תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר הזה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוצרי העריכה. יותר מ-1800 אנשים תמכו בספר הזה ואיפשרו לו לצאת לאור.

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

העותק הזה נמכר ל:

arik500@gmail.com

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

תודה וקריאה נעימה!

ללמוד תכנות ג'אווהסקריפט בקלות

רן בר-זיק

מהדורה: 1.0.0

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

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

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

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

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

עריכה לשונית: יעל ניר

הגהה: חנן קפלן

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

תוכן העניינים:

על הספר *12*

על המחבר *12*

על העורכים הטכניים *13*

דניאל שטרנליכט..... 13

גיל פינק..... 13

יגאל סטקלוב..... 13

תום ביגלאייזן 14

צחי נמני 14

על ג'אווהסקריפט *14*

איך לומדים *16*

ארגנו לעצמכם סביבת עבודה מסודרת..... 17

קראו את הפרקים לפי הסדר..... 17

קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות 17

פתרו את התרגילים לדוגמה בסוף כל פרק 17

התייעצו עם אחרים 17

הגיעו לסדנאות ולמיטאפים 18

תרגלו, תרגלו, תרגלו..... 18

התקנת סביבת העבודה ודרך הלימוד *18*

משתנים *24*

טקסט..... *27*

מספרים..... *31*

מציאת השארית 33

אופרטורים מקוצרים 34

סוגי מידע פרימיטיביים נוספים..... 37

בוליאני 38

משתנה לא מוגדר 38

ריק 39

Symbol..... 39

מציאת הסוג של המשתנה..... 40

הערות 42

בקרת זרימה – משפטי תנאי 43

אופרטורים השוואתיים נוספים..... 45

אופרטורים לוגיים 48

אופרטור שלילי 51

אופרטור תנאי 52

אופרטור המשווים ערכים..... 53

switch case 60

קבועים 67

בקרת זרימה – פונקציות 68

פונקציה עם ארגומנטים 73

ארגומנטים עם ערכים דיפולטיביים 74

הפונקציה כאובייקט 75

Hoisting 76

closure..... 77

פונקציה אנונימית ופונקציית חץ 79

פונקציה אנונימית כמשתנה..... 80

פונקציה אנונימית שמבודדת מהסקופ הגלובלי..... 82

אובייקטים 87

מחיקת מפתח 91
הכנסת פונקציה כערך..... 92
אובייקט כקבוע..... 92
מערכים 96
מערכים ומחרוזות טקסט 101
this-ו new 102
תבנית טקסט 108
לולאות..... 111
לולאת for 111
לולאה אינסופית 121
לולאת while 121
לולאת do while..... 123
לולאת forEach 123
לולאת for of 126
לולאת map..... 130
לולאת filter..... 132
לולאת sort..... 136
לולאות על אובייקטים..... 136
לולאות for in 136
Object.keys..... 139
ג'אווהסקריפט בסביבת דפדפן 145
הסבר כללי על HTML 145
מזהים של תגיות HTML..... 148
גישה אל תגיות באמצעות ג'אווהסקריפט 148
אלמנטים של HTML מתורגמים לאובייקטים של ג'אווהסקריפט..... 149
אירועים עם HTML וג'אווהסקריפט..... 150

הצמדת אלמנטים של HTML לאלמנטים אחרים של HTML באמצעות ג'אווהסקריפט 152
שינוי עיצוב 153
סלקטורים של DOM 155
אירועים נוספים 158
פעפוע של אירועים 160
הצמדת אירועי ג'אווהסקריפט לאלמנטים ב-HTML 164
דיבאגינג 170
אובייקטים גלובליים ואובייקטים מובנים 175
parseInt 178
eval 178
Math 179
Date 180
JSON 182
setTimeout 184
ביטויים רגולריים 187
טיפול בשגיאות 193
finally 196
מבני נתונים מסוג Map ו-Set 198
Map 199
Set 200
תכנות אסינכרוני – קולבקים 202
Promises 211
שרשור הבטחות 215
קיבוץ הבטחות 218
פונקציית async 221
AJAX 225

מתודות של HTTP וארגומנטים נוספים 227

ES6 Classes 229

ומה עכשיו? 234

נספח: Best Practices 236

מה זה Best Practices ולמה כדאי ליישם אותם? 236

Best Practices שכל מפתח מקצועי צריך להכיר 237

בחירת שמות 237

KISS 239

DRY 239

לא להמציא את הגלגל 239

Make it work, make it right, make it fast 240

תיעוד 240

ניהול גרסאות 241

לבקש עזרה 241

ביקורת עמיתים – Code Review 242

Tech Design 243

Best Practices ואוטומציה 244

מה זה linting? 244

ESLint 245

רשימה של Best Practices והחוק הרלוונטי של ESLint 246

בלי מספרי קסם 246

השוואה קפדנית: === 246

קוד לא נגיש 247

חלוקה לחלקים קטנים 247

אורך מינימלי ומקסימלי לשמות משתנים 248

בלי eval 248

בלי משתנים שלא הוצהרו 248

נספח: בדיקות, יציבות ואיכות קוד 249

קצת רקע 249

מבנה בדיקות 250
בדיקות יחידה 251
בדיקות קצה לקצה (End-to-End) 252
בדיקות ממשקי משתמש (UI Tests) 253
ספריות ופריימוורקים מומלצים 253
סיכום..... 254

נספח: Corvid by Wix 255

הקדמה 255

מה בונים? 255

איך מתחילים? 256

הצגת נתונים ממסד הנתונים באתר 258

Dataset 259
חיבור רכיבים למידע מהטבלה 260

הוספת משימה חדשה 262

Events 264
wixData.insert() – הוספת רשומה לטבלה 267

שינוי סטטוס המשימה 270

wixData.update() – עדכון רשומה בטבלה 272

מספר המשימות שלא הושלמו..... 273

wixDataQuery 274

1. wixData.query – אתחול השאילתה 274

2. בניית השאילתה..... 275
3. הפעלת השאילתה..... 275
$w.onReady() 277

סינון המשימות לפי סטטוס המשימה 278

1. wixData.filter() 279
2. בניית הסינון 279
3. dataset.setFIlter() – הפעלת הסינון על ה-dataset 280

ניקוי המשימות שהושלמו 282

wix-window..... 283
wixWindow.openLightBox()..... 283
wixWindow.lightbox.close() 285
wixData.remove() – מחיקת רשומה מהטבלה 286

נספח – יצירת מסד נתונים.....288

Sandbox | Live.....291

Permissions.....291

סיכום 292

על הספר

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

הספר מתחיל בלימוד בסיסי של משתנים ומגיע עד לימוד תכנות אסינכרוני ו-AJAX. בכל פרק בספר יש הסברים תיאורטיים לצד דוגמאות רבות הממחישות את העקרונות השונים שהוסברו בו, ובסופו תמצאו שאלות דוגמה לתרגול עם הסברים מקיפים על הפתרונות.

נוסף על הספר קיים אתר המכיל מאות תרגילים ופתרונות אשר יסייעו לכל הלומדים להשיג שליטה בעקרונות השפה ובתחביר שלה. הספר מיועד להביא אדם שלא מכיר את ג'אווהסקריפט לנקודה שבה הוא מכיר היטב את השפה ואת התחביר שלה ויודע איך להשתמש בה כדי לפתור בעיות בסיסיות.

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

על המחבר

רן בר-זיק הוא מפתח תוכנה משנת 1996 במגוון שפות ופלטפורמות ועובד כמפתח בכיר במרכזי פיתוח של חברות רב-לאומיות, מ-HPE ועד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הלקוח, הן בצד השרת, ושם דגש על בניית תשתית פיתוח נכונה, על שימוש ב-CI\CD וכמובן על אבטחת מידע.

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

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

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

על העורכים הטכניים

דניאל שטרנליכט

דניאל שטרנליכט הוא מפתח Frontend מאז גיל 14, המייסד של המיזמים Common Ninja ו- There is a bot for that, צרכן כבד של קוד Open Source (ומשתדל גם לתרום בחזרה) ונכון לזמן כתיבת הספר מוביל את גילדת ה-Frontend בחברת אאוטבריין.

נוסף על כך, דניאל הקים ומוביל את קבוצת הווטסאפ FEDs Community שמאגדת מפתחי Frontend מחברות מובילות בארץ ובעולם, ובה מעלים דיונים, מתייעצים ומשתפים לינקים, חדשות ועדכונים מעולם ה-Frontend.

בעבר כתב בבלוג "עיצוב גרפי וטכנולוגיה", וכיום הוא כותב בלוג טכני על טכנולוגיות Frontend ועל NodeJS.

בשאר הזמן הוא נשוי לרונה ואבא להדר ולאייל המתוקים, מתופף, גולש סקי, משחק כדורסל, צופה בסדרות, קורא ספרי פנטזיה ומתח ומדקלם את כל סרטי דיסני בעל פה.

גיל פינק

גיל פינק הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert, Microsoft Developer Technologies MVP והמייסד של חברת sparXys.

כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות ליחידים ולחברות המעוניינים להתמחות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official Course - MOC), מחבר משותף של הספר "Pro Single Page Application Development" (Apress) ושותף בארגון הכנס הבינלאומי AngularUP. לפרטים נוספים על גיל: http://www.gilfink.net

יגאל סטקלוב

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

בעבר היה ממובילי תחום ה-Frontend בחברות Wix ו-Netcraft והוביל פרויקטי פיתוח רבים.

