהם, כי מוצאים הולכים ומתפתחים וככל שפרויקט מתעבה ונוסף עוד שכבות של קוד כך נהיה קשה לתחזק אותו ולהבין מי נגד מי. אם מראש הייסודות רעוים ואף אחד לא מקפיד על **Best Practices**, מהר מאוד תמצאו את עצמכם מול "קוד ספוגטי" (שמעורבב כמו ספוגטי בצלחת) קשה מאוד לתחזק.

אם זכיתם לעבוד בצוות ולא בלבד, Code Review עשוי להיות חלק מה אחירות שלכם, וזה מצוין. העבירו את הקוד שלכם לבדיקה אצל מתכנת אחר והימנוו מהתוצאות. תנו קצת רקע כללי על ההקשר של הקוד ומה ניסיתם להשיג ואפשרו לקוד לדבר בעד עצמו. כן, אם חסר תיעוד – זה יובילו. למי שעבוד בלבד, יש אוריינטנס וקבוצות שבהם אפשר להתייעץ, לשתף קוד ולבקש ביקורת.

את העורות על הקוד מקבלים ונוננים בשתי דרכים. הדרך הראשונה היא בעלפה (פניהם אל פניהם או בטלפון). הדרכן השניה היא בכתב, בצד שינויים 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 ולמוג אותו לתוך הענף הראשי של הפרויקט. מתכנתת שבבודקת קוד תוכל להשאיר עליו הערות, ולאחר שהמפתח יבצע את השינויים היא תוכל לאשר הבקשה כדי למוג את שני הענפים.

כשאתם מקבלים ביקורת, קבלו באהבה הצעות, הערות ושאלות. אל תיקחו את הביקורת כתקיפה אישית נגדכם. זכיתם בעוד זוג עיניים שייעברו על הקוד שלכם, לא משנה למי מכמם יש יותר ניסיון. קחו אותה בשתי ידיים, הקשיבו באמת ועל תרצו למגננה – אפילו אם עבדתם קשה על משה ואולי קצת התאהבתם בפתרון מסוים או נקשרתם אליו, או אם הערה שקיבלתם תדרוש שינוי יחסית גדול והרבה עבודה מצדכם. זכרו, אתם לעבוד ביחד וליצור קוד אICONOTI וברור, לא להראות מי יותר חכם וצדוק.

כשאתם עורכים למשהו אחר Code Review – נסו להבין מה קורה בקוד ולהעירך אם תהיyo מסוגלים להרחב או לשנות אותו במידת הצורך. בוחנו את הקוד אל מול עקרונות מוחים כמו כל אלו שמצוירים בפרק זה. האם השמות הגיוניים וברורים? הפונקציות או הקבצים ארוכים מדי? יש קוד ש חוזר על עצמו והיה אפשר להפוך ליותר DRY? הקוד מסודר ומעומד בצורה עקבית וקריאה? וכן הלאה.

בסוף דבר, בכל יום לומדים משהו חדש. Code Review היא דרך מצוינת ללמידה מהאנשים סביבכם ולהמשיך להתפתח בתור מתכנתים. זה הרgel מעולה שיחד אתכם ממתכנתים זהות קוד פחות טוב ולכתוב קוד יותר טוב. כולם מרוויחים.

Tech Design

כשניגשים למאץ פיתוח רציני, סוף מעשה במחשבה תחילה. כמה שמנסים לקבל באהבה ביקורת, כמעט תמיד מתבאסים לשימוש בסוף התהילה שהוא עדיף להשתמש בספרייה מסויימת שכבר קיימת ולא ידעתם עליה או שאי אפשר לכת על פתרון שסימתם לפתח מşıוקולים שונים שלא הכרתם. לעיתים אפילו תיקעו עם הפתרון הלא-מושלם שלכם כי מאוחר מדי לחזור אחורה ולתקן.

כדי להימנע ממצבים כאלה, ראוי לעשות תכנון טכני, שהוא מסמך בפורמט גמיש. יש אורי התמחות בהםם הגיוני להשתמש בתרשיimi זרים (למשל תכנון של ארכיטקטורת שרת או מסדי נתונים); כשעובדים גם בצד לקוח וגם בצד שרת מקובל לתאר את ה-API שמצפים לבנות לкриאות בינהם – لأن שולחים כל בקשה, אילו פרמטרים מעבירים ואין בדיקת תיראה תגובה טיפוסית.

בעזרת Tech Design טוב אפשר לוודא שמבנים זה את זה ולהתחיל לפתח את הצדדים במקביל, ולא לגנות בסוף שככל אחד תכנן מבנה נתונים שונה לחולוטין ולהתוכח מי צריך להתאים את עצמו לאחר.

את התכנון הזה תעבירו לביקורת עמיתים, דבו עליו וצאו בדרך רק אחרי שהסתמם על הפרטים – וכך תמנעו לפחות חלק מההפתעות גדולות והיקרות בשלבים מאוחרים יותר.

Best Practices ו-אוטומציה

از כתבתם Tech Design וקיבלתם אישור לצאת לדין, עבדתם קשה ואז הגיעו ערמה ענקית של הערות ב-Code Review. איך בכלל מתחילים לעבור על זה? ועם כל הנבוד

וההערכה, איך לא לוקחים את זה קשה כשאלוי אתם מרגישים ש"מחפשים אתכם"
ושכמעט על כל שורה יש למשהו מה להגיד?

עבדתם יפה! לדברים יש נטייה להתבלגן כי ככל בסך הכל אנושיים. היו לכם כוונות טובות, אבל יכול להיות שפה שנחתם להיות עקביים בסגנון, שם העדפתם להעmis יותר מדי על פונקציה אחת, וmdi פעם לא הקפdatum על מהשו מהכללים המומלצים. זה טבעי, אבל כשמכפילים את הנטיה האנושית לעגל פינות ולפספס פרטים במספר אנשי צוות ובכמה חדשני פיתוח, האיניות מידדרת מהר מאוד.

אחד הדברים שהכי עוזרים בRICTOK המעמך של הביקורת, ובכלל בשמיירה על איניות על ידי איתור ומונעת שגיאות מראש, הוא אוטומציה. יש כלים אוטומטיים מעולים, שאציג בהמשך, שעוזרים לשמור על סדר ולהקפיד על הכללים שתקבעו לעצמכם, בלבד או כצוות.

כן, כשהתגינו ל-Review תדעו שבחלק נכבד מההערות פשוט לא תיתקלו, כי כבר קיבלתם פידבק מיידי או תיקון אוטומטי בזמן אמיתי. זה כבר מאחוריכם, וחסכתם גם מהעמימות וגם עצמכם דיוונים על ריווחים, שורות ארוכות מדי ועוד המונן הערות, מהותיות יותר או פחות. יהיה קל, מהיר ונעים יותר להתמקד ולהקשב לביקורת שכן תגיע כשהיא מצומצמת יחסית.

בצotta, מומלץ מאוד להחליט פעם אחת (ሞקדם ככל האפשר) על הקווים המנחים שלכם, להיעזר בכלים האוטומטיים שתלמידו עליהם בחלק הזה וליצור מערכת אוטומטית שתפקיד על הקוד שנכנס לפרויקט. לרובה השמוכה, מתכנתים אחרים כבר בנו מערכות כאלה, ואנו נתנו לפועלות האוטומציה של פיקוח על הקוד שם – Linting.

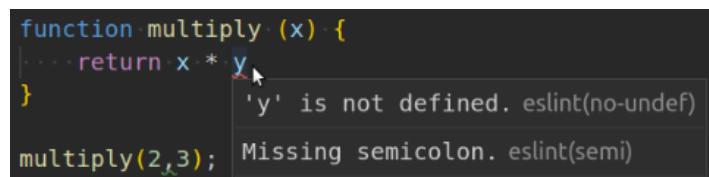
מה זה linting?

המילה `lint` (لينت) באנגלית פירושה מון – הלכלה המctrבר בכיס של בגד או הנדרים הקטנים שנוצרים על גבי הסרגל לאחר כמה כביסות. בעת כתיבת וערכון קוד מתואספים לאת לאט שגיאות או דפוסים לא רצויים אחרים שנוגדים את ה-Best Practices שקבעתם – זהו המון, וכן יש לעבור על הקוד עם כלי שמסיר את המון; בהשאלה, קוראים לו `linter`.

לינטרים יודעים לזהות את השגיאות והדפוסים שהגדרתם, להתריע על הימצאיהם ולפעמים אפילו לתקן אותם אוטומטית.

בעולם התוכנה, הלינטר הוא כלי שתפקידו לבחון את הקוד שכתבתם ולהזהיר משגיאות, באגים או טעויות סגנוניות. הלינטר הראשון פורסם ב-1978, בדק קוד שנכתב בשפת C ונעשה בו שימוש לבדיקת מערכת הפעלה `Unix`.

עורכי טקסט מודרניים המיעודים לכתיבה קוד ג'אווהסקריפט מפעילים בודק שגיאות מובנה שזהירות את המשמש משגיאות שימנעו מהקוד לזרוץ בצורה תקינה בדף. אם כן, למה עדין צריך לינטר? קוד יכול לפעול באופן תקין ועדין לא לעמוד ב-Best Practices ובכללים שקבעתם מסוימות שונות, וכך רצים להריץ על הקוד גם לינטר שיזהיר מחריגות ומטעויות מכל הסוגים.



```
function multiply(x) {
    return x * y;
}
multiply(2,3);
```

`'y' is not defined. eslint(no-undef)`

`Missing semicolon. eslint(semi)`

LINTR יכול להזהיר מפני בעיות כמו שימוש במשתנים שלא הוגדרו, קריאות לפונקציות שאין קיימות, הפרת מוסכמות בנווגע לירוח וסדרו של קוד, שימוש בפייצרים של השפה שנוטים להוביל לביעות אבטחה ועוד. הלינטר יסמן בעיות באמצעות סימן בקו מזג זג אדום לשגיאות וירוק לאזהרות, בדומה לסימון של שגיאות כתיב בمعدדי תמלילים. בעבר עבר מעל הקוד המסומן אפשר יהיה לראות את הסיבה ולעתים גם הצעות לשיפור הקוד.

הכלי הראשון בעולם הג'אווהסקריפט שעשה זאת היה `JS-Lint`, שאוסף סט מוגדר מאוד של כללים בצורה קשורה – מי שלא אהב את הדעות החותכות של היוצר שלו, או שיתת הג'אווהסקריפט דאגלס קרוקפורד, נאלץ להשתמש בכלי אחר בשם `SHHint`, שכן אפשר יותר גמישות.

עם השנים וההתפתחות המהירה של ג'אווהסקריפט, ESLint של ניקולס זאקאס תפס תאוצה, בין היתר בזכות מבנה מאד גמיש. ESLint מאפשר להגדיר ברמה פרטנית את הערכיים הרצויים לכללים קיימים (כמו למשל אורך שורה מקסימלי או איפה בדיק מותר או אסור להשאיר שורות ריקות), אבל גם להגדיר כלליים משלכם, ולהשתמש بكلות generalize שארחים יצרו, כמו שארחיב עוד מעט. נכון לעכשיו, ESLint הוא ה-*Linter* הפופולרי ביותר עבור ג'אווהסקריפט, ולכן מתאים בו.

ESLint

עורכי ג'אווהסקריפט מודרניים יודעים להציג הערות של ESLint בצורה אוטומטית או בהתקנה פשוטה של Tosf. האזהרות והשגיאות שיוצגו נקבעות על פי קובץ הגדרות, בדרך כלל בשם `eslintrc`, בתיקייה הראשית של הפרויקט. אם אצלכם עדין אין קובץ זהה, קראו על `init` ESLint וצרו באמצעותו קובץ הגדרות משלכם – מומלץ להתחיל מאחד הסטים הקיימים שיוצאו לכם במהלך האתחול.

אם בחרתם סט חוקים שונה לכם להשתמש בו אבל יש בו כמה חוקים שלא נראהים לכם, אל תהססו להיכנס לקובץ ההגדרות ולשנות אותם.

בחלק הבא עבורו על כמה Best Practices, חלקם חדשים וחלקם כברזכירתי קודם, לצד הכלל ב-*ESLint* שמסייע לציתם להם.

רשימה של Best Practices של ESLint והחוק הרלוונטי של

בלי מספרי קסם

שם הכלל: no-magic-numbers

"מספרי קסם" הם מספרים שמופיעים בקוד בלי הסבר מפורש, כמו הצבה למשתנה שהשם שלו יכול להאיר עיניים. הצורך בולט בחישובים, וביחוד בחישובים שימושיים כמה מספרים. ברגע הכתיבה ברור איך מגיעים אליהם ויום אוחודש אחר כך כבר מגדירים בראש ולא מבינים מה הסיפור של המספר זהה.

דוגמה לקוד בעיתי עם מספר קסם:

```
setTimeout(ringBuzzer, 180000);
```

למה 180,000 בעצם? בלי הערה שתשפר למה התכוון המשורר, קשה לקלוט במהירות במה מדובר. אפשר להוסיף העירה כמו "three minutes from now", אבל בולדיה לוקח זמן להבין מה אומר המספר הזה ואיך לשנות אותו אם בקשו מכם להאריך את ההמתנה לחמש דקות. מחשבים מהירים מאוד בחישוב, לא צריך ללחוץ עליהם וללעוס עבורים את המספרים מראש. חשוב יותר שהקוד יהיה מובן לכם ולשאר בני האדם שיצטרכו להבין את הקוד, אז מפרקם את המספר ללא מושבר לגורם:

```
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);
```

השווואה קפדיות: `==`

שם הכלל: `eqeqeq`

זה הכלל של לינטרים בג'אוּהַסְקְרִיפֶט, עוד מ-JSLint המקורי. ג'אוּהַסְקְרִיפֶט מאפשרת להשוות בין ערכים מסוימים. זו תכונה מעניינת שמאפשרת גמישות ויכולת להיות שימושית, אבל היא בקלות עלולה לבלב ולהטעות.

כדי להימנע מטעויות קשות לאיתור כתוצאה מההשוואה המתירנית, מקובל ומומלץ להקפיד על שימוש בהשואות הקשות יותר. מומלץ לשמש ב-`==` לבדיקת שוויון `!=` לבדיקת אי שוויון. כך משווים בין ערכים בלי להמיר את סוג המשתנים תוך כדי השוואה.

קוד לא נגיש

שם הכלל: `no-unreachable`

תוֹן כְּדִי `debugging` ו-`refactor`, קורה שנשארת פקודה שיוצאה מקטע הקוד בשלב מוקדם, כמו `return`, `throw` ואחרות. אם מופיע קוד אחרי פקודה כזו, בדרך כלל מדובר בטעות.

בדוגמה הבאה, כל מה שאחרי `return true` לעולם לא ירוז, ESLint תדגיש את הקטע הלא נגיש:

```
function validateEmail (email) {
    return true;
    if (email.length) {
        let result = true;
        // if (!email...) {}
        // TODO: complete validation
    }
    return result;
}
```

חולקה לחלקים קטנים

שמות הכללים: max-lines, max-lines-per-function, max-len

קשה לתפוס הרבה לוגיקה ברכף אחד, וכך שבסכיות ספר או מאמר מומלץ להשתמש בנדיבות בפסקאות, כתורות ועמודים, וכך גם בקוד חשוב לא להתפתות להעmis המון קוד על שורה ארוכה מדי, פונקציה אחת שעשויה הכלול או קובץ אחד של אלפי שורות.

החוקים הבאים הם חוקים שאין עליהם מוסכמו, אך הם יכולים לעזור לכם בכך שימושו תමיר או זהירה שהקוד שאתם כתובים הפך לארוך ומסובבל. אם תבחרו להשתמש בהם כדאי שתתגלו להסכמה עם חברי הצוות שלכם בנוגע לגודלים שמקובלים על כולם.

אורך שורה (מספר תוויים מקסימלי בשורה)

יש להקפיד על אורך שורה שאינו עולה מספר תוויים מסוימים. הסיבה לכך היא שרוצים שורה שנכנסת ברוחב המסך ללא צורך לגולל הצדיה בעת קריאת הקוד. קוד קרייא הוא קוד שאפשר לראות את כלו או את רובו בביטחון. בעבר, כאשר גודל המסך היה קטן יותר, היה מקובל להגיד אורך שורה של 80 תוויים, אבל בימינו, כשהמסכים הרבה יותר גדולים, נהוגים אורכי שורה של 120 ואף 140 תוויים.

אפשר להימנע משורות ארוכות בכמה דרכים. הסיבות העיקריות לשורות ארוכות הן רשיימה ארוכה של תנאים וקוד מקוון. רשיימה ארוכה של תנאים קל לשבור לכמה שורות. קוד מקוון יותר קשה לתיקן, אך אם דואגים לשמור על פונקציות קצרות ופשטות מצליחיםelmaneu זאת ברוב המקרים.

אורך פונקציה (מספר שורות מקסימלי בתחום פונקציה)

אין מוסכמה בנוגע לאורך פונקציה למורות שכולם מסכימים שפונקציות ארוכות זה רע. החוק הזה טוב בעיקר כדי להזכיר לכם שהגוזם, שכן אם תשימושו בו כדאי לבחור מספר שnoch لكم. אם איןכם בטוחיםizia מספר לבחור, התחילה עם מספר השורות שנכנסות לכם במסך ללא גלילה.

עוד דרך לשמור על פונקציות קצרות היא לכתוב פונקציות קצרות שיש להן תפקיד אחד בלבד. רמז לכך שפונקציה עשויה יותר מדי אפשר לקבל כשתנסו לתת לפונקציה שם. אם במהלך הניסיון תגלו בשם הפונקציה את המילה "and", תבינו שניתיתם להכניס יותר מדי

לתוכ פונקציה אחת. ברגע שזיהיתם מקרה כזה, זה הזמן לחלק את הפונקציה לשני חלקים או יותר.

אורק 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 והרגליים ולכנן כדי להגביל את עצמום.

אורק מינימלי ומקסימלי לשמות משתנים

שם הכלל: id-length

שמות משתנים>Kצרים מדי או ארוכים מדי הופכים את הקוד ללא קרייה. שמות קצרים מדי מקשימים להבין מה מטרת המשתנה ושמות ארוכים מדי מסרבלים את הקרייה. כשבוחרים שם למשנה נדרש בקווים המניחים שדנתי בהם תחת הכוורת "בחירה שמות".

בלוי eval

שם הכלל: no-eval

כפי שכבר הוסבר מוקדם יותר בספר, פונקציית eval היא פונקציה מסוכנת מבחינה אבטחת מידע. תמיד אפשר וכדי להימנע משימוש ב-eval באמצעות חלופות שונות לפתרון הבעיה.

בלי משתנים שלא הוצהרו

שם הכלל: **undef-no**

בג'אויסקייפ אפשר להציב ערך לתוך משתנה שלא הוצהר קודם. במקרה זהה המשתנה ייווצר על האובייקט הגלובלי (window, במקרה של ריצה בדף)

```
function greet(name, title) {  
  let firstName = name.split(" ")[0]; //  
  //משתנה שזמין רק בתוך הפונקציה  
  prefix = title || "Mr."; //  
  //זהירות! המשתנה מוגדר על האובייקט הגלובלי  
  console.log(`Hello, ${prefix} ${firstName}!`); // "Hello, ser Jaime!"  
}  
greet("Jaime Lannister", "ser");  
console.log(prefix); // "ser"
```

מלבד זיהום האובייקט הגלובלי, דבר זה גם פותח את הקוד לבאים לא צפויים כתוצאה מחוסר תשומת לב ל-**scope** שפועלים בו. מכיוון שהיא משתמשת בתוקן המשתנה שלא הוצהר מתבצעת כמעט תמיד בטעות, הכלל עוזר לאתר מקרים כאלה ולהימנע מהם.

לסיום, כדי ליצור תוכנות אינטואיטיביות שהקוד שלהם ברור ונוח לתחזקה חשוב להקפיד על **Best Practices**, כמו אלו שסקרטטי ברשימת החלקים כאן. הפעילו שיקול דעת בבחירה ובהתאמת הכללים שלפיהם תעבדו, כדי ליצור לעצמכם תהילה עבודה יעיל וקיים ממש טוב לאורך זמן.

נספח: בדיקות, יציבות וaicות קוד

מחבר: דניאל שטרנלייכט, חברת Outbrain

ברכוט! אם הגיעتم לנספח זהה, נראה אתכם כבר מבינים איך ג'אווהסקריפט עובד ואתם מוכנים להתחיל לבנות אפליקציות ווב. אבלרגע אחד לפני שאתם מתחילהם, רציתי לספר לכם קצת איך לוודא שהקוד שלכם כותבים הוא יציב, איקוני, ויכול לעמוד ב מבחן הזמן כשמפתחים שאיתם תעבדו יעשו בו שינויים בעתיד.

קצת רקע

באוטוביין, חברת המלצות התוכן הגדולה בעולם שמנישה המלצות למליארד משתמשים מדי חודש, עובדים יותר מ-200 מפתחים שכותבים לפני שורות קוד ומשחררים עשרות גרסאות ביום. האתגר הוא לא קטן, שכן כל שינוי בקוד שפתח עושה צורך להיבדק ולהיבחן לפני שהוא עובר ללקוחות.

בקנה מידה כל כך גדול, איך מודאים שהקוד חדש שנכנס לא שובר את החוויה שהלקוחות מצפים לקבל? איך אפשרים למפתחים המשיך לעשות שינויים בקוד בלי ליצור צוואר בקבוק בתהיליך העלאת גרסאות?

התשובה: בדיקות. המון המון בדיקות. למעשה, לפני כל העלאת גרסה באוטוביין רצוט, באופן אוטומטי, לא פחות מאלפי בדיקות!

מבנה בדיקות

כשכתבים בדיקות, אפשר בקלות להגיע למסב שבו יש כל כך הרבה עד שהולכים לאיבוד. בדיק בשביל זה קיימים היום כל הספריות והפרימיטוקים שעוזרים לסדר את הבדיקות. רוב הספריות מסודרות במתודולוגיה בשם **BDD** behavior-driven development או **behavior-driven development** 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 מסויים. תחת הפונקציה הזאת אפשר לקרוא לעוד פונקציה מאותנו סוג כדי לתאר תת-סדרה. הפונקציה **test** היא הבדיקה עצמה, שתכיל "טענות" (או **Assertions** באנגלית), ופונקציית **expect** היא הטענה עצמה. אם הטענה נופלת, הבדיקה נכשלה.

כשמדוברים על בדיקות, יש לא מעט סוגי: Unit Test, Integration Test, End-to-End וכו', ולכל בדיקה המטרות שלה.

בנוסף לזה אتمكن בשלושה סוגי בדיקות שונים:

- 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 הפונקציה הבאה, ש יודעת לקבל מספר ולהחזיר את המספר מוכפל בעצמו:

```
function multiply(number) {
  return number * number;
}
```

כדי לוודא שהפונקציה תקינה ועובדת בבדיקה את מה שהוא נדרש לעשות, מרכיבים כמה בבדיקות שונות:

1. בטור התחלה, מעבירים לפונקציה מספר ובודקים שהוא מחזיר 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 מחרוזת טקסט למשל?

```
multiply('text');
```

כשכתבם את הפונקציה לא חשבתם על האפשרות הזה, וכרגע בעקבות הרצאה הזאת היא תחזיר **NaN**, מה שuvwול לגרום לשגיאות בהמשך. התרחש זה, אגב, עלול להתקיים בעיקר בשפות דינמיות כמו ג'אוּהַסְקְּרִיפְט, שבהן משתנים מסוימים לשנות את סוג בזמן ריצעה.

כדי לתקן את הבעיה, אפשר לשדרוג 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 שמודאת שם הParmeter שעובר הוא לא מספר, מצפים לשגיאה:

```
expect(multiply('test')).toThrowError('Parameter is not a number');
```

בדרך כלל רוצים לעשות בדיקות ייחודית על ייחדות עצמאיות באפליקציה כמו utilities, helpers, services – זאת על מנת לנסות את כל האפשרויות ולהגן עליהם מפני טוויות והתנהוגיות שלא צפיתם להן בשימוש בפונקציה. בדיקות ייחודיה הן דינמיות, וצריך לזכור לשנות אותן אחרי שינויים משמעותיים בקוד.

בדיקות קצה לקצה (End-to-End)

از יש בדיקות ייחודיה שיעודו לrox על חתיכות קטנות באפליקציה, אבל זה שככל אחד מהחלקים עובד בנפרד לא אומר שהם מסונכרנים ועובדים היטב יחד. נוסף על כן, בדיקות ייחודיה לא בודקות תרחישים שבהם מעורבות מערכות אחרות – לצורך העניין בדיקות ייחודיה שרצות על קוד ג'אוהסקריפט בצד הלוקה לא יודעות להגיד אם ממש המשמש עובד כמו שצריך עם הצד השרת.

קחו לדוגמה מפעל לייצור רכב מסווג פורד. יש מחלוקת אחת שאחראית להרכבת המנוע, אחת שאחראית לשולדה ואחת למחשב הרכב. המנוע עובד מציין, השולדה נראהית מעולה, המחשב עובד מציין. ההרכבה של המנוע לשולדה עוברת בשלום, אבל כשבאים לחבר את המחשב, מגלים שהוא מיועד לרכיבים של פורמוללה 1.

בדיק בשביל זה קיימות בדיקות E2E. בעזרת בדיקות E2E אפשר לבדוק שהאפליקציה עובדת מקצה אחד לקצה שני. בדיקות E2E יכולו תרחישים שמשלבים מערכות אחרות. ויוודאו שהבדיקה מצילהה הגיע ממצב א' למצב ב' ללא הפרעות. לשם הדוגמה אקח אפליקציית חיפוש שכולם משתמשים בה מדי יום: גוגל.

הינה התרחיש שרצים שהבדיקה תבדוק:

1. לך לאתר google.com
2. לחץ על אלמנט תיבת חיפוש עם ID בשם "search".

3. הקלד את המילים "אוטוביון" בתיבת החיפוש.
4. לחץ על מקש Enter.
5. בדוק שה-URL הנוכחי הוא google.com/search.
6. בדוק שקיים query param בשם "q" עם הערך "אוטוביון".
7. בדוק שתיבת החיפוש בעלת ID מסווג "search" מכיל את הערך "אוטוביון".

שיםו לב: הבדיקה נעשית בסביבת דפדפן וmdmaja פעילות של משתמש אמיתי באתר אמתי. הרעיון הוא לבדוק אם האפליקציה עובדת ללא התחשבות במבנה שלה או בטכנולוגיה שעומדת מאחוריה, סוג של "קופסה שחורה".

בדיקות ממשקי משתמש (UI Tests)

בדיקות קצה לכמה הן מעולות ויודעות לתת מענה לתרחישים שכולים ממערכות אחרות, אבל לרוב הן בודקות "Happy flows" – מקרים שבהם הכל עובד כמו שצריך. אבל מה לגבי מקרי קצה או מקרים שבהם המערכת שהבדיקות תלויות בהן לא עובדות או לא מחזירות את מה שמשחק המשתמש מצפה לקבל?

הבה נראה נדוגמה את התורחיש שבודק חיפוש בגוגל. התורחיש יוצא מתוך נקודת הנחה שהשרטים שאחראים על החזרת תוצאות החיפוש עובדים ומחזירים תשובה מסויימת. אבל מה יקרה אם תהיה בעיה בשרטטים והם לא יחזירו תשובה? או לחלופין הם יחזירו תוצאות ריקות? האם האפליקציה שבניתם תדע להתמודד עם התרחישים הנ"ל ותיראה כמו שאתם מצפים שהיא תיראה?

אפשר לנכתב סדרה של בדיקות קצה לכמה שעוברות על תרחישים מסווג זהה, אבל כאמור בדיקות קצה לכמה הן יחסית כבדות, והמטרה שלهن היא להריץ בדיקות שיוצאות מתוך נקודת הנחה שהכל עובד.

מפתחי אוטוביון מספרים: נתקלנו בבעיה הזאת ורצינו שתהיה לנו דרך להריץ בדיקות על ממשחק המשתמש ועל איך הוא מגיב למקרי קצה ולתרחישים מורכבים, אז הוספנו עוד שכבה של בדיקות שבאה לנכונות מקרים מסווג זהה. אנחנו קוראים לבדיקות האלה "בדיקות ממשקי משתמש" (UI Tests באנגלית).

כדי להריץ את בדיקות ה-**UI** צריכים סביבה סגורה שלא תלולה בשרתים (או לפחות לא בשרתים אמיתיים), אז בנינו כדי בשם "ליאונרדו" שידוע לזהות בקשות לשרת, להשתלט עליה ו לדמות תשובה שאנו מגדירים מראש.

כך למשל אפשר לראות איך האפליקציה מתמודדת עם תשובות ריקות, עם שגיאות שגיאות מהשרת ועם מצבים שבהם השרת לא מגיב. כמו כן אפשר לבדוק מצבים שבהם התגובה מהשרת איטית.

בדיקות מהסוג זהה הן מהירות, יציבות ויעילות משומשין להן את יכולת לבדוק כל תרחיש שורצים ללא תלות במערכות אחרות.

הכלי ליאונרדו הוא כלי open source וזמין לנולם בGITהאב תחת הכתובת:

<https://github.com/outbrain/Leonardo>

ספריות ופרימורקים מומלצים

יש לא מעט ספריות ופרימורקים לכתיבה בדיקות בג'אווהסקריפט, הינה כמה שאנו, באוטוביון, משתמשים בהם וממליצים לכם בחום לנסות:

- **Jest** – ספרית הבדיקות של פיסבוק. נותנת מענה לרוב סוגי הבדיקות ובעלת API אינטואיטיבי לכתיבה הבדיקות ולהתמודדות עם שגיאות. עובדת מעלה עם ספריות מודרניות כמו Vue, React, Angular, 1.
- **Jasmin** – ספרית בדיקות ותיקת ומאוד פופולרית. טובת מאוד בבדיקות יחידה ועומדת בפני עצמה ללא תלות במערכת כלשהי.
- **Karma** – כלי פשוט שיאפשר לכם להריץ קוד ג'אווהסקריפט ובמקביל לבצע בדיקות בדףנים.

סיכום

למדתם מה אומר המושג "בדיקות" בכתיבה קוד, ואיך הוא יעזר לכם לשמור על יציבות ועל קוד נקי. ראייתם דוגמאות לבדיקות יחידה וביקורת קצה לקצה והבנתם למה הן כל כך חשובות. וגם המלכנו לכם על ספריות ופרימורקים שייעזרו לכם לנתח בבדיקות.

עוד משהו קטן שווה לציון: בקבוצות פיתוח מסוימותלקחו את עניין הבדיקות רחוק יותר ויאמצנו מתודולוגיה בשם "פיתוח מונחה-בדיקות" או באנגלית **Test-Driven Development** (בקיצור TDD). לפי המתודולוגיה זו, בדיקת ייחידה תיכתב עוד לפני כתיבת הקוד שאוטו היא בודקת. קצת על הקצה, אבל אם תחליטו לילכט עם המתודולוגיה הזאת, אתם יכולים להיות בטוחים שהקוד שתכתבו יעבד בצורה חלקה.

נספח: Velo by Wix

מחברת: מור גלעד, חברת Wix

הקדמה

Velo היא פלטפורמת פיתוח אפליקציות שנבנתה על העורך הויזואלי של Wix. העורך הויזואלי של Wix נותן למפתח את היכולת לייצר מסך ויזואלי מפותח מאוד מבלי להשתמש ב-HTML או ב-CSS. הוא מכיל מאות רכיבים חזותיים (elements) שאפשר לעצב אותם באינסוף אפשרויות. בעזרתו אפשר ליצור אתרים מרהיבים.

הפלטפורמה של Velo מאפשרת:

- לכתוב קוד בג'אווהסקריפט עבור הפעולות 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 Velo.

אפשר לראות את האפליקציה שנבנה בכתובת <https://www.wix.com/velo-dev/todo-app>

- הקוד בספר נכתב במטרה למד ולהמchioש בצורה ברורה ופешטה ולאו דווקא לגרום למערכת לעבוד בצורה אידיאלית ויעילה.



The screenshot shows a todo application interface. At the top, there is a search bar with the placeholder "I want to do..." and a button with the number "1". Below the search bar, a header displays "1 item left" (number 3), "40 All Tasks", "Completed" (radio button), "Active" (radio button), and "Clear completed" (button). A circular badge with the number "5" is located next to the "Completed" button. The main area contains two tasks: "Go to the gym" (checked) and "Finish writing the book" (unchecked).

תיאור כללי של אפליקציית המשימות:

1. תיבת טקסט (text input) וכפתור ההוספה המאפשרים למשתמש להוסיף משימה חדשה.
 2. רשימת המשימות – עבור כל משימה רואים את הסטוס שלה, אם היא הושלמה או לא.
 3. מספר הרשימות שנותרו לבצע.
 4. רכיב עם כפturno בחרה (Radio group) המשמשים לסינון המשימות. אפשר לראות את כל המשימות (All Tasks) את המשימות שהושלמו (Completed) או את המשימות שעדיין לא הושלמו (Active).
 5. כפתור למחיקת כל המשימות אשר הושלמו. בלחיצה על הכפתור תופיע למשתמש חלונית שתשאל אותו אם הוא מעוניין לבצע את הפעולה. אם המשתמש יסכים, כל המשימות אשר הושלמו – יימחקו.
- במהלך הפרק השתמשו בהרבה פונקציות Sh-velo מספקת עבור מתכנתים וסביר את כולן. אם ברצונכם להרחיב את הידע שלכם וללמוד פונקציות נוספות Sh-velo reference לבקר באתר המידע של Velo:

<https://www.wix.com/veo/reference/>

 עבור כל פונקציה שאלמד פה, אוסף הפניה לאזור שבו היא מוסברת באתר המידע.