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

תום ביגלאייזן

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

תום התחיל את דרכו בתחום ה-Frontend בתחילת שנות האלפיים מכיוון פחות צפוי – בפיתוח אפליקציות בג'אווהסקריפט לממירים של יס בסטארט-אפ קטן. אחרי שזה נסגר, עבד כעצמאי ובכמה סטארט-אפים, הפך לאחד המומחים הראשונים בארץ ל-cssi HTML ולא היה בקיא ממנו בבאגים של css עם rtl באקספלורר 6. תרם לפרויקטים שונים בקוד פתוח, היה שותף בגיור פלטפורמת ניהול התוכן דרופל ובנה את אחד הטמפלטים העבריים הפופולריים באותה תקופה. תום גם עיצב את האייקונים שעדיין נמצאים בשימוש עד היום בנגן הווידיאו 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 ואנימציות בפלטפורמת בניית האתרים של Wix.

צחי נמני

צחי נמני הוא מומחה בפיתוח, סרבר, אוטומאציה ודב אופס

מתמתחה בטנולוגיות הבאות: Java, Kotlin, Node Js, C#, Docker, Kubernets, Git, Selenium, Appium, Cypress ו Terraform

כיום הוא עובד בחברת הייטק בתחום התיירות.

בנוסף, צחי מייעץ ומרצה לחברות וליחידים בנושאים של CI/CD, Automation Development, Docker, Kubernetes ועוד.

כרגע הוא עובד על סדרה של הרצאות בנושא Docker and Kubernetes

צחי מאמין גדול בקוד פתוח ותורם לפרוייקטים שהוא משתמש בהם.

על ג'אווהסקריפט

ג'אווהסקריפט החלה את צעדיה הראשונים ב-1993 כשפה שפועלת בסביבת דפדפן ונועדה להעשיר דפי HTML. בג'אווהסקריפט יכולנו ליצור אנימציות ופידבק למשתמשים – כל דבר שהיה קשור לאתרי אינטרנט דינמיים, למשל דברים שנראים היום מאוד טריוויאליים ופשוטים כמו לחיצה על כפתור שעושה פעולה כלשהי בדף. אף על פי שההתחלה שלה הייתה צנועה, ברנדון איאך, ממציא השפה, יצר אותה מלכתחילה כשפה גמישה מאוד. הגמישות הזו, וגם חוסר ההבנה של רבים מהמתכנתים שהשתמשו בה בנוגע לעקרונות הבסיסיים שלה, גרמו ללא מעט מתכנתים בשפות אחרות לזלזל בה. גם השם שלה לא סייע לתדמית. השם ג'אווהסקריפט נקבע מסיבות שיווקיות בלבד

– JAVA היא שפת תכנות פופולרית, ואנשי נטסקייפ חשבו שכך יוסיפו לתדמיתה. בפועל השם הזה לא ממש עזר, ואין כמובן שום קשר בין ג'אווה לג'אווהסקריפט.

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

מיקרוסופט התנגדה בתחילה ליישום השפה ויישמה שפה משלה בשם Jscript בדפדפן אינטרנט אקספלורר, שהייתה בנויה בדומה לג'אווהסקריפט. למרות היריבות הגדולה בין אנשי מיקרוסופט לאנשי ECMA, שנוצרה כתוצאה מפיתוח שתי שפות שנשענות על שני תקנים מתחרים, העקרונות של ג'אווהסקריפט שולבו גם בגרסה של מיקרוסופט. הפופולריות של השפה עלתה כאשר מקרומדיה (יוצרת 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. בניגוד לשיחות שלום אחרות 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 הואצו דרמטית. דפדפן כרום, שמריץ ג'אווהסקריפט באופן יוצא דופן, נכנס אל השוק בסערה ואפשר למפתחי ג'אווהסקריפט לכתוב סקריפטים שפועלים על מנוע V8 העוצמתי של כרום ולהריץ ג'אווהסקריפט במהירות מסחררת. השימוש ב-AJAX תקשורת אסינכרונית עם השרת – נכנס לפעולה, החליף שיטות מיושנות כגון Long pulling ואפשר לאתרים לספק חוויות שימושיות מדהימות למשתמשים. בשנים האחרונות, פריימוורקים וספריות ג'אווהסקריפט אפשרו פונקציונליות מורכבת מאוד וספריות אחרות אפשרו כתיבה של ג'אווהסקריפט גם לטלפונים ניידים ואפילו בקלות. הראשונות שבספריות האלו נקראו MooTools ו-jQuery והן אפשרו לכל מתכנת לכתוב אפליקציות בקלות. הספריות האחרונות נקראות ריאקט, אנגולר ו-vue והן מאפשרות לבנות תוכנות מורכבות מאוד על גבי הדפדפן (צד הלקוח). ג'אווהסקריפט לא נותרה מוגבלת רק לצד הלקוח, כלומר לדפדפנים ולמכשירי קצה אחרים; המימוש של ג'אווהסקריפט לצד השרת, הידוע בכינויו node.js, הפך לפופולרי גם בשרתים. ג'אווהסקריפט מריצה כיום אפליקציות מורכבות גם בצד השרת, במיוחד אפליקציות שצריכות לבצע קריאות ולשרת מיליוני משתמשים. כיום אפשר למצוא ג'אווהסקריפט בכל מקום: באתרי אינטרנט, באפליקציות של טלפונים ניידים, באפליקציות המיועדות למחשבים רגילים וכמובן בשרתים. הביקוש למתכנתי ג'אווהסקריפט נמצא בשיאו ואין זה פלא – אפשר לעשות בשפה הזו המון דברים יישומיים כמעט מאפס. יש כל כך הרבה ספריות וכלי עזר, עד שכמעט בכל שבוע יוצאת ספרייה שימושית חדשה. בעזרת ידע מועט אפשר לעשות הרבה מאוד. מה שחשוב הוא ידע בסיסי בשפה.

בשנים האחרונות, תקן ES מתעדכן בכל שנה ומתווספים אליו תכונות ושימושים חדשים. ספר זה מעודכן לגרסה האחרונה של ECMAScript. חשוב לזכור שאם התקן מתעדכן, אין פירוש הדבר שהעדכון החדש מופיע מייד בדפדפנים שמריצים ג'אווהסקריפט או בשרתים שמריצים ג'אווהסקריפט, אלא לוקח זמן עד שהעדכונים החדשים

ביותר עושים את דרכם אל הדפדפנים/שרתים שכולנו משתמשים בהם. אם שמעתם מפתחי אינטרנט "מקטרים" על דפדפנים ישנים – זו בדיוק הסיבה.

זה המניע לכתיבת הספר. הבן שלי, כיום מתכנת בזכות עצמו, ניסה ללמוד ג'אווהסקריפט מאפס ולא הצליח למצוא ספרים בעברית. החומר שיש כיום בעברית בנוגע לג'אווהסקריפט הוא דל ומיושן. חלק מחוברות העזר הנמצאות בבתי הספר מתייחסות לתקנים שהפסיקו להיות בשימוש בשנת 2007! התייחסות אמיתית לתקנים החדשים ביותר של השפה, שיצאו בשנת 2017, אין בנמצא בעברית. התחלתי לכתוב הסברים עבור בני, ומפה לשם הבנתי שאני חייב לקחת את זה הלאה.

ג'אווהסקריפט היא שפה שקל ללמוד. בניגוד לשפות אחרות, לא נדרשים בה סביבת שרת מורכבת או כלי פיתוח שעולים כסף. לא נדרש ידע מקיף במדעי המחשב. כל שצריך הוא לפתוח Notepad במחשב, לפתוח דפדפן ולהתחיל ללמוד ולפתח. צריך גם הדרכה נכונה ומשמעת עצמית. אני מקווה שבספר הזה תמצאו לפחות הדרכה נכונה. המשמעת העצמית – עליכם. אני מאמין שככל שתתקדמו בספר תראו את פירות הלימודים, הניצוץ בעיניים יתחזק ולא תצטרכו עוד משמעת עצמית – אתם פשוט תתאהבו בג'אווהסקריפט בכלל ובפיתוח לווב בפרט. אל תדלגו על הפרק שבו אני מסביר איך ללמוד; זהו הפרק החשוב ביותר בספר.

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

רן בר-זיק

איך לומדים

לא למדתי מדעי המחשב באוניברסיטה או במכללה. למען האמת, עד גיל מאוחר מאוד לא למדתי תכנות בעזרת מדריך. רוב מה שאני יודע למדתי ללא הדרכה, ומכמה סיבות: הראשונה היא שכאשר עשיתי את צעדיי הראשונים בתכנות, בשנת 1996, החומר שהיה זמין באינטרנט היה דל מאוד. על חומר בעברית לא היה מה לדבר, והחומר באנגלית סיפק בדרך כלל רק את הדוקומנטציה. אני לא ממש מתגאה בכך; למידה לבד ללא הדרכה היא למידה מאוד לא יעילה. הדרכה משמעה לא רק מרצה מנוסה, אלא גם אתר אינטרנט שבו יש הסבר מקיף, פורום או קבוצה ברשת חברתית זו או אחרת (לא רק פייסבוק), שיש בהם אנשים מנוסים שיכולים לסייע או להפנות לחומרי עזר. היא גם מקום שבו אפשר לתרגל ולהתנסות. למרבה המזל, בימים אלו קיימים שלל חומרים, עזרה וסיוע. גם הספר הזה הוא הדרכה. לא תידרשו לצלול לתוך הדוקומנטציה של ECMAScript על מנת להבין את השפה, אבל גם בעזרת הספר תמצאו דרך טובה ללמידה.

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

ארגנו לעצמכם סביבת עבודה מסודרת

הפרק הראשון עוסק בבניית סביבת העבודה והוא הפרק החשוב בספר. ארגנו את המחשב שלכם וסדרו לעצמכם תיקייה מאורגנת שבה תשמרו את כל התרגולים. אם המחשב שלכם מקפיץ התראות של עדכוני ג'אווה או משהו בסגנון דומה, טפלו בכך.

ודאו שאתם יכולים להיכנס לאתר הלימודי שמלווה את הספר ושהסיסמה שלכם תקינה ופועלת.