איך מתחילה?

הכניינו עבורכם תבנית מעוצבת מראש שממנה תוכלו להתחיל =>

<https://www.wix.com/website-template/view/html/2186>

העורך הויזואלי של Wix עשיר ביכולותיו ויש הרבה מאוד מה ללמוד עליו. אבל מכיוון yourselves שאותם לומדים כרגע ג'אוהסקרייפט, רציתי לחסוך לכם את זמן העיצוב והבניה של הרכיבים באפליקציה על ידי שימוש בתבנית ולהשאיר לכם רק את החלק המנה – כתיבת הקוד. כמובן שאתם יכולים לשנות את העיצוב ולהוסיף רכיבים חדשים.

Edit this site

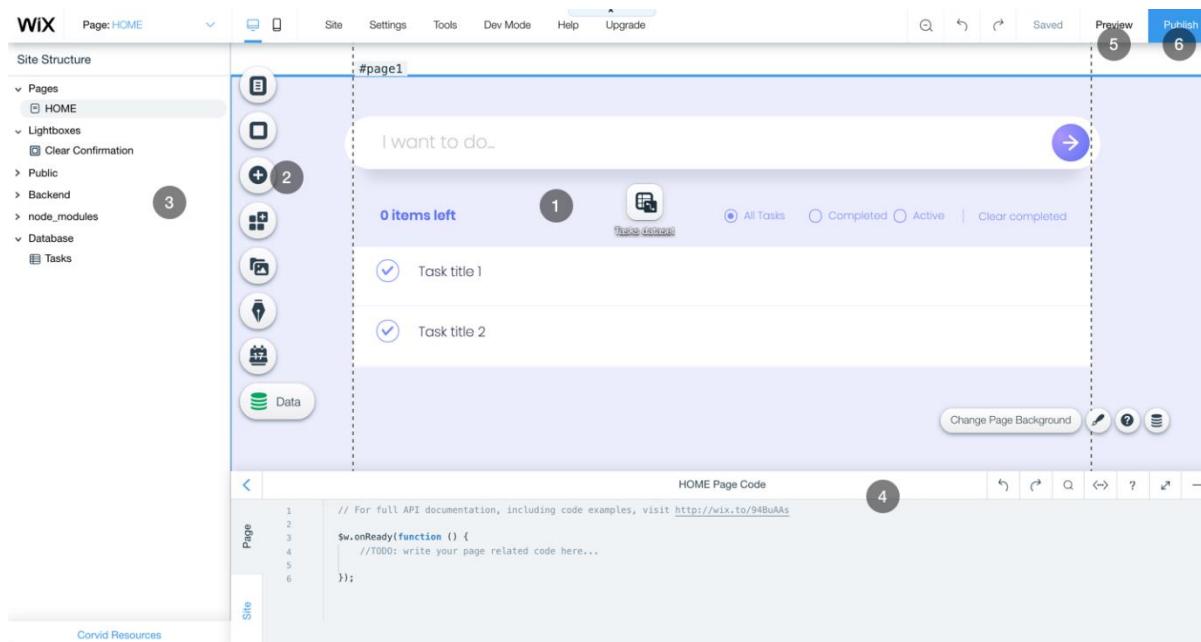
לאחר שנכנסתם לתבנית, לחזו על [הירשמו ל-Wix](#) באמצעות כתובת האימייל שלכם, וכעת אתם נמצאים בעורך הויזואלי של Wix – ברוכים הבאים ☺

כדי להפעיל את Velo, עברו בעזרת העכבר מעל Dev Mode, בתפריט שנמצא בחלק

Turn on Dev Mode

העליון של האתר ואז על

אחריו שהפעילם את Velo, תראו את המסך הבא:



از מה בעצם רואים פה?

1. עורך ויזואלי. האזור שבו עורכים את הצד הוויזואלי של האתר. אל אзор זה מוסיפים רכיבים ומשנים את העיצוב שלהם. אפרט לגבי הרכיבים בהמשך.
2. הכפטור +, להוספה רכיבים חדשים לאתר. תפריט הרכיבים מכיל מאגר של מאות רכיבים שאפשר להוסף לאתר על ידי גירעה.
3. מבנה האתר | **site structure**. פאנל שמציג את כל העמודים, חלונות (lightbox), קובצי הקוד ומסדי הנתונים שיש באתר.
4. בדוגמה הנוכחית יש כרגע עמוד אחד שנקרא **HOME**, חלונית שנקראת "Clear Confirmation" ומסד נתונים המכיל טבלה שנקראת **Tasks**.
5. סביבת הפיתוח. המקום שבו כתבים את הקוד. אפשר לכתוב קוד עבור כל עמוד באפליקציה. עבור כל עמוד חדש באפליקציה, מתחילה עם תבנית קוד מוכנה המכילה **(\$w.onReady)** – שדבר עליה בהמשך.

- * יכול להיות שסבירת הпитוח שלכם תהיה מצומצמת בשורה התחתונה. לחזו על כדי להרחיב אותה.
6. כפתור הצג | **preview**. מציג את האתר שעבדתם עליו עד עכשיו. לרוב, משתמשים בכתופור זהה תוך כדי עבודה על האפליקציה לפני שרצו לפרסם אותה לעולם. במהלך הפרק, אעשה עצרות מדי פעם כדי לראות מה עשיתם עד עכשיו. בעצרות האלו אבקש שתלחזו על **preview**.
 7. כפתור פרסום | **publish**. מפרסם את האתר שבניתם עד עכשיו וחושף אותו לכל העולם.

התבנית שעה אתכם בונים את האפליקציה מגיעה מוכנה עם מסד נתונים, שבו טבלת **Tasks** שמכילה משימות. תלמדו איך להוסיף לטבלה זו משימות חדשות, לשנות את סטטוס המשימה, למחוק משימות ועוד כל מיני פעולות שעוזרות לנו מידע.

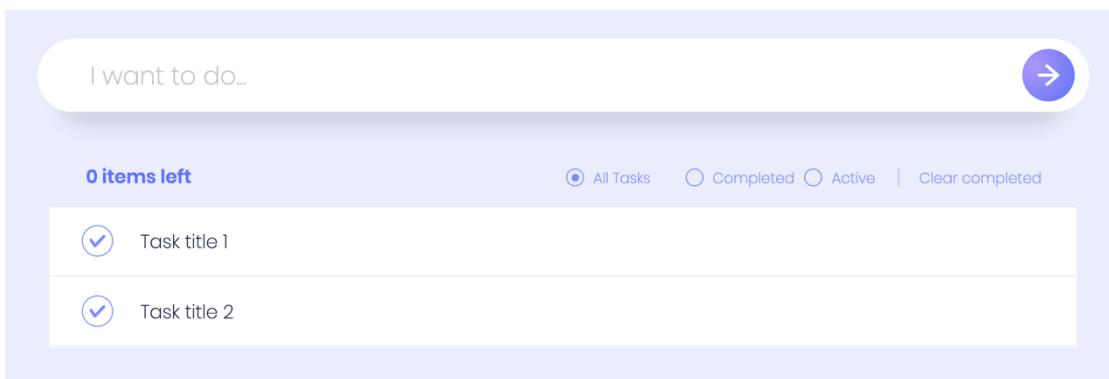
אתם יכולים לראות את המשימות שנמצאות בטבלה על ידי לחיצה על Tasks ב-site **structure**.

כל שורה מייצגת משימה ולכל משימה יש שני מאפיינים:

- Title** -
- Completed** -

אם אתם רוצים ללמוד איך ליצור את הטבלה בעצמכם, קפצו לסוף הפרק, לנוסף שבו הוסיףתי לכם פירוט והרחבת על בניית טבלאות.

לפני שמתחילים לכתוב קוד, לוחצים על **preview** ומבטוננים בנקודת הפתיחה:



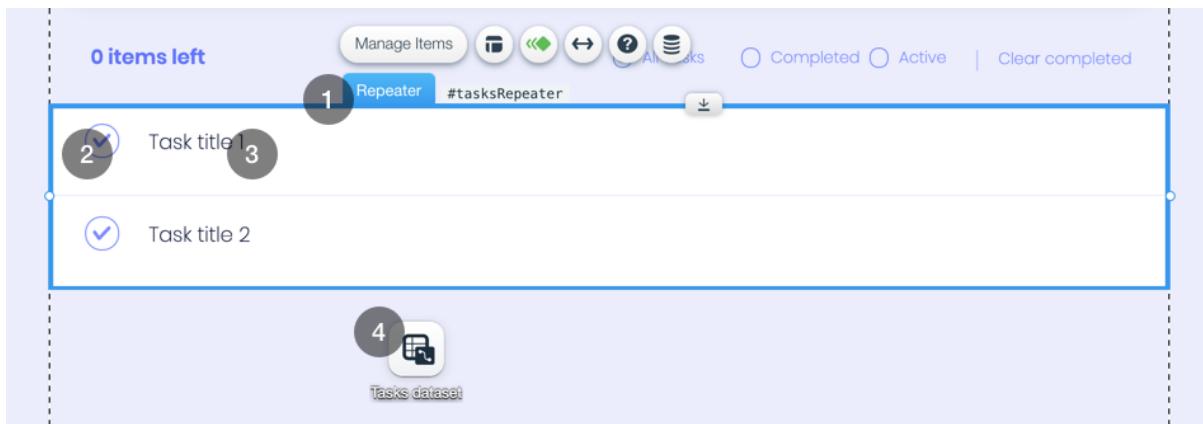
כפי שאפשר לראות, כל הרכיבים מוכנים ומעוצבים מראש אבל אף כפטור לא מ给我 ללחיצה ומספר המשימות שנותרו איננו נכון.

כדי להמשיך לעבוד על האפליקציה. Back to Editor עכשו לוחצים על

הציג נתונים ממסד הנתונים באתר

בחלק זה אלמד איך אפשר להוסיף נתונים מהטבלה על ידי "חיבור" רכיבים מהאפליקציה קציה לטבלת Tasks שלנו. כך חוסכים כתיבת קוד טריוויאלית. מובן שnitן לעשות את זה גם באמצעות קוד – אבל אל דאגה, תכתבו הרבה שורות קוד ממש בקרוב 😊

מתחילה ב�建ת הרכיבים שייקחו חלק בהציג המשימות שבטבלת Tasks של האפליקציה.



1. – רכיב שיוצר בתוכו כמה עותקים של רכיב אחר. כל עותק מכיל עיצוב זהה אך מיידע משתנה. מספר העותקים שייצורו הוא משתנה ונקבע לפי מספר הפריטים שעליו להציג. במקרה זה, מספר המשימות שהוא יציג.

[https://www.wix.com/velo/reference/\\$w.Repeater.html](https://www.wix.com/velo/reference/$w.Repeater.html)

2. | תיבת סימון – רכיב המאפשר כניסה ערך בוליאני (true | false) | Checkbox

[https://www.wix.com/velo/reference/\\$w.Checkbox.html](https://www.wix.com/velo/reference/$w.Checkbox.html)

3. | רכיב טקסטואלי – רכיב שמציג טקסט.

[https://www.wix.com/velo/reference/\\$w.Text.html](https://www.wix.com/velo/reference/$w.Text.html)

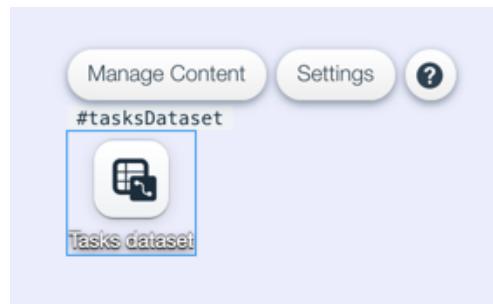
4. – הרכיב שמקשר בין המידע בטבלת Tasks לבין מרכיבים אחרים בעמוד.

<https://www.wix.com/velo/reference/wix-dataset.html>

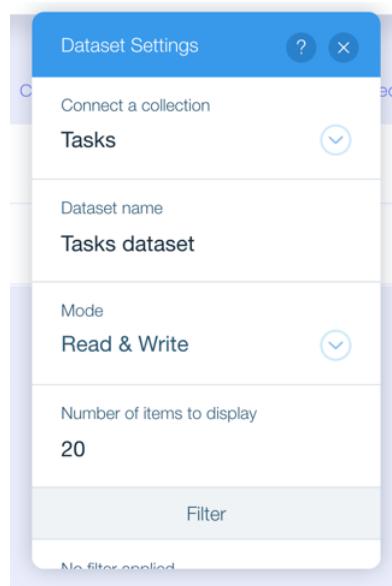
Dataset

ה-dataset מחבר רכיבים בעמוד למידע מטבליות נתוניים. רכיב זה הוא וירטואלי, קלומר, הוא יעשה את תפקידו בניהול מידע מידע בעמוד אבל מבקרי העמוד לא יראו אותו באפליקציה הסופית.

אפשר להסתכל על ההגדרות של ה-dataset בעזרת לחיצה על כפתור settings:



از נראה שה-dataset מחובר לטבלת Tasks ומצוא במצב של קרייה וכתיבה (read & write).



הוא צריך להיות מוגדר במצב של קרייה וכתיבה - **Read & Write**, מכיוון שתרצו אפשרות להציג את המשימות אבל גם לשנות את סטטוס ה-field completed שלן.

יש עוד שני מצבים ל-**dataset**:

- מצב שמאפשר רק קריאה של נתונים מהטבלה **Read Only**
- מצב שמאפשר רק הוספה של נתונים חדשים לטבלה **Write Only**

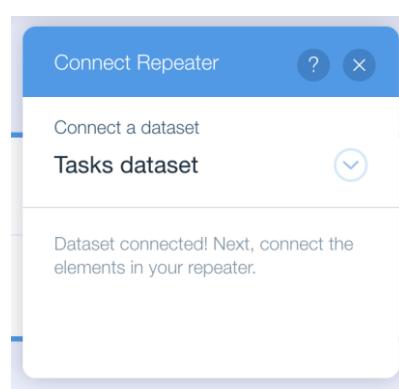
נוסף על כן, **Number of items to display** מראה את כמות הרשומות המקסימלית שה-dataset ישולף מהטבלה. אם אתם רוצים לתרום ביצה של ביותר מ-20 משימות, אתם יכולים לשנות את זה בהגדרות.

חיבור רכיבים למידע מהטבלה

כעת, משתמשים ב-dataset על מנת להציג את המשימות בכל שורה ב-**repeater**. על מנת לעשות זאת לוחצים על האיקון של מסד הנתונים, שמופיע כאשר לוחצים עם העבר על ה-**repeater**.



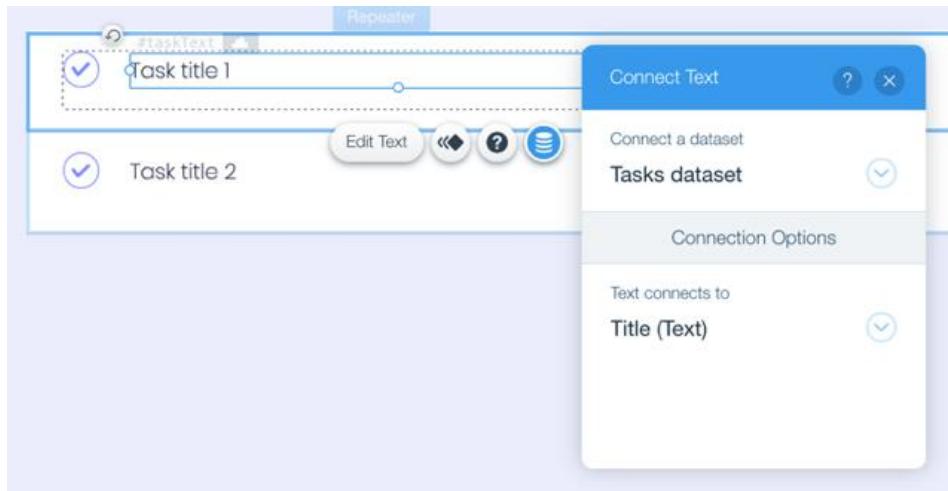
כעת נפתח פאנל החיבור (connection panel) של ה-**repeater**. מכיוון שאתם רוצים לחבר את ה-**repeater** ל-**dataset** שיצרתם, בחרו אותו ב-**dataset**.connect a dataset.



התוצאה של הפעולה זו היא שמספר הקופסאות ב-**repeater** ישוכפלו כמספר הרשומות שה-**dataset** שולף מטבלת המשימות.

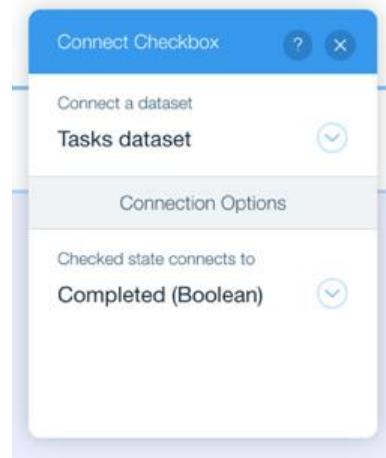
הצעד הבא הוא להציג כל משימה בשני הרכיבים שיש בתוכן ה-**repeater**.