קראו את הפרקים לפי הסדר

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

קראו כל פרק פעמיים וכתבו לעצמכם את כל הדוגמאות

קראו את הפרק פעם אחת קריאה שוטפת. לאחר מכן קראו אותו שוב. הפעם קחו את כל דוגמאות הקוד, העתיקו אותן לסביבת העבודה שלכם ושחקו בהן! נסו לשנות את הערכים, לגרום לשגיאות, לכתוב קוד דומה.

אני מאמין שקוד לומדים דרך הידיים ולא רק דרך הראש. חשוב להבין את התיאוריה ואת הרעיונות מאחורי מה שמנסים לעשות, אך ללא מימוש, הידע הזה יתנוון וייעלם, בדיוק כמו בשפת דיבור. אם לא תשתמשו ותתרגלו, גם הלימוד התיאורטי המעמיק ביותר לא יהיה שווה הרבה. הקריאה הראשונה נועדה ללימוד התיאוריה, הקריאה השנייה נועדה לתרגול. לימוד שפה הוא לא מרוץ! קחו את הזמן, כתבו את כל דוגמאות הקוד ונסו לכתוב כאלו משלכם.

פתרו את התרגילים לדוגמה בסוף כל פרק

בסוף כל פרק יש תרגילים לדוגמה. נסו לפתור אותם. אל תתייאשו מהר ואל תרוצו אל הפתרון אלא שברו קצת את הראש. הצלחתם לפתור? נהדר. קראו את הפתרון המוצע ואת ההסבר וראו אם הם דומים לשלכם או שונים במקצת. יש יותר מפתרון אפשרי אחד לכל בעיה...

התייעצו עם אחרים

יש לא מעט קבוצות בעברית (בפייסבוק, אבל לא רק) המיועדות ללימוד ג'אווהסקריפט ולדיון 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, ובעברית.

אל תהססו לקרוא, למשל דיון על משמעות השפה, להשתתף בדיונים או לסייע למישהו שהידע שלו דל משלכם. אל תשכחו להבין את רוח הדברים בקבוצה ולא להציק או להעיק. רוב המשתתפים בקבוצה הם אנשים עובדים שלאו דווקא זמינים להודעות מיידיות.

הגיעו לסדנאות ולמיטאפים

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

תרגלו, תרגלו, תרגלו

הספר לא שווה הרבה בלי תרגול מקיף ושימוש בו. אל תעברו לפרק הבא לפני שתסיימו את כל התרגולים שקשורים לפרק שקראתם. זה לא קריטי, אלא סופר-קריטי.

התקנת סביבת העבודה ודרך הלימוד

ג'אווהסקריפט יכולה לרוץ בסביבת שרת או דרך הדפדפן. איך זה בדיוק עובד? ג'אווהסקריפט נמצאת בקובץ טקסט. כן, בדיוק כמו זה שאפשר ליצור באמצעות כתבן הטקסט (Notepad) שיש בכל מערכת חלונות. בדפדפן מותקן כלי שלוקח את קובץ הטקסט הזה ומריץ אותו ואת הפקודות שנמצאות בתוכו. אם הדפדפן היה אדם, הוא היה פותח את קובץ הטקסט וקורא את מה שיש בתוכו, למשל: "לך ימינה ופתח את הדלת", ועושה בדיוק מה שכתוב. הפעולה הזו נקראת בלשון הפופולרית "רינדור", מלשון render בלעז. הדפדפן לוקח את קובץ הג'אווהסקריפט ומריץ אותו. קובץ הג'אווהסקריפט יכול לעשות כל מיני דברים ולהציג או לא להציג אותם.

איך הדפדפן טוען את קובץ הג'אווהסקריפט? יש כמה דרכים לעשות זאת, אבל כרגע ארצה ללמד אתכם איך לכתוב קובץ ג'אווהסקריפט ולראות אותו פועל על מנת להבין את כללי השפה ולכתוב משהו באופן ראשוני ביותר. החלק החשוב והקשה ביותר הוא יצירת סביבת העבודה, כלומר סביבה ממוחשבת שבה אפשר להקליד ג'אווהסקריפט ולראות אותה עובדת. סביבה זו היא חשובה מאוד כאשר לומדים, כיוון שלימוד של שפת תכנות נעשה ראשית כול "דרך הידיים" וחשוב מאוד לא רק לקרוא אלא גם לתרגל. וכדי לתרגל צריך סביבה שמאפשרת להקליד פקודות שפה, לשמור ולראות את הפלט. אני שב ומדגיש: התקנת הסביבה היא החלק הקשה ביותר בתחילת לימוד שפה חדשה וגם החשוב ביותר. לפיכך כדאי להיאזר בסבלנות, לקחת נשימה ארוכה ולזכור שדווקא עכשיו מתמודדים עם החלק הקשה ביותר.

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

שימו לב: מתכנתים מנוסים לא משתמשים בעורך הטקסט הפשוט אלא בעורך טקסט משוכלל יותר שנקרא IDE או Integrated Development Environment – סביבת פיתוח משולבת. הסביבה הזו מאפשרת השלמת קוד, מציינת שגיאות בכתיבה וגם יכולה להריץ את הקוד עצמו.

IDE מעולה הוא Visual Studio Code. הוא חינמי, מבוסס קוד פתוח, כתוב בג'אווהסקריפט (כן, כן), נתמך על ידי מיקרוסופט וניתן להורדה פה: https://code.visualstudio.com/. אחרי ההתקנה תוכלו לפתוח את התוכנה, לבחור ב-File ואז לפתוח את התיקייה שבחרתם ולפתוח או ליצור בה קבצים.

IDE מוצלח אחר הוא Atom. גם הוא חינמי ומבוסס קוד פתוח וגם הוא... כתוב בג'אווהסקריפט. הוא נתמך על ידי גיטהאב וניתן להורדה פה: https://atom.io/. הוא די דומה ל-Visual Studio Code ואחרי ההורדה וההתקנה שלו אפשר להפעיל אותו בקלות.

עורך טקסט נוסף שנחשב לאמין וטוב הוא תוכנה חינמית בקוד פתוח שנקראת Notepad++. הוא ניתן להורדה בקישור הבא: https://notepad-plus-plus.org/download והוא בסיסי יותר משני קודמיו.

שימוש ב-Visual Studio Code או ב-Atom הוא מומלץ יותר כיוון 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.

הסביבה הטובה ביותר ללימוד היא יצירת קובץ HTML שטוען קובץ ג'אווהסקריפט. קובץ HTML הוא קובץ שהדפדפן יודע לפרש ולהציג, והוא יכול לקרוא לקובץ ג'אווהסקריפט באופן כזה שהדפדפן ירנדר אותו. כאמור, רינדור הוא הרצת הפקודות שנכתבות בקוד ג'אווהסקריפט. יש ליצור במחשב תיקייה – זה יכול להיות ב"המסמכים שלי" או על שולחן העבודה – ובתוכה ליצור קובץ ששמו source.js וקובץ ששמו index.html.

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

בקובץ ה-HTML כתבו את הקוד הבא:

```
<!doctype html>
<html>

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

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

</html>
```

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

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

אחרי ששמרתם את תוכן שני הקבצים, פתחו את הקובץ בדפדפן כרום או בפיירפוקס (לא באדג'). אם הכול תקין, תראו שכתוב על המסך "Hello World!" – כתבתם את הג'אווהסקריפט הראשון שלכם!

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

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

1. בדקו שאתם פותחים את הקבצים בדפדפן כרום או בפיירפוקס. בדקו שאין תוספים מיוחדים לדפדפנים שעלולים לחסום את הרצת הקובץ על ידי הרצה של מצב פרטיות.
2. בדקו שהקלדתם את הטקסט כשורה בקובץ source.js ללא רווחים או תווים מיוחדים.
3. נסו להשתמש ב-Atom או ב-Visual Studio Code. שימו לב שאין התראות על שגיאות הקלדה. כאן למשל מובאת דוגמה של שגיאת הקלדה שביצעתי. אם קיבלתם התראה כזו, בדקו שוב שלא טעיתם בגרשיים ושלא הכנסתם רווחים מיותרים:

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

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

כיוון שאתם כבר מפתחי ג'אווהסקריפט מקצועיים, כדאי שתלמדו להשתמש בקונסולה של הדפדפן. מדובר בממשק מיוחד שמאפשר "לדבג" את ג'אווהסקריפט. הפועל "לדבג" כוונתו להסתכל על צפונות הרינדור ולראות ממש את הפלט של השפה. נשמע מסובך? יש להבין איך הדפדפן מנסה להריץ את הקוד שלו (מה שנקרא "רינדור", מלשון הרצה, באנגלית) ולקבל מידע נוסף במקרה שהוא נכשל, כלומר "זורק" שגיאה צבועה באדום בקונסולה ואף מפנה לשורה הבעייתית.

פתחו את כלי המפתחים של הדפדפן. בכרום ובפיירפוקס לחצו Ctrl + Shift + i אם יש לכם חלונות או Cmd + Shift + i אם יש לכם מק. אפשר לעשות את זה גם דרך התפריט העליון בשני הדפדפנים. לאחר פתיחת כלי המפתחים, לוחצים על 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 מחדש חשובה. הדפדפן לא יודע שהוכנסו שינויים בקובץ הג'אווהסקריפט, ויש לגרום לו לטעון מחדש את קובץ הג'אווהסקריפט על מנת להריץ את הפקודות החדשות. אחרי הטעינה מחדש הסתכלו על לשונית ה-Console. אם הכול תקין, המסך יהיה ריק, אך בקונסולה תראו Hello World! יש!

חשוב: אל תדלגו על השלב הזה. בכל שלבי הלימוד כדאי להשתמש בקונסולה, שהיא הרבה יותר נוחה להצגה. אם 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 את הדפדפן?

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

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

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

תרגיל:

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

פתרון:

היכנסו לקובץ source.js, מחקו 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. לחצו על Ctrl + Shift + i, לחצו על הלשונית Console וראו את התוצאות.

תרגיל:

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

פתרון:

צרו קובץ targil.js באותה תיקייה של הקובץ index.html. יש לוודא שזה שמו האמיתי של הקובץ ושמערכת ההפעלה לא מצמידה לקובץ עם סיומת txt. את זה עושים על ידי בדיקה בהגדרות התצוגה של מערכת ההפעלה. פתחו את קובץ ה-HTML בעזרת עורך טקסט (מומלץ להשתמש ב-Visual Studio Code) ושנו את שם קובץ הג'אווהסקריפט ל-targil.js.

```html
<!doctype html>
<html>

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

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

</html>
```

בקובץ targil.js כתבו את השורה הזו:

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

פתחו את הקובץ index.html בדפדפן ולחצו Ctrl + Shift + i על מנת להציג את כלי המפתחים. בחרו את הלשונית Console ובחנו את התוצאה.

משתנים

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

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

מה קורה פה? יש כאן הגדרת משתנה ששמו variant. ההגדרה נעשית באמצעות המילה השמורה let. מילה שמורה היא מילה מיוחדת בשפה שהשימוש בה שמור למקרה מסוים. במקרה הזה, let שמורה אך ורק להגדרת משתנים. לאחר מכן מכניסים ערך למשתנה. הפעולה הזו נקראת "הצבת ערך" או "השמה", והיא נקראת כך מכיוון שהערך מוצב בתוך המשתנה. במקרה הזה מדובר בטקסט הכולל את המילים Hello World.

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

אפשר לקצר ולהציב את הערך מייד בהגדרת המשתנה:

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

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

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

אם תציצו בקונסולה, תראו שמודפס המשפט "Hello World". מדוע? כיוון שזה מה שיש בתוך המשתנה.

מן הסתם, משתנה ניתן לשינוי. נסו את הקוד הזה:

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

מה לפי דעתכם יוצג בקונסולה? יוצג "I am a new version". למה? כיוון שהערך הקודם "נדרס" על ידי הערך החדש. הכנסתם (כלומר הצבתם) ערך חדש לתוך המשתנה, ועכשיו 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 בניית ארון או קופסה ואפשר בכל פעם להכניס לתוכו ערך אחר.

שימו לב: אחרי שמשתנה 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 כללים מחייבים. אפשר להשתמש בכל אות שהיא ובסימנים _ או $ בתחילת השם, ובכל האותיות והמספרים ובסימנים _ או $ בהמשך השם.

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

שימו לב: אפשר להשתמש בעברית בהגדרת המשתנים, אבל מומלץ שלא לעשות את זה גם כיוון שהקוד שלכם יהיה פחות קריא וגם כיוון שזה עלול להוביל להתנהגות מוזרה ולצרות. אתן לכם טיפ: בעולם התכנות אל תחפשו צרות כי יש מספיק מהן גם כך.

שימו לב: בגרסאות קודמות של ג'אווהסקריפט השתמשו במילה השמורה var להגדרת משתנה. בתקנים החדשים כבר לא מקובל להשתמש ב-var כיוון שלשימוש בו יש השלכות שנדון בהן בהמשך הספר. הוא נשאר איתנו בעיקר בשביל תאימות לאחור עבור קוד שנכתב בשנים עברו.

תרגיל:

צרו משתנה בשם המורכב לפחות משתי מילים, הדפיסו אותו בקונסולה וודאו שבהדפסה בקונסולה יופיעו המילים "I know JavaScript":

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

הסבר:

יצירת המשתנה נעשית באמצעות המילה השמורה let. השמת הטקסט נעשית באמצעות הסימן =. שימו לב שהטקסט מוקף בגרשיים. פקודת ה-console.log שלמדנו בפרק הקודם משמשת להדפסת הטקסט - ובמקרה הזה המשתנה שהיא מקבלת.

תרגיל:

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

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

הסבר:

יוצרים את המשתנה myVar ומכניסים לתוכו את הטקסט "me not know JavaScript" ממש ברגע היצירה. לאחר מכן דורסים את הערך הזה באמצעות השמה נוספת. 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 console.log:

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

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

הסבר:

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

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

טקסט

טקסט (באנגלית text string) הוא סוג מידע חשוב מאוד ובסיסי מאוד במערכות מידע. משתמשים בטקסט בכל מקום ובכל מערכת, ומדובר בחלק חשוב מאוד בלמידת קוד – יותר ממתמטיקה ומפעולות חשבוניות. בעבר היה קשה לעבוד עם טקסט שאינו באנגלית. אם אתם מבוגרים מספיק אתם ודאי זוכרים כל מיני אותיות מעוותות שהופיעו באתרים או בתוכנות. היום קל לעבוד במגוון שפות בתוכנות, באתרים וכמובן בג'אווהסקריפט, בזכות תקן חדש שנקרא "יוניקוד". יוניקוד מאפשר לכתוב בעברית, ביפנית, ברוסית ובכל מערכת כתב אחרת בקלות ובלי חשש להופעת ג'יבריש והוא מוטמע במחשב שלכם באופן אוטומטי.

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

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

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

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

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

```
let part1 = 'foo';
```

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

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

שימו לב: הרווח בין שמות המשתנים לבין הסימן + לא הכרחי להרצה תקינה אבל נוח מאוד לקריאה.

חשוב לציין שהשם התקני של טקסט הוא "מחרוזת טקסט".

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

נשאלת השאלה מה קורה אם רוצים להכניס מחרוזת טקסט עם גרש, משהו בסגנון הזה: I don't know JavaScript. לדוגמה:

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

יחזיר שגיאה בקונסולה. כיוון שכל מחרוזת טקסט חייבת להיות מוקפת בגרשיים, המנוע של ג'אווהסקריפט חושב שכל מה שמגיע אחרי הגרש הוא משתנה ולא יודע מה לעשות איתו. בדוגמה הבאה, החלק המודגש הוא מחרוזת הטקסט, והחלק שאינו מודגש הוא מה שהמנוע של ג'אווהסקריפט לא יודע מה לעשות איתו:

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

יש כמה דרכים לפתור את הבעיה הזו. הדרך הטובה ביותר היא לציין בפני המנוע של ג'אווהסקריפט שהגרש שיש במילה don't הוא גרש שמהווה חלק ממחרוזת הטקסט. איך עושים את זה? באמצעות הסימן \ :

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

ההדפסה תציג את הטקסט כמו שצריך, עם הגרש. הפעולה הזו נקראת escaping ובה לוקחים טקסט שהיה שובר את שפת התכנות והופכים אותו לבטוח. לחלופין, אפשר ליצור איתו תווים שלא ניתן לכתוב במקלדת, למשל ירידת שורה. איך כותבים ירידת שורה? בעזרת \n. העתיקו את הדוגמה הזו:

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

ותראו מה קורה. בקונסולה יופיעו שלוש שורות.

ייתכן שתיתקלו ב-escaping במקומות נוספים. אם רוצים לכתוב רק \, צריך לעשות לו escaping ולכתוב \\.

כעת, כשאתם מרגישים בנוח בכל מה שקשור לטקסט או, נכון יותר, למחרוזת טקסט בג'אווהסקריפט, הבה נסבך מעט את העניינים. בג'אווהסקריפט מחרוזת טקסט נחשבת לסוג מידע. באנגלית זה נקרא Data Type. יש כמה סוגי מידע, כמו מספרים למשל, אבל סוג המידע של מחרוזת טקסט נקרא string והוא עומד ברשות עצמו. כאשר יוצרים משתנה ומכניסים לתוכו טקסט, בעצם אומרים שהמשתנה הוא מסוג מחרוזת טקסט. ברגע שמשתנה מסוים מקבל לעצמו מידע והופך למשתנה מסוג מסוים, הוא מקבל גם סט של יכולות מיוחדות, ממש כמו קלארק קנט שהופך לסופרמן. מהשנייה שנכנס למשתנה טקסט, אפשר לעשות בו כמה פעולות שאפשריות אך ורק לסוג המידע של הטקסט.

כך למשל אפשר לבצע על המשתנה המכיל את הטקסט פעולה שתגרום לכל האותיות שלו להיות ראשיות (כלומר קפיטלס). לדוגמה:

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

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

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

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

מבלבל? הבה נדגים זאת שוב בפעולה נוספת. הפעם צרו משתנה והכניסו לתוכו טקסט. המשתנה הראשון, מהרגע שיש בתוכו טקסט, יכול לבצע פעולות של טקסט ולהחזיר את התוצאה אל משתנה 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, שאחד מהם מכיל את המילה Hello, והאחר מכיל את המילה World. חברו ביניהם, הכניסו את התוצאה אל משתנה שלישי והדפיסו אותה:

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

הסבר:

יוצרים ומציבים ערכים במשתנים בשם myVar1 ו-myVar2 (יש להקפיד שמחרוזות הטקסט יהיו מוקפות בגרשיים בודדים). מגדירים משתנה שלישי בשם answer ושמים בתוכו את הסכום של myVar1 ו-myVar2. ההדפסה נעשית כרגיל. שימו לב שרווח גם נחשב לאות. אם לא מכניסים רווח באחת ממחרוזות הטקסט בתוך המשתנים, לא יהיה רווח.

תרגיל:

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

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

הסבר:

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

תרגיל:

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

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

הסבר:

יוצרים שני משתנים ריקים בשם myVar ו-answer. הראשון מכיל את מחרוזת הטקסט JavaScript. אחרי שמכניסים לתוכו את הערך הזה, הוא מסוג טקסט ואפשר להשתמש בפעולה toUpperCase. את תוצאות הפעולה הזו מכניסים ל-answer ומדפיסים. אחרי ההדפסה משתמשים בפעולה toLowerCase על מה יש ב-myVar, מכניסים את התוצאה אל answer ומדפיסים.