מתחלים ברכיב הטקסט. פותחים את פאנל החיבור של רכיב הטקסט (זה דרוש לחיצה ראשונית על ה-*repeater* ולאחר מכן לחיצה על רכיב הטקסט) ומחברים את הטקסט של הרכיב לשדה **Title** מטבלת **Tasks**.



- אולי שמתם לב שה-*dataset* שיצרתם כבר מחובר. זה קורה באופן אוטומטי מכיוון שה לחברתם את ה-*repeater* ל-*dataset* זהה.
- 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 ידרוש לחיצה על ה-*repeater*, לאחר מכן לחיצה על ה-*group* שמכיל את ה-*checkbox*, ולבסוף על ה-*checkbox*) ומתחברים את הערך **checked** 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? ייש!
עכשו הגיע הזמן להתחיל לכתוב קוד – **חגיגת!**

הוספת משימה חדשה

הרכיבים שיעזרו לכם להוסיף משימה חדשה:



1. **תיבת טקסט | text input** – מאפשרת להכניס את תיאור המשימה.

[https://www.wix.com/velo/reference/\\$w.TextInput.html](https://www.wix.com/velo/reference/$w.TextInput.html)

2. **כפתור | Button** – בלחיצה עליו תמוסך המשימה מתיבת הטקסט לutable Tasks.

[https://www.wix.com/velo/reference/\\$w.Button.html](https://www.wix.com/velo/reference/$w.Button.html)

אם רוצים לאפשר למשתמשים להוסיף משימה חדשה, יש ללמידה איך בלחיצה על הכפתור → מוסיף משימה חדשה לתוך מסד הנתונים ומציגים אותה ב-repeater.

Events

[https://www.wix.com/velo/reference/\\$w.Event.html](https://www.wix.com/velo/reference/$w.Event.html)

בדומה לאירועים בג'אווהסקריפט, גם הרכיבים של Velo מאפשרים הפעלה של אירועים (events).

אלמד אתכם שני אירועים על שני רכיבים שונים:

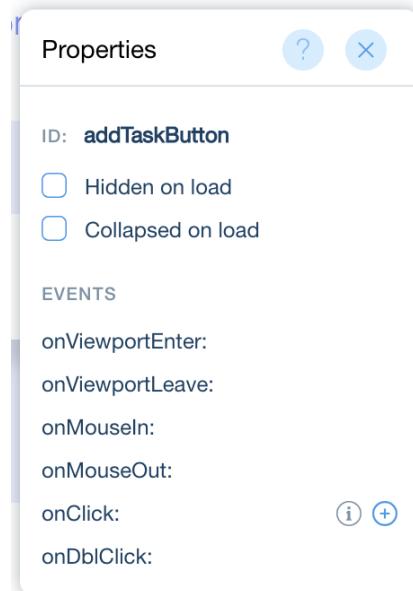
- **button.onClick event** - שנקרא כאשר המבקר באפליקציה לוחץ על כפתור.

[https://www.wix.com/velo/reference/\\$w.Button.html#onClick](https://www.wix.com/velo/reference/$w.Button.html#onClick)

- **textInput.onKeyPress** - שנקרא כאשר המשתמש לוחץ על כפתור במקלדת בזמן שהוא נמצא בתוך רכיב ה-input .text

[https://www.wix.com/velo/reference/\\$w.TextInput.html#onKeyPress](https://www.wix.com/velo/reference/$w.TextInput.html#onKeyPress)

מתחלים בהוספה event לכפתור → . עושם זאת על ידי פתיחת פannel ה-properties .View Properties על של הכפתור, בקлик ימינו על הכפתור ולחיצה על



פאנל ה-`properties` מחולק לשני חלקים:

בחלקו העליון אפשר לראות את ה-ID של הרכיב. בעזרת ה-ID אפשר לבצע פעולות עם הרכיב דרך הקוד. בתבנית שהכנתנו לנו, נתתי מראה לכל רכיב ID שם שימושי, אבל אתם יכולים לשנות את זה.

נוסף על כן, מתחת ל-ID מופיעות הגדרות הנראות של הרכיב בזמן העלייה של העמוד.

בחלקו התיכון של הפאנל נמצאת רשימה של כל ה-`events` הנתמכים עבור הרכיב:

- **onViewportEnter** – מופעל כאשר ה캡טור מופיע בחלון של הדף
- **onViewportLeave** – מופעל כאשר ה캡טור נעלם מחלון הדף
- **onMouseIn** – מופעל כאשר המשתמש מעביר את העכבר מעל ה캡טור
- **onMouseOut** – מופעל כאשר העכבר יוצא מגבולות ה캡טור
- **onClick** – מופעל בלחיצה על ה캡טור
- **onDoubleClick** – מופעל בלחיצה כפולה על ה캡טור

* לכל רכיב יש events שונים בהתאם לפונקציונליות שהוא מספק. ניתן לראות הסבר מדויק על כל ה-`events` של ה캡טור בlienק:

[https://www.wix.com/velo/reference/\\$w.Button.html](https://www.wix.com/velo/reference/$w.Button.html)

במקרה הזה רוצים להוסיף משימה כאשר המשתמש לוחץ על ה캡טור. לכן, עוברים עם העכבר על הצד הימני של event `onClick` ולחיצים על כפתור ה- . לחיצה על ה캡טור

תכנית באופן אוטומטי שם ל-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 הceptor, יוקרא ה-event שלכם. הנה נראה שהוא אכן נקרא בכל לחיצה. מוסיפים כתיבה ל-console:

```
export function addTaskButton_click(event) {
    console.log('button clicked');
}
```

שימוש לב שהפונקציה שלכם מקבלת ארגומנט שנקרו event. הארגומנט הזה הוא אובייקט המכיל מידע על ה-event שנקרו. האובייקט הזה משתנה בהתאם לסוג ה-event.

* מעתה תכתבו ותעתיקו הרבה שורות קוד. אם אתם רוצים שהקוד יעבור הזה באופן אוטומטי, לחזו על הceptor , שנמצא לצד הימני של החלק העליון בסביבת הפיתוח.

עוברים ל-preview, לוחצים על הceptor ובתחתית הדף יפתח ה-console של Velo ויציג את מה שכתבתם בכל פעם שתלחצו. עובד לכם? מעולה! ממשיכים להלאה ושמרים את המשימה.

\$w
[https://www.wix.com/velo/reference//\\$w.html](https://www.wix.com/velo/reference//$w.html)

על מנת לשמר את המשימה שהמשמש כתוב, יש "להוציא" את הטקסט שנכתב מתוך ה-field `text`. את זה עושים בעזרת שימוש בפונקציה **w**.

w היא פונקציה שזמיןה בקוד ומאפשרת לבחור רכיבים מתוך העמוד באמצעות הקוד, בדומה לפונקציה `document.querySelector`. לכל רכיב סט מאפיינים ופונקציות ייחודי המאפשר אינטראקציה עם הרכיב, לקבל ולשנות את המידע שלו, לשנות את מצבו ולהירשם ל-events שהוא חושף.

איך בוחרים רכיב בעזרת `$$`

זכרים את ה-ID שיש לכל רכיב בפאנל ה-properties? אז על מנת לקבל את המשתנה של הרכיב משתמשים ב-ID שלו. ואז, כדי לבחור אותו, קוראים לו `$$` עם ארגומנט מסווג טקסט, המכיל את ה-ID של הרכיב ומתחיל ב-#.

לדוגמה, אם תפתחו את פאנל ה-properties של תיבת הטקסט שלכם, תראו שה-ID שלה הוא `taskInput`, וاز תבחרו אותה כך:

```
$$('taskInput').value
```

אחריו שיש תיבת טקסט, רוצים לדעת מה הטקסט שהמשתמש הכניס לתוכה. לכל רכיב יש מספר רב של נתונים שהוא יכול לספק. במקרה זה צריך取 את ה-value שהמשתמש הכניס לתיבה:

```
export async function addTaskButton_click(event) {
  const taskTitle = $$('taskInput').value;
  console.log('taskTitle:', taskTitle);
}
```

למידע נוסף על משתנה ה-value של תיבת הטקסט:

[https://www.wix.com/velo/reference/\\$w.TextInput.html#value](https://www.wix.com/velo/reference/$w.TextInput.html#value)

עוברים לו `preview`, כתבים את המשימה בשורה המשימה, לוחצים על הכפתור והידד!
רואים את הטקסט שככתבתם.



הצעד הבא => לשמר את המשימה כמשימה חדשה בטלית ה-Tasks.

wix-data

<https://www.wix.com/velo/reference/wix-data.html>

wix-data הוא מודול המאפשר עבודה מול המידע שיש לכם בטבלאות. בעזרתו, אפשר להוסיף רשומות לטבלה, למחוק רשומות וגם לעורך רשומות קיימות.

במהלך הפרק Learned בכל פעם שימוש ופונקציונליות חדשה של `wix-data`.

על מנת להשתמש במודול, יש לייבא אותו מהאזור העליון של קובץ הקוד:

```
import wixData from 'wix-data';
```

הוספה רשומה לטבלה – **wixData.insert()**

<https://www.wix.com/velo/reference/wix-data.html#insert>

מתחילים עם היכולת להכניס משימה חדשה לטבלה. זה אפשרי באמצעות הפונקציה **הסינכרונית insert()** לפונקציה זו מعتبرים שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה – במקרה זה **Tasks**.
2. המשימה שאותה רוצים להוסיף. המשימה תהיה במבנה של אובייקט, שבו המנתנים יהיו זמינים למאפיינים של המשימה בטבלה והערכים שלהם.

از על מנת להוסיף משימה חדשה לטבלה, מייצרים את אובייקט המשימה שיכיל:

– טקסט המתאר את המשימה, שנלקח מתיבת הטקסט **title** –

– משתנה בוליאני המציג את סטטוס המשימה ויאתחל ב-**false** – **completed**

קוראים ל-**wixData.insert()** עם האובייקט שיצרתם. נוסף על כן, משלימים את החוויה בכך שמוסיפים את המשימה החדשה רק במידה שהיא לא ריקה:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }
    await wixData.insert('Tasks', newTask);
  }
}
```

* שימוש לב שמיון שהפונקציה **wixData.insert()** היא אסינכרונית, מוסיפים **async** ו-**event**.

עכשו קופצים שוב ל-preview, מוסיף משימה חדשה, לוחצים על Enter, חזרים לאeditor של Wix ועוברים לעורך התוכן של טבלת Tasks כדי לראות אם המשימה החדשה הותוספה. התווסף? **מעולח!**

| |  Title | Completed |
|---|---|-----------|
| 1 | new task | |
| 2 | Real task 1 | ✓ |
| 3 | Real task 2 | |
| 4 | Real task 3 | |

אבל אני מניח ששפתם לב שחררים כמה דברים כדי שהחויה תהיה שלמה:

1. שורת המשימה לא הטרוקנה אחרי הוספת המשימה.
2. המשימה לא הותוספה לרשימת המשימות שלכם.

ציינתי קודם שבعزורת \$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 שכלל רכיב גגיל באתר ניתן לשינוי, כך גם הדאטא-set.dataset. לכן מבקשים מה-dataset.refresh() לקרוא מחדש את המידע מהטבלה בעזרת קריאה לפונקציה האсинכרונית ()

```
await $w('#dataset1').refresh();
```

למידע נוסף על פונקציית ה-refresh של dataset:

<https://www.wix.com/velo/reference/wix-dataset.Dataset.html#refresh>

זה הקוד המלא:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }
  }
}
```

```

    await wixData.insert('Tasks', newTask);
    $w('#taskInput').value = '';
    await $w('#tasksDataset').refresh();
}
}

```

ושוב, עוברים ל-wview`prev` ומכניסים משימה חדשה. איזה כיף! זה עובד!

תרגיל:

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 – ואת `textInput.onKeyPress` והמשמעות של הארגומנט `event` שמקבלים בפונקציה.

פתרונות:

לאחר פתיחה של פאנל ה-`onKeyPress` event ויצירה של event `properties` יתווסף
נוסף לסביבת הפיתוח. אבל הפעם יהיה צורך לוודא שהכפטור שעליו המשתמש לחץ הוא
כפטור `Enter`, ורק אז לשמר את המשימה ולבצע את כל הפעולות הנדרשות.

```

export async function taskInput_KeyPress(event) {
  if (event.key === 'Enter') {
    // add new task here..
  }
}

```

מכיוון שלא מומלץ לשכפל קוד, מוצאים את פועלות 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 event ה-`taskInput_KeyPress` הפוך לאсинכורוני ונוסף לו `.async`.

עוברים שוב ל-`view` ומכניסים משימות חדשות באמצעות לחיצה על `Enter` וגם בעזרה של לחיצה על כפתור ההוספה.

שינוי סטטוס המשימה

از אפשר לראות את המשימות הנוכחיות ולהוסיף משימות חדשות. אבל המשתמש עדיין לא יכול לעדכן שהמשימה הושלמה.

בכל פעם שהמשתמש משנה את הסימון בכפתור הבחירה (checkbox), שנמצא בתוך `repeater`, נרצה לעדכן את הבחירה החדשה שלו בטבלה עבור המשימה הרלוונטית. מתחילה בהוספת event `onchange` על ה-`checkbox` פאנל ה-`properties`, כמו שprzedתם קודם:

```

export function completedCheckbox_change(event) {
  //Add your code for this event here:
}

```

למידע נוסף על event onChange של כפטור הבחירה:

[https://www.wix.com/velo/reference/\\$w.Checkbox.html#onChange](https://www.wix.com/velo/reference/$w.Checkbox.html#onChange)

* מכיוון שה-x-checkbox נמצא בתוך ה-event repeater, ה-event שונח על checkbox אחד וופעל אוטומטית על כל אחד ממה-checkboxes שיש בכל איבר ב-repeater. ככלומר אין צורך לכתוב event לכל checkbox אם יש לנו repeater.

כדי לשנות את הערך הבוליאני של מאפיין completed של המשימה לערך החדש שהמשתמש סימן ב-x-checkbox, קודם قول צריך להציג את הערך החדש, ועושים זאת על ידי שימוש במשתנה ה-target, שנמצא על ארגומנט ה-event שומuber לפונקציה:

```
export function completedCheckbox_change(event) {
  const completed = event.target.checked;
}
```

למידע נוסף על משתנה ה-target של event:

[https://www.wix.com/velo/reference/\\$w.Event.html#target](https://www.wix.com/velo/reference/$w.Event.html#target)

לאחר מכן נרצה לעדכן את הערך החדש בטבלה.

wixData – שילוף רשומה מהטבלה

<https://www.wix.com/velo/reference/wix-data.html#get>

על מנת לייצר את המשימה שאותה רוצים לעורך, יש לשלווף את הרשומה זו מהטבלה כדי שככל הנתונים הנוכחיים שלה יהיו זמינים.

למודול wixData יש פונקציה אסינכרונית הנקראת (**get**), שמאפשרת שילופה של משימה מהטבלה. לפונקציה זו מעבירים שני ארגומנטים:

1. שם הטבלה שמנה רוצים לשלווף את המשימה – **Tasks**
2. המזהה הייחודי של המשימה – **ID**.

לכל משימה שמכניסים לטבלה מתווסף באופן אוטומטי מזהה שנקרא **id**. המזהה הזה מאפשר לזהות באופן ייחודי את המשימה.

از איך בעצם משיגים את המזהה הייחודי זהה מתוך המשימה ב-`repeater` שלו לוחץ המשמש?

בעזרת ארגומנט ה-`event`. כפי שכתבתי קודם, ארגומנט ה-`event` משתנה בהתאם ל-`event` שנקרה. אבל יש חשיבות למקום הרכיב שבו מתרחש ה-`event` ובמיוחד אם הוא בתוך ה-`repeater`. אם ה-`event` הופעל והרכיב שהוא הופעל ממנו נמצא בתוך ה-`repeater` אז נקבל באובייקט ה-`event` משתנה נוסף, שונה event רגיל, שנקרה context ומכיל את ה-ID של הרשומה שצרכים מהטבלה:

```
event.context.itemId
```

למידע נוסף על המשתנה ה-`context` של `event`:

[https://www.wix.com/velo/reference/\\$w.Event.html#context](https://www.wix.com/velo/reference/$w.Event.html#context)

וכעת, כשייש בידיכם המזהה הייחודי של הרשומה, אפשר לקרוא `get()` ולקבל את הרשומה הנוכחית:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;
  const item = await wixData.get('Tasks', itemId);
}
```

* שוב, יש להפוך את ה-`event` שלכם לאסינכרוני כיון ש-`wixData.get()` היא אסינכרונית.

עדכן רשומה בטבלה – `wixData.update()`

<https://www.wix.com/velo/reference/wix-data.html#update>

יש לכם רשומה, כפי שהיא שמורה בטבלה. עכשו, יש לשנות את ערך ה-`completed` שלה לערך חדש ולעדכן אותו בטבלה.

למודול `wixData` יש פונקציה `update()` שמאפשרת עדכן של רשומה מסוימת. הפונקציה זו מקבלת שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה – `Tasks` –
2. המשימה המעודכנת שרוצים לשמור.

מייצרים משימה מעודכנת ואז קוראים `get()` `wixData.update()` עם העדכן:

```
export async function completedCheckbox_change(event) {
```

```
const completed = event.target.checked;
const itemId = event.context.itemId;

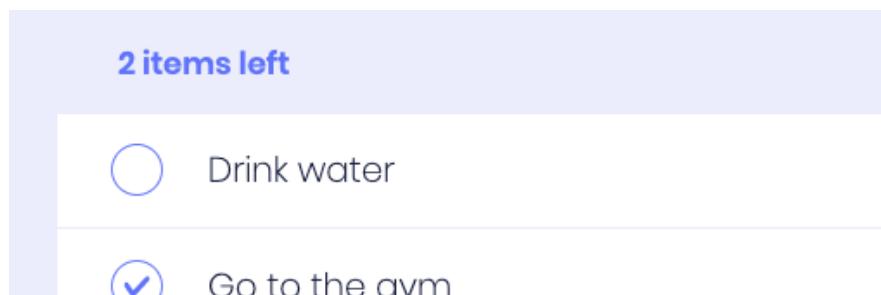
const item = await wixData.get('Tasks', itemId);
const updatedItem = Object.assign({}, item, { completed });

await wixData.update('Tasks', updatedItem);
}
```

הבה נבדוק אם זה עובד ב-wix.preview. מסמנים כמה משימות וחזרים לעורך התוכן לבדוק אם זה באמת עודכן.

נמדד! זה עודכן! הפקנציונליות הבסיסית של האפליקציה עובדת.

מספר המשימות שלא הושלמו



היכולת הבאה שורצים להוסיף לאפליקציה היא תצוגה של מספר המשימות שלא הושלמו (שהערך הבוליאני שלhn במאפיין `completed` הוא false) ברכיב הטקסט שמופיע מעל רשימת המשימות.

מתחלים במבנה פונקציה אסינכרונית חדשה בתחתית קובץ הקוד (מחוץ לפונקציה זה-) (onReady), שתפקידה יהיה לעדכן את רכיב הטקסט במידע הרצוי. שמה: **updateActiveTaskCount**

```
async function updateActiveTaskCount() {
```

הצעד הבא הוא לספר את מספר המשימות שמאפיין `completed` שלhn הוא false. את זה עושים בעזרת פונקציונליות נוספת `wixDataQuery`, ונקראת `wixDataQuery`.

wixDataQuery

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html>

`wixDataQuery` היא שאלתה המאפשרת לבצע בקשה למידע מטבלת המשימות. הבקשת יכולה להכיל שילוב של תנאים שיגדרו את התוצאות שלhn מצפים.

יצירת שאלתה נעשית בשלושה שלבים:

1. אתחול השאלתה בהינתן שם הטבלה שמנה רוצים לקבל את המידע.
2. בניית הפקודות שגדירות את השאלתה.
3. הפעלת השאלתה.

1. אתחול השאלתה **wixData.query**

<https://www.wix.com/velo/reference/wix-data.html#query>

את השאלתה מתחילה בעורთ הפקציה (`wixData.query`), שמקבלת ארגומנט אחד: שם הטבלה שאליה רוצים להוסיף את הרשומה החדש. מתחילים לבנות את השאלתה בעורת:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
}
```

2. בניית השאלתה

לאחר מכון בונים את התנאים של השאלתה באמצעות מגוון פונקציות. במקרה זה יש לקבל רק את המשימות שלא הושלמו. כלומר, כל המשימות שערך ה-`completed` שלהן הוא `false`.

- לכן, משתמשים בפונקציה `eq` (המשמעותה שווה – `equal`), שמקבלת שני ארגומנטים:
 1. שם המאפיין שבאמצעותיו רוצים לסנן את התוצאות – במקרה זה `completed`.
 2. הערך שרוצים שיהיה לתוצאות – במקרה זה `false`.

לחידע נוסף על פונקציית `eq`:

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#eq>

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
}
```

* יש פונקציות רבות ומגוונות שמאפשרות בניה של שאלתה ואפשר לקרוא עליה על בילנק:

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html>

3. הפעלת השאלתה

לביצוע השאלתה אפשר להשתמש בשתי פונקציות אסינכרוניות שונות:

.1. `find()` – החזרת כל האיברים שעוניים על השאלתה.

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#find>

.2. `count()` – החזרת מספר האיברים שעוניים על השאלתה.

<https://www.wix.com/velo/reference/wix-data.WixDataQuery.html#count>

מכיוון שציריך לדעת רק את מספר המשימות שלא הושלמו, נשתמש בפונקציה `:count()`:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
    .count();
}
```

כעת, כשהשגנו מספר המשימות שלא הושלמו, כותבים אותו ברכיב הטקסט.

מתחילה ביצור הטקסט שורצים לראות:

```
let activeTaskText;
switch (activeTaskCount) {
  case 0 :
    activeTaskText = 'Completed all tasks';
    break;
  case 1 :
    activeTaskText = '1 item left';
    break;
  default:
    activeTaskText = `${activeTaskCount} items left`;
    break;
}
```

עכשיו, מציבים את הטקסט החדש בערך ה-`text` של רכיב הטקסט שה-ID שלו הוא `.activeTaskCount`.

זכירים איך עושים את זה?

```
$w('#activeTasksCount').text = activeTaskText;
```

למידע נוסף על משתנה ה-`text` השיין לרכיב הטקסט:

[https://www.wix.com/velo/reference/\\$w.Text.html#text](https://www.wix.com/velo/reference/$w.Text.html#text)

הינה כל הפונקציה:

```

async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
    .count();

  let activeTaskText;
  switch (activeTaskCount) {
    case 0 :
      activeTaskText = 'Completed all tasks';
      break;
    case 1 :
      activeTaskText = '1 item left';
      break;
    default:
      activeTaskText = `${activeTaskCount} items left`;
      break;
  }

  $w('#activeTasksCount').text = activeTaskText;
}

```

יצרתם פונקציה שמעדכנת את מספר 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 – בסוף ה-event completedCheckbox_change

```

export async function completedCheckbox_change(event) {
  // completed update code

  await updateActiveTaskCount();
}

```

3. כאשר העמוד סיים להיטען בעזרה `$w.onReady`

`$w.onReady()`

[https://www.wix.com/velo/reference/\\$w.html#onReady](https://www.wix.com/velo/reference/$w.html#onReady)

זהוי פונקציה שרצה כאשר כל הרכיבים של העמוד סיימו להיטען. בפונקציה זו אפשר לכתוב קוד שירוץ לפני שהמשתמש יתחיל לבצע שינויים באפליקציה.
לכן, קוראים ל-`updateActiveTaskCount` בתוך הפונקציה `onReady` כך:

```
$w.onReady(function () {
  updateActiveTaskCount();
});
```

ובורים ל-`view`.

- מודדים שכמות המשימות הראשונית נכונה
- מוסיפים משימה חדשה ומודדים שמספר המשימות שלא בוצעו עולה ב-1
- מסמנים משימה שלא בוצעה ומודדים שמספר המשימות שלא בוצעו יורד ב-1
- מורידים סימן למשימה שבוצעה ומודדים שמספר המשימות שלא בוצעו עולה ב-1

עובד? שיגעון! הלה!

סינון המשימות לפי סטטוס המשימה

עכשו נסנן את המשימות המוצגות לפי הסינון שהמשתמש יבחר בcptori הרדיו (radio button group) שנמצאים מעל רשימת המשימות:



למידע נוסף על רכיב cptori הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html)

להלן הנתנויות הרצויות עבור כל אחד מהcptoriים:

- **All Tasks** – כל המשימות יופיעו ללא סינון

- רק המשימות שהושלמו (שהמשתנה completed שלהם שווה ל- **Completed**) יופיעו (true)
- המשימות שעדיין לא הושלמו (שהמשתנה completed שלהם שווה ל- **Active**) יופיעו (false)

מכיוון שהסינון יבוצע לאחר שהמשתמש ילחץ על אחד הcptורים, ה-event המתאים הוא radioGroup.onChange – מוסיפים אותו כמפורט קודם:

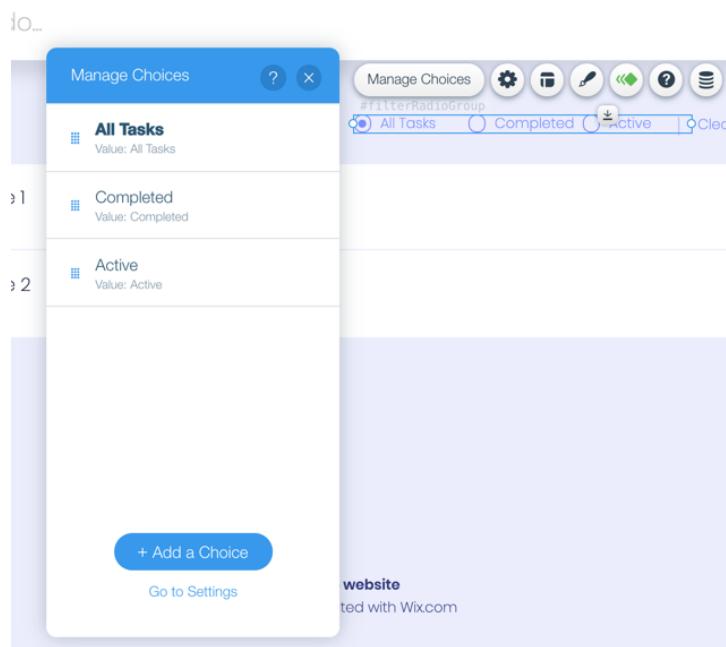
```
export function filterRadioGroup_change(event) {
  //Add your code for this event here:
}
```

لمידע נוסף על event onChange ה- event של cptורי הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html#onChange](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html#onChange)

בשלב הבא מבינים מהו סוג הסינון שבחר המשתמש ומממשים אותו.

עבור כל אחת מהאפשרויות ב-radio group מציגים 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 שהוא חלק מכל cptור רדיו, וכשמו כן הוא – הנתון (או הערך) שאוטו אנו רוצים לקבל מהמשתמש.

```
export function filterRadioGroup_change(event) {
  const filterValue = $w('#filterRadioGroup').value;
}
```

למידע נוסף על משתנה ה-value של כפתורי הרדיו:

[https://www.wix.com/velo/reference/\\$w.RadioButtonGroup.html#value](https://www.wix.com/velo/reference/$w.RadioButtonGroup.html#value)

עכשו מבצעים סינון של המידע שהdataset מביא אל העמוד בהתאם לבחירה של המשתמש.

wixDataFilter

<https://www.wix.com/velo/reference/wix-data.WixDataFilter.html>

wixDataFilter מאפשר לנו לסנן את המידע שמניע אליו מה-dataset לפי תנאים מוגדרים.

סינון מידע מה-dataset נעשה בשלושה שלבים:

1. אתחול המסנן () .wixData.filter()
2. בניית הפקודות שמנגידו את הסינון.
3. הפעלת הסינון על ידי העברת של אובייקט הסינון שייצרתם ל-dataset.

wixData.filter() .1

<https://www.wix.com/velo/reference/wix-data.html#filter>

מייצר אובייקט סינון, שמועבר לדataset, ובכך מאפשר לסנן את המידע שהdataset מביא לעמוד:

```
let filter = wixData.filter();
```

2. בניית הסינון

לאחר מכון בונים את הסינון על ידי שימוש באותם התנאים שמנגידו שאליתה.

במקרה הזה צריכים ליצור שלושה סוגי שונים של סינון:

כאשר המשתמש לוחץ על Completed, רוצים לסנן את כל המשימות שלא הושלמו ולהציג רק את המשימות שהושלמו. בעצם רוצים להשאיר את כל המשימות שערך ה-

שלהן שווה ל-true. לצורך זה משתמשים ב-`eq` (בדומה לשאלתה שיצרתם קודם):

```
let filter = wixData.filter().eq('completed', true);
```

כאשר המשתמש לוחץ על Active, רוצים לסנן את המשימות שהושלמו ולהציג את המשימות שלא הושלמו. פה, רוצים להשאיר את כל המשימות שערך `completed` שלהן 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 - הפעלת הסינון על ה-`dataset.setFilter()` .3

<https://www.wix.com/velo/reference/wix-dataset.Dataset.html#setFilter>

`dataset.setFilter()` הוא פונקציה אסינכרונית שמבצעת את הסינון על רכיב ה-`dataset` היא מקבלת ארגומנט אחד – אובייקט ה필טר שיצרתם עד עכשיו.

לאחר שקוראים לפונקציה זו, ה-`dataset` יבצע קריאה מחודשת למשימות מהטבלה, אבל ללא משימות שאינן עוננות על קритריון הסינון.

```
await $w('#tasksDataset').setFilter(filter);
```

עכשו מחבריהם הכל ייחד ל-`event` של ה-`onchange` שיצרתם קודם:

```

export async function filterRadioGroup_change(event) {
  const filterValue = $w('#filterRadioGroup').value;

  let filter;
  switch (filterValue) {
    case 'Completed':
      filter = wixData.filter()
        .eq('completed', true);
      break;
    case 'Active':
      filter = wixData.filter()
        .eq('completed', false);
      break;
    case 'All Tasks':
      filter = wixData.filter();
      break;
  }

  await $w('#tasksDataset').setFilter(filter);
}

```

לא מספיק לקרוא לפונקציית הסינון רק כאשר המשתמש לוחץ על אחד מכפתורי הסינון. יש לשנן את המשימות בעוד כמה מצבים בקוד. לכן נתחיל בהעברת קוד הסינון לפונקציה `filterTasks` ששם היא:

```

async function filterTasks() {
  const filterValue = $w('#filterRadioGroup').value;

  // create filter code

  await $w('#tasksDataset').setFilter(filter);
}

```

עכשו, קוראים לפונקציה מכל המקומות השונים שדורשים אף הם סינון:

1. כאשר המשתמש לוחץ על אחד המכפתורים מה-buttons radio group (כמו שעשיתם):

```

export async function filterRadioGroup_change(event) {
  await filterTasks();
}

```

2. כאשר המשתמש משנה את סטטוס completed-of-the checkbox של אחת המשימות:

```

export async function completedCheckbox_change(event) {
}

```

```
// change completed status
// update uncompleted counter
await filterTasks();
}
```

3. כאשר המשתמש מוסיף משימה חדשה:

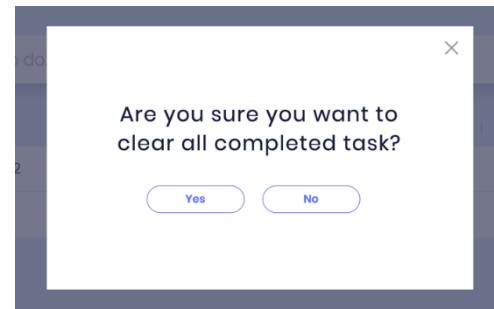
```
async function addNewTask() {
  const taskTitle = $w('#taskInput').value;
  if (taskTitle.length !== 0) {
    // add new task
    await updateActiveTaskCount();
    await filterTasks();
  }
}
```

עבורוים ל-w-view, "משחקים" עם הסינון הזה עתה נכתב, משנים סטטוס משימות כאשר נמצאים על סינון מסוים ומודאים שהן נשאות או נעלמות ולבסוף, מוסיפים משימה חדשה כאשר נמצאים במצב סינון של משימות שלא הושלמו ומודאים שהמשימה לא מופיעה בראשימה.

שמחה וSSHONI יש סינון! ☺

ניקוי המשימות שהושלמו

עכשו ממסים את כפתור ה-Clear completed, שבלחיצה יפתח את חלון האישור:



- אם המשתמש ילחץ Yes => סגורים את חלון האישור ומוחקים את כל המשימות שהושלמו
- אם המשתמש ילחץ No => סגורים את חלון האישור ולא עושים דבר

את חלון האישור כבר הכתינו בעברכם וקרأتي לו Clear Confirmation. תמצאו אותו ב- Lightboxes Site Structure.

כמו שלמדתם (דרך פאנל ה-  Clear completed) מוסיפים event של לחיצה על כפטור ה- :properties

```
export function clearCompletedButton_click(event) {  
    // Add your code for this event here:  
}
```

ועכשיו אלמד איך פותחים את חלון האישור.

wix-window

<https://www.wix.com/velo/reference/wix-window.html>

wix-window הוא מודול שמאפשר עבודה מול חלון הדף שנעליו עובדים. על מנת להשתמש במודול, יש ליבא אותו מהאזור העליון של קובץ הקוד (בדומה ליבוא של **wix-data** :)

```
import wixWindow from 'wix-window';
```

wixWindow.openLightBox()

<https://www.wix.com/velo/reference/wix-window.html#openLightbox>

המודול wixWindow תומך בפתיחת של חלון חדש על ידי קריאה לפונקציה האсинכורונית openLightBox, שמקבלת שני ארגומנטים:

1. שם החלון שאוטו רוצים לפתוח. במקרה זה – 'Clear Confirmation'
2. האובייקט שרוצים להעביר לחלון. במקרה זה אין צורך בכך.