מספרים

בדיוק כמו שאפשר להכניס טקסט לתוך משתנים, אפשר להכניס אליהם מספרים. למשל:

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

המספר 12 הוא ללא גרשיים. הוא ממש נכנס כמו שהוא, כיוון שהוא מספר. מספרים טהורים יכולים להיכנס ללא גרשיים. להזכירכם, מחרוזת טקסט, שעליה למדנו קודם, מוקפת בגרש או בגרשיים. מספרים לא מוקפים בגרש או בגרשיים.

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

אם מחברים בין מספרים אזי החיבור ייעשה בדיוק כפי שהייתם מצפים:

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

אפשר לבצע פעולות אפילו בשלב ההשמה. למשל:

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

התשובה שתודפס תהיה 11. 2 ועוד 4 ועוד 5.

בדומה לחיבור, אפשר לעשות פעולת חיסור בעזרת הסימן "-". למשל:

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

שימו לב שהתוצאה 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- ועוד 12-.

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

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

התוצאה כאן תהיה 16. 2 כפול 8.

אפשר גם לבצע חילוק בעזרת הסימן "/".

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

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

אפשר לעבוד גם עם חזקות. חזקות מסמנים 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 המיוחדות של מחרוזת טקסט שראיתם שאפשר להפעיל על משתנים מסוג מחרוזת טקסט, גם למשתנים מסוג מספר יש פעולות מיוחדות. למשל, פעולה המגבילה את המספר לאחר הנקודה העשרונית. אם תחלקו 10 ב-3, תקבלו 3.3333333 עד דלא ידע. אפשר להעיף את הנקודות העשרוניות. איך? באמצעות פעולה מיוחדת בשם Math.round():

```
let foo = 10;
let bar = 3;
let answer = foo / bar;
```

```
let finalAnswer = Math.round(answer);
console.log(finalAnswer);
```

הבה ננתח את קטע הקוד הזה.

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 הזה? 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. לְמה? ל-3. אם תדפיסו את finalAnswer תקבלו 3.

הינה טבלה קצרה שמסכמת את כל הפעולות הבסיסיות שאפשר לעשות:

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

מציאת השארית

אם אתם זוכרים מתקופת בית הספר היסודי, יש דבר כזה שנקרא שארית. כמה זה שש חלקי ארבע? אחת ושארית שתיים. כלומר, החלק מהמספר השלם שהוא קטן מהמחלק. תשע חלקי שמונה 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). אפשר לקרוא לו שארית:

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

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

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

אפשר להציב את תוצאות האופרטורים ישירות בתוך המשתנה. למשל, אם רוצים להציב בתוך משתנה את התוצאה של 4 כפול 2, אין צורך ליצור משתנה שיכיל 4, משתנה שיכיל 2 ומשתנה שיכיל את המכפלה של שניהם, כלומר משהו כזה:

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

במקומו, אפשר לכתוב משהו כזה:

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

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

אופרטורים מקוצרים

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

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

מה יש פה? מגדירים משתנה ששמו הוא 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 ההפוכה של חיסור:

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

כאן למשל יוצרים משתנה ומציבים בתוכו את המספר 10. באמצעות האופרטור -- מורידים ממנו 1. כיוון שחוזרים על התוצאה שלוש פעמים, המשתנה שווה ל-7.

הבעיה היא שזה מכוער... יש גם אופרטור קיצור שיכול להחסיר או להוסיף איזה מספר שרוצים. כך, למשל, גם הקוד הזה ידפיס 7.

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

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

הינה טבלה חלקית של האופרטורים המקוצרים הנפוצים ביותר:

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

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

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

תיתן את התשובה המדהימה מחרוזת טקסט של 11. למה? כי bar הוא מסוג טקסט. אם אתם כבר מכירים שפת תכנות, זה השלב שבו אתם מתמלאים בזעם או בתדהמה. אבל כאמור ג'אווהסקריפט היא שפה סלחנית ותנסה להגיע לפשרה אם מדובר בחיבור בין טקסט למספר. בחיבור בין מספר לבין חלק מסוגי המידע האחרים, תתקבל שגיאה של NaN, שהיא ראשי תיבות של Not a Number.

זה נראה משונה לעין בלתי מיומנת, אבל ג'אווהסקריפט היא שפה שסוגי המשתנים בה הם implicit – כלומר המנוע של השפה מנסה לנחש אותם ולא ממהר לזרוק שגיאה כמו בשפות אחרות. כאמור, משתמש לא מיומן עלול להתבלבל, ולמתכנתים בשפות אחרות זה נראה כאוטי, אבל ברגע שמתרגלים - מדובר בתכונה נוחה מאוד.

תרגיל:

צרו שלושה משתנים שיכילו את המספרים 2, 4 ו-6 והציגו את תוצאת החיבור שלהם:

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

הסבר:

יוצרים שלושה משתנים ושמים בתוכם מספרים באופן מקוצר כפי שלמדנו. שלושת המספרים מחוברים והתוצאה נכנסת למשתנה answer, שבתורו מודפס.

תרגיל:

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

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

הסבר:

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

חשוב: צריך לשים לב שסוג המידע של המשתנה השתנה וכדאי להימנע מלהציב באותו משתנה סוגי מידע שונים בשלבים שונים של ריצת תוכנית. זה מתכון לטעויות בהמשך.

תרגיל:

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

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

הסבר:

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

תרגיל:

צרו משתנה שיכיל את השארית של 10 חלקי 3:

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

הסבר:

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

תרגיל:

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

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

או

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

הסבר:

האופרטורים המקוצרים מאפשרים להוסיף 1 בקלות. יוצרים משתנה ומציבים בו את המספר 10. השתמשתם באופרטור המקוצר ++ להוספת 1 בדיוק למשתנה הזה. אפשר להשתמש באופרטור המקוצר 1=+ כדי להוסיף 1.

סוגי מידע פרימיטיביים נוספים

בפרקים הקודמים למדנו על מידע מסוג מחרוזת טקסט ומסוג מספר. סוגי מידע אלו נקראים "סוגי מידע פרימיטיביים". כשאומרים שסוג מידע הוא פרימיטיבי לא מתכוונים לכך שהוא לא אוכל בסכין ובמזלג אלא שהוא מהווה יחידת מידע בסיסית ולא מתוחכמת שבאה עם השפה. חוץ ממספר וממחרוזת טקסט יש עוד כמה סוגי מידע כאלה.

בוליאני

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

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

שימו לב שה-true, בדיוק כמו מספר, אינו מוקף בגרשיים. אם הוא היה מוקף בגרשיים הוא היה מידע מסוג מחרוזת טקסט. פעולה מיוחדת שיש לסוג מידע בוליאני היא המרה למחרוזת טקסט באמצעות הפעולה toString():

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

מה קורה פה? קודם כול יוצרים משתנה ומכניסים לתוכו את הערך true. הערך הוא ערך בוליאני. ה-true מגיע ללא גרשיים כיוון שהוא מילה שמורה. אם תדפיסו אותו, תראו שהקונסולה מראה אותו כ-true. מהרגע שהמשתנה הזה קיבל ערך בוליאני, הוא מסוג מידע בוליאני, ועל מידע בוליאני אפשר להשתמש בפעולות מיוחדות. אחת מהן היא toString(), שגורמת לערך הבוליאני להפוך למחרוזת טקסט. השורה השלישית מכניסה את הערך של המשתנה הבוליאני אל המשתנה bar ואז מדפיסים אותו. חדי העין יבחינו שיש הבדל בקונסולה בין ההדפסה הראשונה לשנייה. הסיבה היא שבהדפסה הראשונה המשתנה הוא בוליאני ובהדפסה השנייה הוא מחרוזת טקסט.

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

משתנה לא מוגדר

"לא מוגדר" (undefined) הוא הערך הראשוני של כל משתנה עוד לפני ההגדרה 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). כלומר, המשתנה לא מכיל דבר. שימו לב שלא מדובר ב-0 ולא במחרוזת ריקה, אלא בכלום. ריק. ההגדרה 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);
```

שימו לב: ההבדל בין undefined ל-null לא נראה כרגע משמעותי אך הוא משמעותי מאוד. undefined משמעותו משתנה שהוגדר אך לא אותחל בערך כלשהו. null משמעותו משתנה שהוגדר ואותחל כ-null. כדאי לזכור שמדובר בשני סוגי מידע שונים לחלוטין. המידע הזה יהיה חשוב בהמשך הדרך. כדאי לציין שיש כאלו שחושבים שמדובר בטעות בעיצוב השפה וש-null לא אמור להתקיים.

Symbol

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

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

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

שימו לב: Symbol הוא סוג מידע שנכנס רק בגרסאות האחרונות של ג'אווהסקריפט ושימושי 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 המציגה 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, ל-foo יש סוג מידע טקסט. באמצעות האופרטור typeof, המשתנה bar מכיל את המידע על סוג המידע של foo. אם תדפיסו אותו תראו string.

לעיתים typeof עלול להטעות ועדיף 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:

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

הסבר:

המשתנה הבוליאני foo נוצר עם הערך false ללא גרשיים. אם הייתם יוצרים אותו עם גרשיים היה נוצר משתנה של מחרוזת טקסט. המשתנה הבוליאני יכול להפעיל על עצמו פעולה מיוחדת של toString() שממירה את הערך שלו למחרוזת טקסט. מחרוזת הטקסט הזו נכנסת למשתנה 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:

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

הסבר:

יצירת הערך הבוליאני נעשית באמצעות המילה השמורה true, שיוצרת משתנה מסוג 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 והמירו אותו למשתנה שאין בו ערך:

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

הסבר:

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

תרגיל:

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

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

הסבר:

בתחילת הקוד יוצרים משתנה foo ובודקים אותו באמצעות typeof שערכו נכנס ל-bar. אפשר לראות שבהתחלה foo הוא undefined. כשמכניסים לתוכו מספר, סוג המידע בו הוא number וכשמכניסים לתוכו מחרוזת טקסט, ששונה מהמספר בכך שיש סביב המספר גרשיים, הוא מסוג string. כשמכניסים ערך בוליאני, שהוא המילה השמורה true, הסוג הוא Boolean, וכשמכניסים Symbol הוא הופך לסוג Symbol. לבסוף מכניסים את הערך null. ה-typeof שלו הוא object מסיבות היסטוריות שלא אכנס אליהן פה.

הערות

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

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

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

כאן למשל התוצאה תהיה undefined, כי ההגדרה של a נמצאת בהערה. מקובל בהערות לשמור על רווח בין שני תווי ההערה // להערה עצמה, אך אין חובה לעשות זאת.

לעיתים רוצים לכתוב הערה בכמה שורות. לפיכך משתמשים בתווים "/*" ו-"*/". בתחילת גוף ההערה כותבים /* ובסיומה */:

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

במקרה הזה דבר לא יודפס כי גם הגדרת המשתנה וגם ההדפסה 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 11. באמצעות האופרטור המקוצר ++, שכבר למדנו עליו, נוסף 1 למשתנה הזה. על מנת לציין את הערך החדש נוספה הערה.

תרגיל:

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

```
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 מאוד להבין מה bar מכיל.