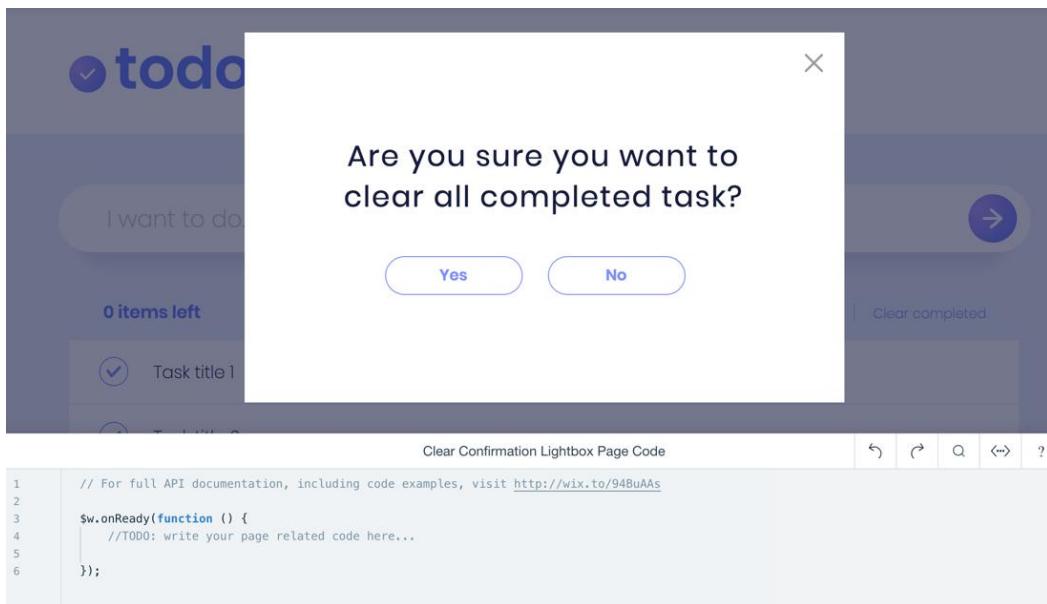
ערך ההחזרה של ה-promise יהיה האובייקט שיוחזר על ידי החלון.
לכן, על מנת לפתח את החלון כותבים את הקוד:

```
export async function clearCompletedButton_click(event) {
  const shouldClearCompleted = await wixWindow.openLightbox('Clear
Confirmation')
}
```

ועוברים ל-preview כדי לוודא שהחלון אכן נפתח.

כעת רוצים לדעת אם המשמש לחץ על Yes או על No, כדי לדעת איך לפעול. ככלומר,
רוצים שההתשובה שתחזר מה-promise כשהחלון יסגור תהיה ערך בוליאני, שאם הוא
חוובו – ימחקו המשימות שהושלמו ואם הוא שלילו – לא יקרה דבר.

עוברים לחלון ה-`event` כדי להחזיר תשובה ל-`event`.



גם פה, כמו בעמוד ה-HOME שלכם, אפשר לכתוב קוד בעורך הקוד שבתחתית המסך.
ועכשיו, מוסיפים event ללחיצה על כפתור ה-**Yes**:

```
export function approveBtn_click(event) {
    // Add your code for this event here:
}
```

ומוסיפים event ללחיצה על כפתור ה-**No**:

```
export function rejectBtn_click(event) {
    // Add your code for this event here:
}
```

כעת רוצים לסגור את החלון בלחיצה על כל אחד מהכפתורים. מתחילהם ביבוא המודול **wix-window** לחלק העליון של הקוד:

```
import wixWindow from 'wix-window';
```

wixWindow.lightbox.close()

<https://www.wix.com/velo/reference/wix-window/lightbox-obj/close>

ב**wix** היה פונקציה שסגרת את החלון שבו נמצאים ומחזירה את המשמש לעמוד שפתח את החלון. הפונקציה זו מקבלת ארגומנט אחד: ערך שינון **wixWindow.openLightbox()**.
התשובה של ה-**promise** שחרר מ-

לכן, בלחיצה על **Yes** מוחזרים **true** ובלחיצה על **No** מוחזרים **false**:

```

export function approveBtn_click(event) {
  wixWindow.lightbox.close(true);
}

export function rejectBtn_click(event) {
  wixWindow.lightbox.close(false);
}

```

אש! חוזרים לעמוד ה-HOME לפונקציה onClick שהתחלנו לכתוב מוקדם, כדי לבצע את המחיקה במידה שהמשתמש ילחץ על Yes. במידה שהמשתמש בחר למחוק את המשימות שהושלמו, מבצעים את המחיקה בשני שלבים:

1. שאלתה שתחזיר את כל המשימות שערך ה-completed שלhn הוא true.
2. מחיקה של כל המשימות שהתקבלו מהשאלתה.

למヂתם מוקדם יותר אין לבצע שאלתה דומה, אבל בשאלתה הקודמת רציתם לקבל את מספר המשימות ולכн השותמשם ב-.count(). הפעם תשתמשו 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/velo/reference/wix-data.WixDataQuery.html#find>

כשקוראים ל-.find(), חזר אובייקט שמכיל כל מיני דברים. אחד מהם הוא ה-items שמחזיק מערך של כל הערכים שחזרו – במקרה זה, מערך של כל המשימות שחזרו מהשאלתה.

כעת עוברים על כל אחת מהמשימות הללו ומוחקים אותן באמצעות:

מחיקת רשומה מהטבלה – **wixData.bulkRemove()**

<https://www.wix.com/velo/reference/wix-data.html#bulkRemove>

מחיקת רשומות מהטבלה נעשית על ידי שימוש בפונקציה האסינכרונית **bulkRemove()**, שמקבלת שני ארגומנטים:

1. שם הטבלה שמננה רוצים למחוק את הרשומות – במקרה זהה `.Tasks`
2. מערך המכיל את ה-`ids` של הרשומות שאוזם רוצים למחוק.

זכירים את מזהה `-id` שנכנס באופן אוטומטי לכל משימה בטבלה? מעולה!
از עכשו תשתמשו בו על מנת ליצור מערך של ה-`ids` שאנו רוצים למחוק.

```
const queryResult = await wixData.query('Tasks')  
  .eq('completed', true)  
  .find();  
const completedTasks = queryResult.items;  
const completedTasksIds = completedTasks.map(task => task._id);
```

את המערך שיצרנו נעביר לפונקציה האסינכרונית `bulkRemove`

```
await wixData.bulkRemove('Tasks', completedTasksIds);
```

ולקינוח, קוראים שוב לפונקציה `filterTasks()`, כדי להסיר מהתצוגה את המשימות שנמחקו.

ולקood המלא:

```
export async function clearCompletedButton_click(event) {  
  const shouldClearCompleted = await wixWindow.openLightbox('Clear  
Confirmation');  
  
  if (shouldClearCompleted) {  
    const queryResult = await wixData.query('Tasks')  
      .eq('completed', true)  
      .find();  
    const completedTasks = queryResult.items;  
    const completedTasksIds = completedTasks.map(task => task._id);  
  
    await wixData.bulkRemove('Tasks', completedTasksIds);  
    await filterTasks();  
  }  
}
```

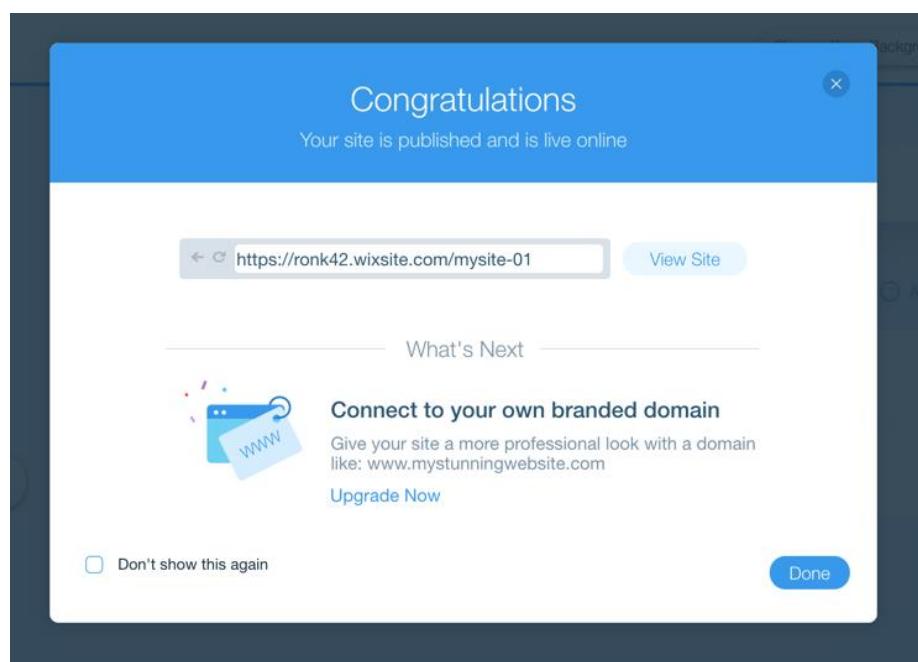
עכשו ל-`preview`. צריך לוודא שיש כמה משימות שהושלמו וכמה משימות שלא הושלמו ולבצע שתי בדיקות שונות:

1. לחיצה על Clear completed ולאחר מכן לחיצה על No => יש לצפות לכך ששום דבר לא יקרה.
2. לחיצה על Clear completed ולאחר מכן לחיצה על Yes => יש לצפות שנכל המשימות שהושלמו יימחקו.

ממש כמו שסויימתם!
זכירים את נפטור-h**publish** מתחילת הפרק?
עכשו הזמן ללחוץ עליו!

אחרי לחיצה על נפטור-h**publish** תהיה האפליקציה שלכם זמינה כאתר לכל דבר וכל העולם!

הכתובת של האפליקציה שלכם תוצג בחלון שייפתח לאחר שתלחצו על **publish**:



* אחרי שפרסמתם את האפליקציה, המשימות שהוספتم בזמן שעבדתם על האפליקציה לא יופיעו ב-preview, מכיוון שאתם עובדים על מסדי נתונים שונים עבור פיתוח ועבור האפליקציה האמיתית.

תוספה – ייצרת מסד נתונים

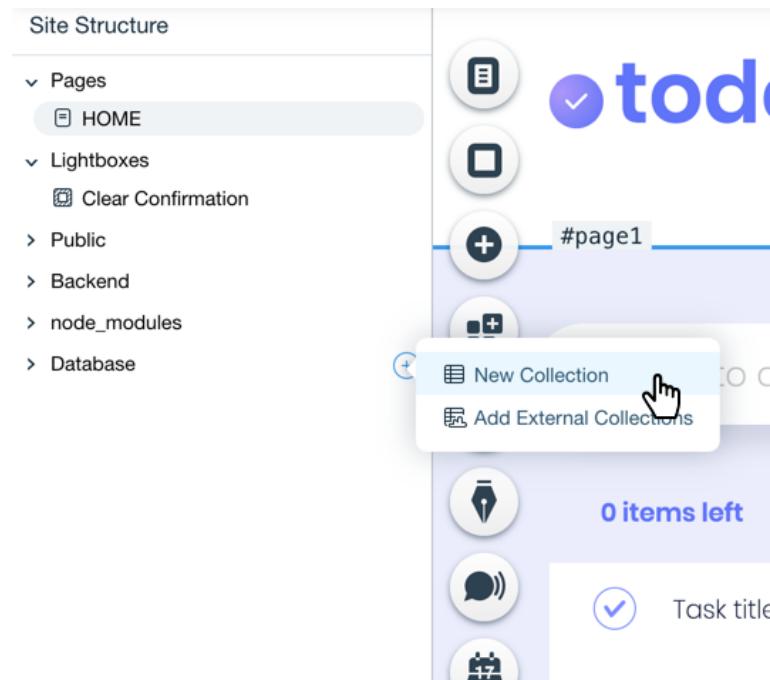
מסד נתונים הוא אמצעי המשמש לאחסון מסודר של נתונים. במקרה של Velo, הנתונים הללו מאוחסנים במודול של טבלאות (collections) ובכל טבלה העמודות מייצגות שדות (fields) והשורות מייצגות רשומות שונות.

נהוג לשיק כל טבלה לישות מסוימת באפליקציה, ואז כל שורה מייצגת מקרה ספציפי של הישות.

נוסף על כן, כל עמודה בטבלה היא בעצם שדה שמייצג מאפיין מסוים של הישות. לכל שדה יש סוג כגון טקסט, מספר, בוליани, תמונה וכדומה. הנהו נתנו את מסד הנתונים של האפליקציה.

באפליקציה שלכם יש ישות שנקראת משימה (task), ולכן יש ליצור טבלה שנקראת Tasks. כל שורה בטבלה תציג משימה בודדת. על מנת ליצור טבלה חדשה עוברים עם העכבר על כפתור +, שיופיע מעבר על צידו הימני של ה-`site structure` שנמצא בפאנל ה-`database`.

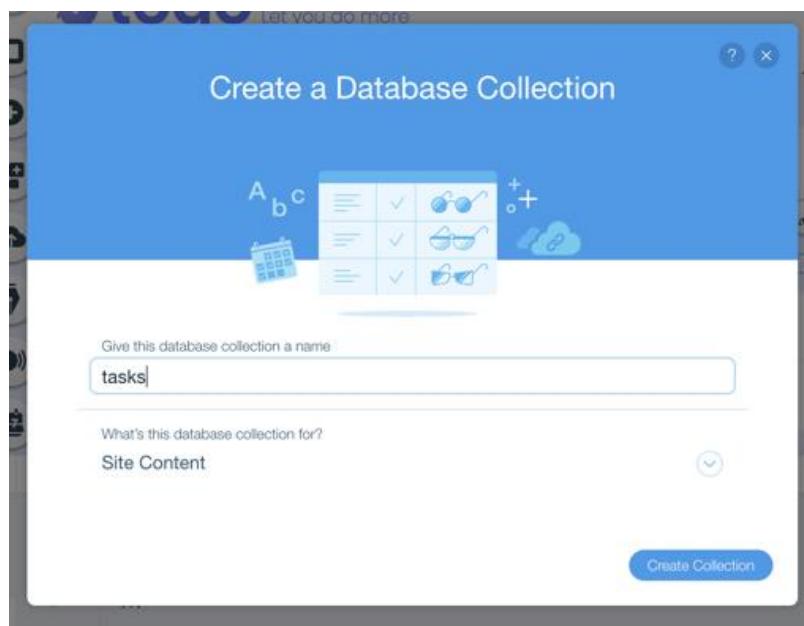
לאחר מכן לוחצים על ה-`New Collection`:



הличיצה תקפיים מסך שבו נתונים לטבלה את השם tasks ולאחר לחיצה על

תיווצר טבלה חדשה.

Create Collection

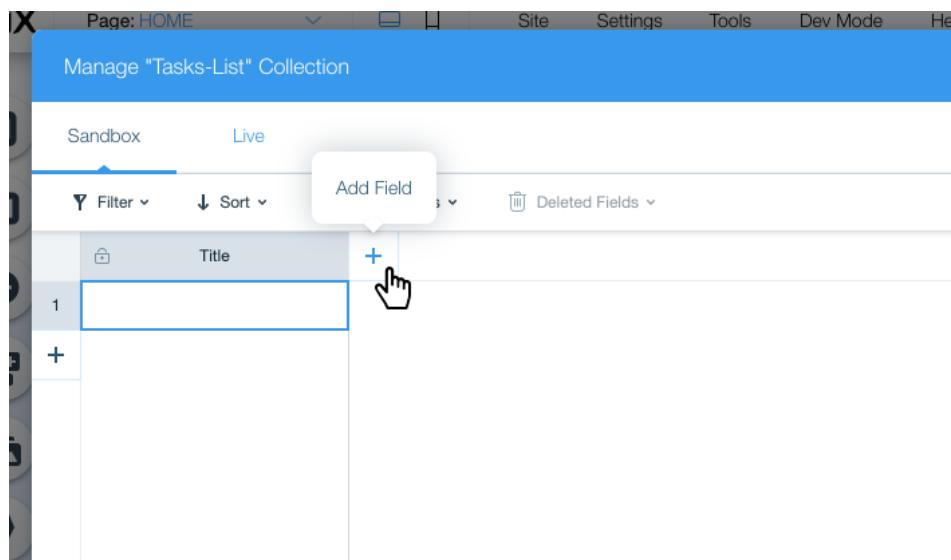


יצירת הטבלה הוסיפה טבלה חדשה site structure database מתחת ל-structure, ובנוסף העבירה אתכם ישיר לאזר שנקרא "עורק התוכן" – content manager, שבו אפשר להוסיף שדות ולערוך את התוכן של הטבלה.

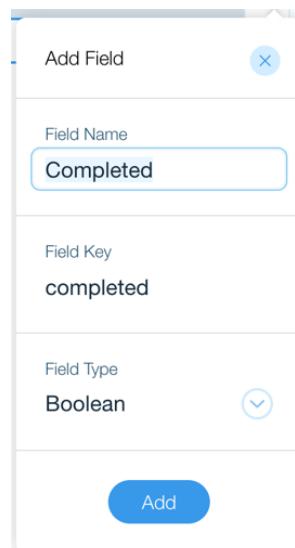
הגיע הזמן לייצר את השדות. כל משימה מכילה את תיאור המשימה בטקסט וערך בוליאני שמנדר את סטטוס המשימה – אם היא הושלמה או לא. לכן, יש צורך בשני שדות:

- שדה **Title** מסוג טקסט (text) – שמיוצר באופן אוטומטי ברגע שמייצרים טבלה חדשה ולכון לא צריכים להוסיף אותו ידנית.
- שדה **Completed** מסוג בוליאן (boolean).

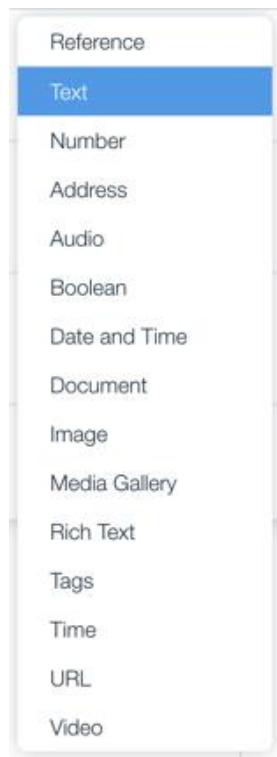
על מנת להוסיף שדה יש ללחוץ על כפתור הפלוס ואז יופתח פאנל ההגדרות של השדה.



בפאנל ההנדרות יש שלושה קלטים שצורך מלא כאשר מיצרים שדה חדש:



- . **Field Name** - שם השדה שרוצים. במקרה זה – **Completed**
- . **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 ליצר. אפשר לבחור מבחר גדול של סוגיים.



כעת קוראים לשדה הנוסף **Completed** ומגדירים אותו מסוג **Boolean**.

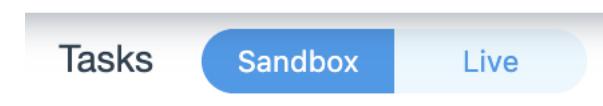
לבסוף לוחצים על **Add** והשדה יתווסף.

כעת יש טבלה בעלת שתי עמודות. אפשר להוסיף רשומות באופן ידני, לשנות אותן וafilו למחוק אותן מהתוך עורך התוכן.

למדתם את רוב המידע שייהי רלוונטי לכם עבור מסדי נתונים ב-**Velox**. עכשיו תלמדו על העבודה עם מסדי הנתונים לאחר שפורסםם את האתר.

Sandbox | Live

אם תשים לב, בחלק העליון של עורך התוכן יש שני כפתורים:



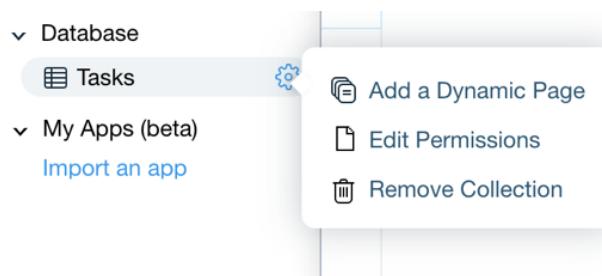
- בליציה על כפתור **Sandbox** יוצגו הרשומות שיוופיעו בזמן עריכה האפליקציה. כמובן, הרשומות שרואים כאשר לוחצים על **preview**.
- בליציה על כפתור **Live** יוצגו הרשומות שרואים באפליקציה האמיתית אחרי **publish**.

Permissions

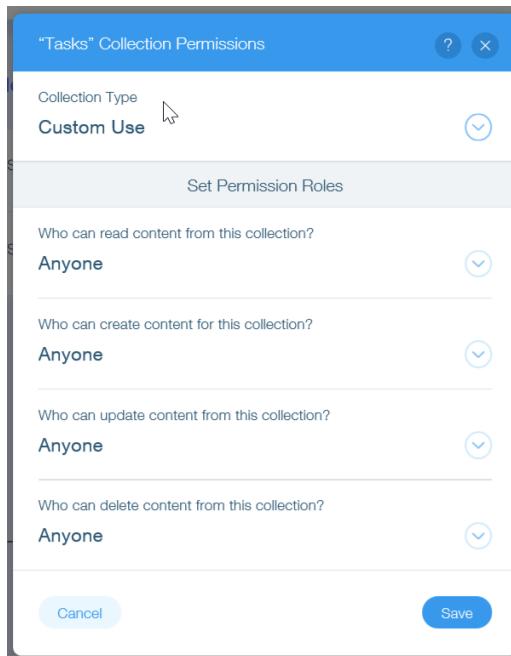
לכל טבלה יש הרשאות שגדירות מי יכול לראות את הנתונים שבטבלה, מי יכול ליצור אותן, לעורוך אותן ולמחוק אותן.
הרשאות הראשוניות שמקבלים כשיצרים טבלה חדשה הן מאוד שמרניות. כלומר, הן לא יתנו יותר מדי יכולות למשתמשים חיצוניים. לכן, הפעולות שמבצעים על הטבלה באפליקציה האמיתית (לא ב-preview) יחזירו שגיאה:

| | |
|----------------|---|
| 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 לעורוך את הגדרות של הרשאות. עבור אפליקציית המשימות שלכם, תרצו שההרשאות ייראו כך:
* בתבנית שיצרתי עבורכם כבר הגדרתי את הרשאות של הטבלה בצורה זו.



סיכום

סיימתם לכתוב את האפליקציה שלכם. מה למדתם?

- חיבור מידע מטבליות הנתונים לאפליקציה
- יצרת אינטראקציות עם רכיבים של העורך הווייזואלי דרך הקוד
- צפיה, הוספה, עריכה ומחיקה של רשומות בטבלאות באמצעות `wix-data`
- יצרת שאלות וסינונים על הרשומות מהטבלה
- פתיחה חלונית וסגירתה באמצעות קוד
- מי שקרא את הנספה – איך ליצור טבלאות חדשות

עכשו הגיע הזמן שלכם ליצור אפליקציות חדשות, לומוד על עוד יכולות `sho-velo` מספקת לכם ולהתנסות בהן.

נספח: ג'אווהסקרייפט מונחה עצמים

מחברים: יהודה גלעד, חברת Chegg, רן בר-זיק

מה זה תכנות מונחה עצמים?

תכנות מונחה עצמים הוא שיטה לנכתב קוד מורכב. עד כה בספר התנסיתם בעיקר בلمידת השפה וכתיבת קוד פשוט יחסית. פונקציה אחת, פקודה אחת או שתיים ומערכות מידע לא מורכבים. אבל בעולם האמיתי אנחנו חייבים לחלק את הקוד לחלקים או ל"חטיות" בדיקון כמו פונקציות או אובייקטים שלמדנו כאן. חלוקת הקוד היא הכרחית כאשר אנו מפתחים מערכות גדולות כיוון שאחרת התוכנה תהיה מאוד מסובבלת. הרבה יותר קל למצוא את הדרך בכמה קבצים, שככל אחד מהם אחראי לחלק אחד בתוכנה מאשר בקובץ אחד ענק. החלוקה לכמה קבצים או כמה "יחידות" היא חיונית.

ישנן כמה דרכים לחלק קוד לחלקים שנייתן לנוהל אותם ביחד. אחת השיטות הבסיסיות ביותר היא תכנות מונחה עצמים המאפשרת למתכננים לחלק את הפונקציונליות של התוכנה שלהם לחלקים שונים. כך חלקים שונים מתהווים עם אפיון פונקציונאלי ייחודי להם – אפשר אף לומר אופי. בתכנות מונחה עצמים לכל עצם ניתן לנכס שני סוגים תכונות: מאפיין או מתודה. כך שבתום הגדרת העצם יש לנו יחידה לוגית בעלי מאפיינים ויכולות פונקציונליות הקשורות אליה באופן בלעדי. לדוגמה תחשבו על כלב – לכלב יש מאפיינים ייחודיים ויכולות פונקציונליות, נבייה למשל. במסגרת השיטה זו אנו משתמשים על המערךת מכלול. אם אפשר לחלק את המכלול לעצמים בעלי מכנה משותף נוכל ליצור "מבנה" יותר מופשטות שמכילות כמה סוגים עצמים. למשל יונקים, או בעלי חיים בדוגמה הכלב.

הבה ונדגים בדוגמה מהחיים. בפרק על AJAX הראינו איך יוצרים קשר עם API של בדיחות צ'אק נוריס. הבה ושוב נניח שיש שני סוגים קריאות. הראשונה לבדיחות צ'אק נוריס על אוכל:

<https://api.chucknorris.io/jokes/random?category=food>

והשנייה לבדיחות צ'אק נוריס על כסף:

<https://api.chucknorris.io/jokes/random?category=money>

נניח ומבקשים ממי להציג את שתי הקריאהות לקומפוננטה. אם אין לנו ארכיטקטורה מסודרת – המכונת אוטו לסייע לוגי של הקוד שלו, אני פשוט אצור שתי פונקציות עם קריאהות fetch. אם יבקשו ממני קריאה שלישית אני אצור קריאה נוספת נספת וכן הלאה. אבל זה לא תכונן נבון של המערכת, כי אם יהיה עוד ועוד קריאות כאלה, אני אלץ לייצור עוד ועוד פונקציות ואז יהיה קשה לתחזק אותן. (ותחזקה היא 90% מהഴור חיים של כל תוכנה). יתרה מכך, ברור לנו שיש כאן מכנה מסוות לכל קריאות-ה-API שלנו, ואם הכתובת של ה-API תשתנה, למשל, אצטרך לשנות את הכתובת בכמה פונקציות. וזה מתכוון לטעויות אנוש. במקרים אחרים באגים, ההופכים את התוכנה ללא יציבה.

מה הדרך? הדרך היא לדמיין עצם בסיסי בעל המאפיינים והפונקציונליות הכליליות שאנו מחפשים, ולהגדיר את התבנית שלו בצורה שמאפשר למחרז את התבנית לצרכים השונים שלו. לדוגמאות אלו קוראים בתכנות מונחה עצמים "קלאסים" שעליהם למדנו. השם הרשמי שלהם במדעי המחשב נקרא "מחלקה", אך לא תמיד מוגדרת אחד בחימם האמיתיים שיקרא לפחות מחלוקת. בפרק הזה אנו משתמשים במונחים המקובלים בתעשייה ולפיין המחלוקת תיקרא בפרק קלאס. העצמים שנוצרים לפי המחלקות השונות הם למעשה מימוש של תבנית מוגדרת מראש. קוראים להם גם אובייקטים – עצמים.

לפני שנראה קוד חשוב לציין עוד שני דברים. אפשרות שנותן לנו תוכנות מונחה עצמים היא להגדיר תכניות מוכנות מראש, כך שנדע את היקף הפונקציונליות של עצמים שאנו עובדים איתם אף מבלי להכיר איך זה נעשה. וגם כדי שנוכל להרחיב את הפונקציונליות שלהם לבחירתנו. היכולת להרחיב פונקציונליות בתכנות מונחה עצמים נקראת ירושא. הכלב "ירוש" את תכונות ההולדה ואת מאפייני המין השונים של כל היונקים. כאשר יונקים הוא מחלוקת שמרחבה למקרים פרטיים של כלבים. כי כלבים יכולים גם לקשש בזבב. ירושא מאפשר להרחיב את הגדרת העצם בהתבסס על תכניות עצמים אחרות. וכך בג'אווהסקריפט תפגשו את המילה extends – מרחב.

עוד דבר חשוב שנציין כאן – הוא שישנם עוד דרכי בג'אווהסקריפט לקטלג פונקציונליות, "ולרשט" אותה. מסיבות היסטוריות תוכנות מונחה עצמים קלאסי איננו הצד החזק של השפה. לדבריו המיסדים של ג'אווהסקריפט נעשו טיעות שקשה לתכן. (Javascript: The Good Parts by Douglas Crockford). ג'אווהסקריפט מצטיינת בגישה התכונות הפונקציוני שלה, עליה יש פרק אחר בספר. כאן נראה את גישת תוכנות מונחה העצמים הקלאסית.

ניצור אובייקט או עצם של חיבור שהוא אב הטיפוס, ואז נירש ממנו את תכונותיו שיאפשרו לנו להתחבר ל-API. ככלمر באב הטיפוס תהיה הפקנציונליות המשותפת, האובייקטים היורשים ירჩיבו רק את הדברים שהן צריכים.

כך למשל באובייקט האב יהיה את החיבור עצמו ל-API, העיבוד של ה-JSON וההזרה של הבדיקה. כאשר מי שיורש מאוbjיקט האב יקבל את כל זה מובנה ויצטרך רק להגדיר על כתובת ה-API שמשמעותו.

```

class APIConnector {
  constructor(category) {
    this.category = category;
    this.url = `https://api.chucknorris.io/jokes/random?category=${this.category}`;
  }

  getJoke() {
    return fetch(this.url, {})
      .then((response) => {
        return response.json();
      })
      .then((jsonObject) => {
        return jsonObject.value;
      });
  }

  getCategory() {
    return this.category;
  }
}

class MoneyJoke extends APIConnector {
  constructor() {
    super('money');
  }

  calculateMoney() {
    console.log('Money not a problem when U R Chuck');
  }
}

let moneyJoke = new MoneyJoke();

moneyJoke.getJoke().then((joke) => {console.log(joke)})

```

בדוגמה זו יש לי קלאס – או תבנית עצם שנקרא APIConnector שהוא קלאס הבסיס שלו. ממנו כל הקלאסים יכולים לרשף. כך למשל קלאס MoneyJoke יירש ממנו וכאשר נוצר עצם מסוג MoneyJoke יוכל לרשף כל המאפיינים ומethods שהוגדרו בклאס

של APIConnector והוא ח奸ו על ידי הקלאס MoneyJoke. בדוגמה זו, הקלאס שיורש גם מפעיל את הקונסטרקטור של הקלאס המקורי עם התכונה שהוא רוצה. אם יש לקלאס היורש מתודות נוספות נוספות, הן מתווספות כרגע. למעשה זו תבנית מובנית בתכנות מונחה עצמים שברגע שעצם נוצר בדמות קלאס מסוים מוצעת מתודות ה-*constructor* בראשונה כדי לאותחל את היותה החדשה שנוצרה. האובייקט שלנו. קונסטרקטור גם עובר בירושה וכל הקונסטרוקטורים של הקלאסים האבות יופעלו לפי הסדר, אלא אם איזושי קלאס בדרך תמנע זאת על ידי קונסטרקטור משלו.

ככה אני יכול ליצור כמה קלאסים שניני רוצה שיורשים מה-*APIConnector*. מה היתרונות? אם כתובות ה-*API* משתנה, או הדרך שבה עובד החיבור ל-*API* משתנה, אני צריך לשנות את זה בקלאס אחד.

באו נחקרו עוד היבט של מערכת היחסים בין מחלוקת שיורשת תכונות מקלאס אב אחר. בדוגמה הבאה יש לנו שני סוגי משתמשים במערכת: משתמש פשוט ומשתמש מנהל. לכל אחד מהסוגים הללו יש תכונות ומchodות מיוחד לסוג המשתמש שלו. בדוגמה שלנו נגדיר קלאס *ShiaPiin* משתמש כלשהו – קלאס שיתאים לנו לכל מיני סוגים של משתמשים במערכת – גם למנהליהם, וגם למנהליהם פשוטים ואולי עוד סוגים משתמשים שנרצה להוסיף בעתיד. כי, זוכרים? 90% ממחזור חיי התכנה הוא במצב התחזקה, لكن סביר מאוד להניח שנרצה להוסיף משתמשים מסוגים שונים בעתיד.

```
class BaseUser {  
    constructor(name) {  
        this.name = name;  
    }  
  
    setName(name) {  
        this.name = name;  
    }  
  
    getName() {  
        return this.name;  
    }  
  
    setAddress(address) {  
        this.address = address;  
    }  
  
    getAddress() {  
        return this.address;  
    }  
}  
  
class AdminUser extends BaseUser {  
    constructor(name) {  
        super(name);  
        this.admin = true;  
    }  
  
    performAdminAction() {  
        // performs action allowed only to admins  
    }  
}  
  
class StandardUser extends BaseUser {  
    contactAdmin() {  
        // Contact the user admin  
    }  
}
```

از מה קורה כאן. הבה נפרק את זה לאות לאט:

1. הגדרנו קלאס בסיסי, שברגע שנוצר עצם מבוסס על הקלאס הזה תרוץ מתודת ה-

`const user_from_padua = new constructor`. זה נראה כך: – נן נוצר עצם בשם `user_from_padua`. ה-

`(BaseUser('Massimo'))`. `user_from_padua` – `constructor` שלנו מקבל פרמטר בשם `name`, ומציב אותו ל-`this.name`.