בקרת זרימה – משפטי תנאי

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

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

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

משפט התנאי if מאפשר לשאול שאלה, במקרה הזה אם foo שווה ל-5. סימן השוויון הוא === והוא בודק אם מדובר במספר המבוקש, במקרה הזה 5. אם התנאי נכון כל מה שקורה בסוגריים המסולסלים מתרחש. אם לא, לא קורה כלום.

שימו לב: === נקרא אופרטור. כן, בדיוק כמו האופרטורים של החיבור, החיסור, הכפל ודומיהם, שלמדנו בפרק המספרים. האופרטור הזה הוא אופרטור של השוואה, ובניגוד לאופרטור נומרי (של מספר) הוא מחזיר ערך בוליאני. כשלמדנו עליו הוא היה ערטילאי מאוד. עכשיו הוא קצת יותר הגיוני, אבל רק קצת. בקרוב הוא ייחקר לעומק.

בדוגמה שלהלן, אם מדובר במספר מסוים אזי 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 בסוגריים המסולסלים מייד אחרי ה-if. אם לא, מתרחש מה שכתוב בסוגריים המסולסלים אחרי ה-else.

כמו שאפשר להשוות מספרים, אפשר להשוות גם מחרוזות טקסט. למשל:

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

באמצעות האופרטור ההשוואתי === בודקים אם המשתנה foo שווה למחרוזת הטקסט five. אם כן – הקוד שיש בתוך הסוגריים המסולסלים הראשונים יופעל והקונסולה תקבל "This is five". אם לא – רק הקוד שבסוגריים המסולסלים לאחר המילה השמורה 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 שילוב של else ו-if:

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

כרגיל, משפט התנאי if בודק אם foo שווה ל-5. אם כן, קטע הקוד שבתוך הסוגריים המסולסלים הראשונים יופעל. אבל כאן יש גם else if עם עוד תנאי, שבודק אם foo שווה ל-6. בסופו של דבר יש תנאי שתופס את הכול. אפשר לשלב כמה else if שרוצים, 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. שימו לב שיש תנאי else if מרובים בקוד הזה, שבודקים את המספר שוב ושוב, בכל פעם כנגד תנאי אחר. אם המספר הוא לא 1 ,2, 3 או 4, יופעל ה-else האחרון.

חשוב לציין שאם תוסיפו סתם else באמצע, לעולם לא תגיעו ל-else if שמופיע אחריו.

עוד משהו שחשוב לציין הוא שצריך להיזהר מהשמה בתוך משפט תנאי. ג'אווהסקריפט היא סלחנית מדי, ואם תכתבו x=y במקום x===y הקוד שלכם יעבוד ולא יזרוק שגיאה – זו הסיבה שבאופרטורים השוואתיים צריך לשים לב היטב מה כותבים ולעולם לא לכתוב = אחד בלבד.

אופרטורים השוואתיים נוספים

כמו === שבודק השוואה טהורה, יש אופרטורים השוואתיים נוספים, למשל !== (סימן הקריאה משמאל לשני סימני השווה) שבודק אי-שוויון ויפעל רק אם המשתנה לא שווה. למשל:

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

האופרטור !== בודק אם foo אינו שווה ל-1. במקרה הזה הוא שווה ל-4, וקטע הקוד בתוך הסוגריים המסולסלים יפעל. אושר גדול! הינה עוד דוגמה:

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

כאן בודקים מחרוזת טקסט. האם foo שונה מ-one? במקרה הזה כן, ולפיכך הקטע שבתוך הסוגריים המסולסלים 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('Bigger than 1');
} else {
    console.log('Smaller than 1');
};
```

במשפט התנאי בודקים אם foo גדול מ-1 באמצעות אופרטור אי-השוויון >, ממש כמו בתרגילי חשבון של כיתה א'. כיוון שבמקרה הזה foo שווה ל-10 ו-10 גדול מ-1, המשפט שבתוך הסוגריים המסולסלים יפעל.

מה לפי דעתכם יודפס בקונסולה אם תריצו את הקוד הבא?

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

התשובה היא "Smaller than 5". מדוע? משפט התנאי בודק אם foo גדול מ-5 ואם כן מדפיס את "Bigger than 5". אם לא, יודפס "Smaller than 5". כיוון ש-foo הוגדר כשווה ל-5, 5 אינו גדול מ-5 ולפיכך התנאי לא מתקיים. איך פותרים את העניין? אפשר להוסיף תנאי נוסף בעזרת else if:

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

אפשר גם להשתמש באופרטור נוסף שמשמעו גדול או שווה:

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

```
};
```

כאן אפשר להשתמש באופרטור =< שמשמעו גדול מ- או שווה ל-. במקרה הזה גדול מ-5 או שווה לו, וכיוון ש-foo שווה ל-5 התנאי מתממש.

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

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

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

הבה נבחן את כל האופרטורים ההשוואתיים שהכרתם:

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

הסבר מאחורי הקלעים בנוגע לאופרטורים השוואתיים

האופרטורים ההשוואתיים בעצם מתרגמים את הביטוי שמעבירים להם לערך בוליאני, כלומר אמת או שקר.

למשל:

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

זאת אומרת שכמו שהאופרטור הנומרי מוסיף או משנה מספר, האופרטור ההשוואתי 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 להחזיר את התוצאות של האופרטור ההשוואתי אל תוך משתנה, למשל אם כותבים את הקוד הבא:

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

אפשר לראות בקונסולה שזה true. משפט התנאי לא מתעניין באופרטורים, במשתנים או בקשקוש 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 יתממש אם מספר מסוים יהיה גדול מ-10 או קטן מ-0. איך עושים את זה? באמצעות האופרטור הלוגי "או" שנכתב כ-"||".

כך למשל:

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

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

הבה נדגים זאת שוב במחרוזת טקסט. נניח שתרצו לבדוק אם משתנה מסוים שווה למחרוזת הטקסט "sunday" או "Sunday". איך עושים את זה? ממש בפשטות. צריכים שאחד משני התנאים האלו יתממש:

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

או:

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

מממשים את ה"או" באמצעות אופרטור לוגי של "או", שכאמור נכתב ||. הינה, ממש ככה:

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

אפשר גם לשלב מספר רב של תנאים. למשל:

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

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

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

כדאי לשים לב שההשוואה היא משמאל לימין. אם התנאי הראשון מתממש, אין בדיקה של התנאי השני.

אופרטור לוגי נוסף הוא האופרטור "גם", שבו אפשר לקבוע ששני 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? הבה נסתכל על הדוגמה הבאה:

```
if (foo === 1 && bar === 1) {
    console.log('foo is 1 and bar is 1');
```

```
} else {
    console.log('bar is not one OR foo is not one');
}
```

יש כאן שני משפטי תנאי עם אופרטורים השוואתיים כמו שאנו מכירים וביניהם, כמו דבק, מחבר 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 סוגריים, ממש כמו במשוואות מתמטיות שלומדים בבית הספר. כך למשל כותבים משפט שבו בודקים אם foo או bar שווים ל-1 וגם baz שווה ל-1:

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

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

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

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

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

אם גם baz יהיה שווה ל-1, התנאי יתממש.

אפשר להתפרע כמה שרוצים עם סוגריים, למשל:

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

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

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

אופרטור שלילי

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

```
!true
```

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

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

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

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

אמנם יש שפות שמאלצות שימוש בתנאי שלילי, אבל הפיכת תנאים נחשבת מנהג פסול מאוד ולא הייתי ממליץ להשתמש בה. כדאי לדעת את זה על מנת להבין את פעילות האופרטור ! שבסופו של דבר ממיר false ל-true, אבל מומלץ מאוד לכתוב את התנאים כמו שצריך. למה כן משתמשים באופרטור הזה? הוא שימושי מאוד לבדיקה אם משתנה מסוים הוא "ריק" ולמניעת else מיותר. כשתצברו ניסיון בתכנות, תגלו שהמון פעמים צריך לבדוק אם משתנה מסוים הוא "ריק", וב"ריק" אני מתכוון ל:

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

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

```
let foo; //can be null, '', 0 or false.
```

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

כל הערכים האלו לבסוף מומרים ל-true על ידי האופרטור הלוגי ! ואז משפט התנאי עובד. בדרך כלל רואים את האופרטור הלוגי בשימושים כאלו.

לסיכום: יש שלושה אופרטורים לוגיים:

שם האופרטור	סימן
או	||
גם	&&
אופרטור שלילי	!

אופרטור תנאי

אופרטור תנאי, "if מקוצר" או ternary operator הוא האופרטור המשמעותי האחרון שתלמדו. בגדול, הוא מאפשר להכניס ערכים למשתנים לפי תנאים מסוימים. כך למשל אם רוצים שהמשתנה bar יהיה שווה ל-9 אך ורק אם המשתנה foo שווה ל-10. אם המשתנה foo לא שווה ל-10, 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 ה**סבר:** משפט תנאי מקוצר מתחיל בשם של משתנה, 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 לשתות אלכוהול" או "אסור לשתות אלכוהול" בהתאם לגיל, כך עושים את זה עם משפט תנאי מקוצר:

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

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

אופרטורים המשווים ערכים

יש סוג נוסף של אופרטורים שמשתמשים בהם פחות ואני באופן אישי לא משתמש בהם בכלל. אעבור עליהם בקצרה כיוון שהם מופיעים לא מעט פעמים, אבל השימוש בהם נחשב למנהג פסול.

נניח שיש את שני המשתנים הבאים:

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

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