2. `this` הוא מילה שמורה של השפה המתיחס לעצם המפעיל את הקוד בזמן אמת,

ובמקרה שלנו זהו האובייקט `user_from_padua` ה-`this`, ובעת הפעלת ה-

`constructor` אנו מוסיפים לו תכונה בשם `name` עם ערך הפרמטר שקיבלנו:

`'Massimo'`. `this.name` כmo המילה "אני", כל אחד אומר את אותה המילה, אבל מתכוון

למשהו אחר – לדוגמה – לעצמו.

יצרנו עצם חדש ויחודי לפי התבנית (הklass, או המחלקה) של `BaseUser`, והגדנו

את המבנה הפנימי שלנו ב-`constructor` כך שתכונת `name` שלו תהיה ל-

`'Massimo'`.

3. לאחר מכן יצרנו קלאס חדש בשם `AdminUser`, המרחיב את הקלאס `BaseUser`

– זה בעצם מה שאומרת השורה: `class AdminUser extends BaseUser`:

4. שכתבנו את פונקציית `constructor`, מה שאומר לנו רוצים שבעם ייצור עצם

חדש מסוג `AdminUser` אנו רוצים שיבוצע קוד מיוחד שלנו. שהוא ...

5. ()`super` הוא קריאה לפונקציה של הקלאס שסמןנו אנו יורשים – דהיינו: אני רוצה

לשכתב פונקציה שקיימת במחלקה האב, שסמןנו אנחנו יורשים, ואני הרץ את

הלוגיקה המקורית לפני שאנו מוסיפים לוגיקה משלהנו. שימו לב, שלא חייבם

להרייך את `super` (עליוון באנגלית), אלא רק אם רוצים לשמור על פונקציונליות

הקודמת גם.

6. בנוסף בקלאס החדש אנו נוסיף עוד תכונה לעצם החדש שלנו בשם `admin`

ונאותחל את ערכה ב-`true`.

7. שימו לב, שלעצמם חדש יכולם להיות הרבה תכונות חשובות לפונקציונליות שלו,

אבל אנו נרצה "לחשוף" רק את הדברים החשובים לנו. זה נקראה API של הקלאס:

המספק דרכו נתחבר לפונקציונליות המיוחדת של אותו קלאס ללא הבנה לגבי איך

פונקציונליות זו מומשת. פרטי המימוש יכולים ואף במקרים רבים צריכים להיות

מוסתרים, ומתקנת טוב "יחשוף" רק מספק מינימלי.

ע"י "חשיפת" ממשק התוכנה אנו מאפשרים ליצור עצמים שונים בעלי יכולות ייחודיות שאפשר להשתמש בהם לפי הצורך. כמו ה-`APIConnector`, "החוšíף" לנו את יכולות חיבור לשרת מרוחק.

תכנון מערכת מאפשר לנו לדמיין את החלקים השונים של הקוד ואף, אם תרצו, להנפיש אותו דרך תכונות וمتודות ייחודיות. כך אפשר לעשות סדר במערכת גדולה ולהתחל לפרט רעיון מסוים לעצמם בעלי תפקידים ומערכות ייחסדים אחד עם השני.

Prototype based

כבר הזכרנו שג'אויסקייפ לא תוכננה לתוכנות מונחה עצמים כלליים. היה פשוט אחרת. מחלקות מופשטות וממוסמם הוא אשליה – סוכר סינטקטי – נועד בזמןנו להפוך את השפה לארקטטיבית בעניין מפתחי ג'אווה ועוד כל מיסי סיבות פוליטיות. למעשה ג'אויסקייפ חזקה יותר ממה שהיא נראה, ומאפשרת עוד לפחות שני סוגי של הרחבות פונקציונליות או כמו "ורושה" – מילה השאלה ממילון תכונות מונחה עצמים כלליים.

במקום לנצל את תכונות העצמים במחלקות מופשטות, בג'אויסקייפ "הגנון" של כל אובייקט נשמר אף הוא כאובייקט ובכך פותח אפשרויות נוספות אולם כדאי לזכור עמוק. כאן נביא רק שתי יתרונות חשובים של עבודה עם הפרוטוטיפ – הוא שם האובייקט נושא-hDNA של כל אובייקט בג'אויסקייפ:

1. עצם יכול לרשף פונקציונליות ישירות מעצם אחר, ולא מקלט מופשט של אותו עצם. כך אם אובייקט `a` מרחיב את פונקציונליות של אובייקט `a`, ומוסיף לו מתודה בשם: `() sayWhat`. עתה אובייקט חדש `c` יכול לרשף מאובייקט `a` וישראל לקבל את כל תכונותיו, ואף לשנות להרחב אוטם עצמו. אין צורך בהגדרת מחלקות המאפיינות שינויים אלה, אלה ישר להרחב אובייקטים.

2. דבר נוסף – סוג של קסם בג'אויסקייפ. אם ננסה לשנות את הפרוטוטיפ של אובייקט כל האובייקטים היורשים ממנו (בעל אותו הפרוטוטיפ) ישתנו מיד! לדוגמה אפשר להרחב את הפונקציונליות של אובייקט `String`-`String` המובנה של השפה שתכיל מתודה ... `hamm` ... `fire` שתדפיס את המחרוזת עם אמוגי של אש. ומיד כל העצמים מסוג `String` יוכל להשתמש בה.

זה נשמע אבסטרקטי, אך בוואו ונדגנים. הנה קלאס של משתמש.

```
class User {
  constructor() {
    this.type = 'user';
    this.theme = 'light';
  }

  setName(name) {
    this.name = name;
  }

  setAddress(address) {
    this.address = address;
  }
}
```

מה קורה מאחריו הקלעים כשהאני מבצע את הפקודה זו?

```
let newUser = new User();
```

מה שקורה הוא שנוצר אובייקט ריק בשם newUser עם רפרנס – ככלומר מצביע אל האובייקט המקורי User. בנויגוד למה שהיינו מצפים בשפה מונחית עצמים רגילה – שבה נוצר אובייקט שלם עם כל המתודות של הקלאס User. בג'אוوهסקריפט אין הבדל בין קלאס לאובייקט.

כשאני קורא ל:

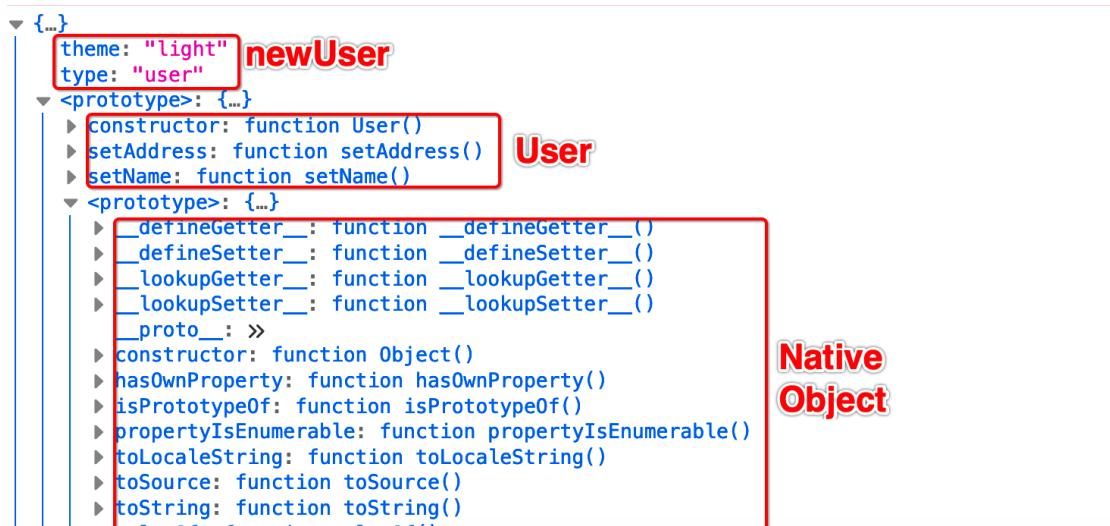
```
newUser.setName('Moshe');
```

ג'אווהסקריפט רاسית ניגשת לאובייקט newUser ובודקת אם יש לו את המתודה setName. אם אין לה את המתודה זו, היא עוברת לאבא – האובייקט (או הקלאס) User, אם היא מוצאת אותו שם.מצוין, היא תפעיל אותו. במידה ולא, היא תמשיך לאבא של User. מי האבא? האובייקט Object שנמצא בסיס ג'אווהסקריפט.

אתם מוזמנים להריץ את הקוד של הקלאס והאובייקט שנוצר מהקלאס:

```
let newUser = new User();
console.log(newUser);
```

תוכלו לראות בכל הפתחים שתחת prototype יש לאובייקט newUser את התכונות שנוצרו בפונקציה הבנאית אבל המתודות עצמן מקורן באובייקט האב – ה"קלאס" User. גם לו יש מתודות אחרות שמנויות מהאובייקט של ג'אווהסקריפט שכל האובייקטים יורשים ממנו.



למה חשוב לדעת את זה? כי מה לפוי דעתכם יקרה אם נכתוב את הקוד הזה?

```
let newUser = new User();
delete User.prototype.setName;
newUser.setName('moshe'); // TypeError: newUser.setName is not a function
```

נקבל הודעה שגיאה! מדוע? כי שוב, העובדה שאנו יוצרים newUser מקלאס User לא אומرت שמעכשו הקשר בין השינויים ניתק. להיפך, ה-newUser הוא אובייקט רוק שמכיל רק מה שקבענו לו בקונסטרקטור והמתודות שלו בעצם מגיעות מקלאס האב User. שינוינו את User? שינוינו את כל מי שיורש ממנו. כי אין כאן בעצם ירושה, יש כאן הפניה דרך ה-prototype. שינוינו את ה-prototype? שינוינו את כל מי שיורש ממנו.

זה יכול להיות שימושי כי אנו צריכים לזכור שגם האובייקט הבסיסי בג'אווהסקריפט, ה-**Object** שוכלים יורשים ממנו גם ניתן לשינוי. כך למשל:

```
class User {  
  
    setName(name) {  
        this.name = name;  
    }  
  
    setAddress(address) {  
        this.address = address;  
    }  
  
}  
  
class DataObject {  
  
    setName(name) {  
        this.name = name;  
    }  
  
    setProp(key, value) {  
        this[key] = value;  
    }  
  
}  
  
let newUser = new User();  
let dataObject = new DataObject();  
  
Object.prototype.isValid = true;  
  
console.log(newUser.isValid); // true  
console.log(dataObject.isValid); // true
```

מה קורה כאן מאחורי הקלעים?

יצרתי שני קלאסים ומهم יצרתי אובייקטים עם פונקציה בנאית. ואז הכנסתי לפורוטוטויפ של Object, האובייקט שהוא חלק מהשפה של ג'אווהסקריפט וכל האובייקטים יורשים ממנו את התכונה `isValid`.

אחרוי הקלעים נוצר בעצם עץ היררכיה הזה:

```
Object (object) -> User (object) -> newUser (object)  
Object (object) -> DataObject (object) -> dataObject (object)
```

כשאני מבקש `newUser.isValid`, מנוע הג'אווהסקריפט בודק ראשית אם זה קיים באובייקט `newUser`. זה לא? נמשיך לפורוטוטויפ של האבא, `User`. זה לא? נמשיך לפורוטוטויפ של הסבא, במקרה הזה `Object`, האובייקט האב של כל האובייקטים של ג'אווהסקריפט. האם הוא נמצא שם? כן! כי אנחנו הכנסנו `isValid` לפורוטוטויפ!

זה כבר מתחילה להיות מעניין, חישבו למשל שאני רוצה פונקציה שתדע אם לאובייקט שלי יש `name`. במידה וכן, הפונקציה תחזיר `true`. במידה ולא, הפונקציה תחזיר `false`. איך אפשר לעשות זהה דבר? בקלהות! על ידי תוספת לפורוטוטויפ של `Object`.

```

class User {

  setName(name) {
    this.name = name;
  }

  setAddress(address) {
    this.address = address;
  }

}

function isValid() {
  if(this.name && this.name !== "") {
    return true;
  } else {
    return false;
  }
}

Object.prototype.isValid = isValid;

let newUser = new User();
let another(newUser = new User();

newUser.setName('Moshe');

console.log(newUser.isValid()); // true
console.log(another(newUser.isValid()); // false

```

זה לא דבר טריוויאלי להבנה, אבל ברגע שambilינו אותו וمبינים את ה-prototype, אפשר באמת לנצל את הכוח והגמישות של ג'אוوهסקרייפט במקום להתייחס אליה בעוד שפה מונחית עצמים. הגמישות הזו מאפשרת לג'אוوهסקרייפט להתפתח ולהיות נפוצה בכל נסחף הרבה פלטפורמות.

ישנו עוד סוג של "ירושה" בג'אוوهסקרייפט – ירושה פונקציונאלית. זאת צורת הרחבת פונקציונליות הטבעית, החזקה והגמישה ביותר בג'אוوهסקרייפט. היא מאפשרת חיבור

אחרת ופושטה יותר על היררכיה פונקציונלית של הקוד, ואף מאפשרת לייצר תוכנות וمتודות מסוימות בرمות שונות של ירושה. השימוש בפונקציות – שהם בעצם גם אובייקטים מן המניין בג'אוوهסקריפט – חופש לנו את כוח ה-*closures* העוצמתי של השפה, המאפשר לנצל מetri נתוניים, וליצור קוד מוסתר מהיורשים ועוד. כתת עולם פיתוח יישומי frontend מאמץ קוד פונקציוני בשתי ידיים. אנו רואים זאת בכיוון של ריאקט למשל, והדברים הופכים לפשטיטים מאד, כאשר ג'אווהסקריפט לא מנשה לחוקות שפות אחרות, אלא משתמש בכשרונות הניתנים לה מלידה.

נספח: שינוי מהמהדורה הקודמת

(מהדורה 2.1.0)

תיקונים לפרק האחרון – הוספת איזכר לספר "לימוד פיתוח ווב מעשי בעברית".

נוסף נספח מפורט על טיופסקריפט.

בפרק על הקלאסים – עדכון מתודות ותכונות פרטיות לפי תקן ES2022 והוספת תרגיל בנושא.

תיקון בהערה של תרגיל בפונקציה `chz` בעמוד 136. בחלק של פונקציה `chz` כמשתנה, הופיע העירה שפלט הקונסולה צריך להיות `name` ולא פונקציה.

עמוד 254 – בפונקציה הבנאית של `Date`, כאשר אין ארגומנט נכתב בטיעות שהערך הוא `time`,. זה אינו נכון והתיקון הוא "כאשר אין ארגומנט, הזמן הוא השניה שעברה מה-`epoch time`".

עמוד 249 - "ניתן לגשת לאובייקט הגלובלי `windows`" תוקן ל-`window`. שם התקין של האובייקט הגלובלי.

עמוד 197 – בולולאת `for` " אחר כך מכניסים את המילה השמורה `ho` ואת המשתנה שבתוכו נמצא המערך". תוקן ל" אחר כך מכניסים את המילה השמורה `ho` ואת המשתנה שבתוכו נמצא האובייקט".

עמוד 149 - במערכות " תכונה נוספת שקיים וייחודה למערך" תוקנה ל"תכונה נוספת נוספת שקיים במערך" כיוון `sh-length` קיים גם במחזוזות טקסט.

עמוד 42 – תיקון שיכול מילוי. במקום " כך למשל אפשר לבצע על המשתנה המכיל את הפעולה טקסט " תוקן ל"כך למשל אפשר לבצע על המשתנה המכיל טקסט את הפעולה".

נספח: שינוי מהמהדורה הקודמת

(מהדורה 2.2.0)

נוסף תת פרק הדן ב-Chat GPT ואיך אפשר להשתמש בו על מנת ללמידה נוספת.

תיקוני שגיאות הקלדה קלות בנספח על טייפס קרייפט.

תיקון שגיאה מהותית בחלק על as. הדוגמה הראשונית לא עבדה כמו שצריך והוחלפה.

תודה רבה על הרכישה של הספר!

עובדתי מאד קשה על הספר זהה: שעות רבות של כתיבה, הגחה, תיקונים ומעבר על תוכרי העירינה. יותר מ-1800 אנשים תמכו בספר זהה ואייפשרו לו לצאת לאור.

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

העותק הזה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נჩבים פרטי הרוכש באופן שkopf למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העברו לו את הפרטים שלכם באתר ומיחקנו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!