```
foo++;
```

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

```
bar++;
```

ייכשל ותופיע שגיאת NaN, שמשמעותה היא "לא מספר". כי "1" זה לא 1.

לא מובן? דוגמה נוספת:

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

המשתנה 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 מחבר את מחרוזות הטקסט ואז 1 ועוד 1 שווה ל-11.

אבל כיוון שאתם מתכנתי ג'אווהסקריפט מנוסים, אתם כבר יודעים שזה נכון רק אם ה-1 הוא מחרוזת טקסט.

כשאתם חמושים בתזכורת הזו 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 ההשוואתי הוא == (שני סימני שווה ולא שלושה) או =! (במקום ==!). כך למשל:

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

התנאי שלעיל כן יתממש, כיוון שהאופרטור ההשוואתי == (להבדיל מ-===) מבצע את ההמרה של סוגי המידע ואז מבצע את ההשוואה – שווי בין ערכים ולא בין סוגי מידע.

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

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

איך בדיוק נעשה התרגום הזה? בשביל זה צריך להכיר את המונחים Truthy ו-Falsy – מאומת ושקרי. כאשר ממירים משתנה לסוג מידע בוליאני, בודקים אם הוא Truthy או Falsy. כל סוג מידע וערך בג'אווהסקריפט נחשב ל-Truthy למעט הערכים הבאים: false, 0, "", null, undefined ו-NaN.

כל מה שהוא Truthy מתורגם ל-true, כל מה שהוא Falsy מתורגם ל-false, בהמרה למשתנה בוליאני. וכך, כשמשתמשים בהמרות בוליאניות בהשוואה, כמו בדוגמה שלעיל, נתקלים בבעיה.

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

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

למה זה ככה? כי מחרוזת ריקה מתורגמת ל-false ו-0 גם הוא מתורגם ל-false. זוכרים ששניהם Falsy? אם כך, שניהם false ואז נעשית ההשוואה, וזה עלול להיות בעייתי. חשבו על מערכת שמקבלת קלט מספרי ועושה איתו דברים. אם נעשתה טעות ונשלח קלט ריק, ואז משתמשים בהשוואה רגילה, תהיה תקלה במערכת. היא תקבל קלט ריק ותחשוב שזה אפס, וההתנהגות בהתאם.

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

תרגיל:

צרו משפט תנאי שמתרחש אם המספר foo שווה ל-10:

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

הסבר:

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

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

קטע הקוד הזה, לעומת זאת, לא ידפיס אותו:

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

למה? כי 10 הוא לא 9.

תרגיל:

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

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

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

יפעל. אם לא, קטע הקוד בתוך הסוגריים המסולסלים מייד 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 הודעת שגיאה.

```
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 משפטי תנאי שבודקים את המשתנה day עם אופרטור השוואה. אם המשתנה day הוא 1, כל קטע הקוד שבין הסוגריים המסולסלים שבאים אחר כך יעבוד, והקונסולה תדפיס את השם Sunday. מייד אחר כך יש else if שבודק אם המספר הוא 2, ואז הקונסולה תדפיס את השם Monday, וכך הלאה. שימו לב ל-if else ולסוגריים שבתוכם יש את התנאי ולסוגריים המסולסלים שפועלים אך ורק אם התנאי נכון.

בסוף הקוד יש else שמתקיים אם כל ה-if שהיו לפניו לא עבדו, ורק אם הם לא עבדו.

תרגיל:

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

```
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 משפטי 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.

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

תרגיל:

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

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

הסבר:

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

תרגיל:

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

```
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 ומדפיס בקונסולה "I am not here".

```
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 מתכנתי הג'אווהסקריפט היו בוחרים לעשות. באמצעות האופרטור הלוגי ! מקבלים true על foo אם הוא null, 0, מחרוזת טקסט ריקה או false. הפתרון השני הוא אינטואיטיבי יותר למי שעושה את צעדיו הראשונים בשפה ומכיל שני אופרטורים השוואתיים ואופרטור לוגי || שמשמעותו היא "או": foo שווה ל-0 או foo שווה ל-null. שימו לב ש-null היא מילה שמורה (לא מוקפת במירכאות). null הוא סוג מידע מיוחד שלמדנו עליו בפרקים הקודמים.

תרגיל:

כתבו משפט תנאי שמשווה בין:

```
let foo = 2;
let bar = ‘2’;
```

ומצליח.

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

הסבר:

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

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" וכך הלאה. אם לא מדובר במספרים 1–7, הקונסולה מדפיסה הודעת שגיאה. איך זה נראה? ככה:

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

יש דרך אלגנטית יותר לכתוב משהו כזה. switch case מאפשר לבצע השוואה של משתנה מסוים לערך כלשהו (מספר, טקסט, ערך בוליאני וכו') ואז להריץ קוד בהתאם לערך ולספק גם פעולת ברירת 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 בדיוק כמו הקוד הקודם. פותחים בהצהרת switch שהיא מילה שמורה. ב-switch שמים משתנה שיכול להכיל כל דבר ואחריו סוגריים מסולסלים. בתוכם יש הצהרות case. case 1 למשל אומר שבמקרה שהערך הוא 1 – שימו לב שמדובר פה במספר ולא במחרוזת טקסט – אם הערך עונה על התנאי הזה, כל הקוד שיש בין הנקודתיים ל-break מתקיים. אם שום תנאי לא מתקיים, כל מה שקורה אחרי ה-defaults מתקיים עד ה-break. לא חייבים את ה-break האחרון, אבל הצבתי אותו בשביל הסדר הטוב.

break היא מילה שמורה שמשתמשים בה כדי לשבור את ה-switch ולצאת ממנו.

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

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

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

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

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

```
let monthNumber = 5;
let monthName;
```

```
let season;
switch (monthNumber) {
    case 1:
        monthName = 'January';
        season = 'Winter';
        break;
    case 2:
        monthName = 'February';
        season = 'Winter';
        break;
    case 3:
        monthName = 'March';
        season = 'Winter';
        break;
    case 4:
        monthName = 'April';
        season = 'Winter';
        break;
    case 5:
        monthName = 'May';
        season = 'Summer';
        break;
    case 6:
        monthName = 'June';
        season = 'Summer';
        break;
    case 7:
        monthName = 'July';
        season = 'Summer';
        break;
    case 8:
        monthName = 'August';
        season = 'Summer';
        break;
    case 9:
        monthName = 'September';
        season = 'Summer';
        break;
    case 10:
        monthName = 'October';
        season = 'Winter';
        break;
    case 11:
        monthName = 'November';
        season = 'Winter';
        break;
    case 12:
        monthName = 'December';
```

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

גם פה, כמו בקטעי הקוד הקודמים, לוקחים משתנה מסוים ושמים אותו בתוך ה-switch. switch היא כאמור מילה שמורה שמודיעה למנוע של ג'אווהסקריפט שיש פה תנאי switch case. מייד אחרי ה-switch והמשתנה שנמצא בסוגריים העגולים יש סוגריים מסולסלים, המכילים תנאים שנקראים case. כל case כזה מכיל ערך שהמשתנה נבחן מולו. אם המשתנה זהה לערך הזה, כל הקוד מהנקודתיים ועד המילה השמורה break עובד.

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

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

```
        season = 'Summer';
        break;
    case 10:
        season = 'Winter';
        break;
    case 11:
        season = 'Winter';
        break;
    case 12:
        season = 'Winter';
        break;
    default:
        monthName = 'n/a';
        break;
}
console.log(season); //Summer
```

אפשר לעשות משהו כזה:

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

זה קורה כי אם לא משתמשים במילה השמורה break, הקוד לא יעצור ואפשר להכניס כמה תנאים שרוצים. נשאלת השאלה, למה להשתמש בזה? אחרי הכול, בעזרת if else אפשר לכתוב את הקוד שלעיל בלי ללמוד switch case

מעיק. זו טענה ולידית, אבל כאן נכנס הנימוק של קוד קריא ואלגנטי. המטרה שלכם כמתכנתים היא לכתוב קוד קריא שקל לכל אחד (גם לכם) להבין. שימוש מרובה ב-if else הוא מבלבל וקשה לקריאה ולתיקון. בלא מעט מקרים, switch case יכול להקל את הקריאה ומשתמשים בו הרבה. לפיכך כדאי וצריך להכיר אותו ולהשתמש בו בכל פעם שיש יותר משלושה if else. זה לא כלל ברזל, אלא רק במקרה של תנאי לא מסובך מדי – אתם לא רוצים להתחיל להכניס משפטי תנאי והצבות בתוך ה-switch case שלכם.

תרגיל:

באמצעות switch case, כתבו קוד שלוקח שם של יום ומדפיס בקונסולה את מספר היום.

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

הסבר:

כותבים את המילה השמורה switch, שמקבלת משתנה בסוגריים העגולים. המשתנה הזה ייבחן כנגד ערכים שונים. פורמט הבחינה הוא כזה:

case value:

המילה השמורה case ומייד אחריה מגיע הערך. אם הערך הזה תואם את הערך שיש במשתנה שנמצא בסוגריים העגולים ליד ה-switch, הקוד שמופיע מייד אחרי הנקודתיים ועד ה-break ירוץ. במקרה הזה מדובר בהדפסה בקונסולה.

שימו לב ששמתי ערך דיפולטיבי שמודפס אם במשתנה יש ערך שלא ציינתי במפורש במבחנים. זה נעשה באמצעות התנאי default.

קבועים

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

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

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

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

```
const MY_CONSTANT = 10;
```

קונבנציה אחת שבה משתמשים כדי להגדיר אותם היא אותיות גדולות עם קו תחתון שמפריד בין מילים, אבל ראוי לציין שממש לא חובה לעשות את זה. אפשר להגדיר קבוע בדיוק כמו כל משתנה אחר.

אם מנסים לשנות את הקבוע, מקבלים שגיאה בזמן הרצת הקוד.

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

נשאלת השאלה, למה להגביל את עצמנו ולהשתמש בקבועים? קבועים יעילים יותר מבחינת זיכרון מסיבות שלא אכנס אליהן כאן. ואם זה נשמע לכם אזוטרי, אפשר לבוא ולומר ששימוש מושכל ב-const מעלה את ערככם בעיני כל מי שיקרא את הקוד שלכם. לפיכך כדאי וצריך להכיר אותו.

תרגיל:

הגדירו קבוע בשם foo והכניסו לתוכו מחרוזת טקסט. הדפיסו את הקבוע:

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

הסבר:

הגדרת הקבוע נעשית באמצעות המילה השמורה const. לאחר מכן אפשר להדפיס את הקבוע בדיוק כמו משתנה.

תרגיל:

הגדירו קבוע בשם foo, הכניסו לתוכו מספר, שנו את המספר ובחנו את התקלה שהקונסולה מציגה.

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

הסבר:

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

בקרת זרימה – פונקציות

פונקציה היא סוג של רכיב בג'אווהסקריפט שמחזיר תוצאה. אפשר להשתמש בפונקציה כאבן בניין לבניית דברים מסובכים והיא החלק החשוב ביותר בשפה.

פונקציה מוגדרת באמצעות המילה השמורה function, שם הפונקציה, סוגריים ריקים וסוגריים מסולסלים שבתוכם המילה השמורה return, שמחזירה ערך שהוא התוצאה של הפונקציה. הערך הזה יכול להיכנס לכל משתנה שהוא.

מבלבל? הבה נסתכל על הדוגמה הבאה:

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

ראשית הוגדרה פונקציה בשם myFunc. ההגדרה נעשית באמצעות המילה השמורה function ושם הפונקציה. במקרה הזה נבחר השם myFunc. אחרי השם יש סוגריים ריקים ואז סוגריים מסולסלים. בתוך הסוגריים המסולסלים מופיעה הפונקציה עצמה, יחידה אוטונומית של קוד שלא תרוץ אם לא נקרא לפונקציה. במקרה הזה אין מי-יודע-מה קוד, רק המילה השמורה return ו-1. כלומר מי שיקרא לפונקציה יקבל 1 בתגובה.

איך מריצים פונקציה? ככה:

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

כאן מגדירים את הפונקציה וקוראים לה. את תוצאות הפונקציה מכניסים למשתנה. מה לפי דעתכם יקבל המשתנה? את מה שהפונקציה מחזירה. במקרה הזה 1.

הבה ננסה ליצור עוד פונקציה:

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

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

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

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

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

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

מגדירים את הפונקציה באמצעות המילה השמורה function כרגיל ומקפידים לשים סוגריים עגולים. בתוך הסוגריים המסולסלים מתרחש האקשן: מגדירים משתנים, עושים חיבורים ומקבלים תוצאה שאותה מחזירים עם... return. כל הדבר הנהדר הזה שהוא הפונקציה לא יעבוד אם לא תקראו לו. קוראים לו באמצעות קריאה בשם הפונקציה והסוגריים העגולים והכנסת מה שהיא מחזירה לתוך המשתנה foo. במקרה הזה הוא יקבל את מה שהפונקציה מחזירה, שזה 30. החגיגה שמתרחשת בתוך הפונקציה – כלומר הגדרת קבועים ופעולות מתמטיות – מתרחשת אך ורק בתוכה. היא לא זולגת החוצה. זה מה שיפה בפונקציה – אפשר לעשות מה שרוצים בתוכה וזה לא יזלוג החוצה. בדיוק כמו לאס וגאס – מה שקורה בפונקציה נשאר בפונקציה, והדבר היחידי שזולג ממנה הוא ה-return.

את היכולת המופלאה הזו 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. מחזירים את result.

כאמור, מה שקורה בפונקציה נשאר בפונקציה, והפונקציה לא תופעל אם לא תקראו לה ולא תכניסו את מה שהיא מחזירה (כלומר מה שיש מייד אחרי ה-return) למשתנה כלשהו. אך חזיז ורעם! שם המשתנה הוא foo! איך זה יכול להיות? הרי 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 been
declared
```

יקפיץ שגיאה ולא ירוץ. מנוע הג'אווהסקריפט לא יכול להריץ אותו כי אחד מכללי השפה הבסיסיים הוא שאי-אפשר להגדיר משתנה וקבוע בעלי אותו שם. אז איך זה קרה פה? התשובה היא כאמור שמה שנמצא בתוך הסוגריים המסולסלים, כלומר בתוך הפונקציה, מתקיים בממד אחר ולא זולג החוצה. הממד האחר הזה נקרא "לקסיקל סקופ" (Lexical Scope), ולפונקציה יש סקופ משלה, כלומר ממד שבו אפשר להגדיר משתנים שיהיו מבודדים לחלוטין מהסקופ של מי שקורא לפונקציה. נושא הסקופ יחזור בהמשך כשנעסוק בנושאים מתקדמים יותר.

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

scope 1

scope 2

אפשר

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

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

```
let bar = calculateMe();
let answer = calculateMe();
console.log(foo); // 50
console.log(bar); // 50
console.log(answer); // 50
```

מה שקורה פה הוא שמחוץ לפונקציה השתמשתם בדיוק באותם שמות שבהם קראתם לקבועים בתוך הפונקציה, והכול עובד. למה? כי מה שקורה בתוך הסקופ של הפונקציה לא רלוונטי ולא זולג לסקופים אחרים. הסקופ שבו עובדים כרגע נקרא "הסקופ הגלובלי" ואם קוראים לפונקציה מתוכו, דבר לא יזלוג החוצה אליו. בפונקציה אפשר לעשות כל מה שרוצים ולקרוא למשתנים כרצונכם. אמשיך לכתוב על סקופים בהמשך.

שימו לב: בדוגמה קראתי לפונקציה מספר רב של פעמים, ובכל פעם הפונקציה רצה כאילו זו הפעם הראשונה. הפונקציה היא יחידה אוטונומית ועצמאית ויכולה להיקרא מספר רב של פעמים. בכל פעם היא תחזיר למי שקורא לה את מה שיש מייד אחרי ה-return.

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

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

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

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

אם יש כמה return, ה-return הראשון הוא שקובע והקוד שמתבצע לאחריו לא ירוץ לעולם. בדוגמה הזו למשל:

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

הקטע שמייד אחרי ה-return בתוך הפונקציה לא ירוץ לעולם.

שימו לב: אפשר להגדיר כמה פונקציות שרוצים ולקרוא להן איך שרוצים. בכל הדוגמאות הוגדרה פונקציה אחת, אבל אין שום בעיה להגדיר כמה פונקציות שרוצים.

פונקציה עם ארגומנטים

אפשר להעביר לפונקציה ערכים מבחוץ ולקבל אותם בתוך הפונקציה. זה נעשה בשני אופנים. ראשית בהגדרת הפונקציה. זוכרים את הסוגריים העגולים הריקים? עכשיו הם לא יהיו ריקים – אני אכניס לתוכם משתנה. המשתנה הזה מוגדר בתוך הפונקציה כאילו הגדרתי אותו עם let. אפשר להכפיל אותו או לקחת אותו ולהציב אותו איפה שרוצים, בדיוק כמו משתנה רגיל. השם המקובל למשתנה כזה הוא ארגומנט.

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

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

הפונקציה multiply מקבלת ארגומנט בשם arg1. כאמור, ברגע שמכריזים עליו בסוגריים אפשר להשתמש בו בפונקציה עצמה. במקרה הזה לוקחים אותו, מכפילים אותו ומכניסים את התוצאה למשתנה answer.

איך מכניסים אותו בקריאה? פשוט מאוד, מעבירים את הארגומנט בסוגריים. מה שמעבירים נחשב ל-arg1.

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

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

הפונקציה היא אותה פונקציה, אבל התוצאה שלה משתנה בהתאם לארגומנט שמעבירים. כשקוראים לפונקציה כך:

```
multiply(10);
```

אז arg1 בתוך הפונקציה מקבל את הערך 10 והתשובה המוחזרת היא 20, arg1 כפול 2 שנכנס ל-answer ומוחזר עם return.

כשקוראים לפונקציה כך:

```
multiply(20);
```

אז arg1 בתוך הפונקציה מקבל את הערך 20 והתשובה המוחזרת היא 40.

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

שימו לב: בדרך כלל משתמשים בפונקציות כאשר כותבים ג'אווהסקריפט כי זו הדרך הטובה ביותר לפרק את הקוד לחלקים קטנים ולהימנע מחזרות על קוד. חזרות על קוד הן דבר שמוטב להימנע ממנו בקוד כיוון שאם רוצים לשנות אותו, נדרשת עבודה רבה.

אפשר להשתמש בכמה ארגומנטים שרוצים. כך למשל נראית פונקציה עם שלושה ארגומנטים:

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

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

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

אפשר להשתמש בארגומנטים האלו כמשתנים מן המניין בתוך הסוגריים המסולסלים 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 דיפולטיביים

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

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

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

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

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

אפשר להכניס ערך ברירת מחדל (דיפולטיבי) לארגומנטים שייכנס אליהם אם מי שקרא לפונקציה לא העביר לה ארגומנטים. עושים זאת באופן פשוט למדי – כך:

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

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

אם לפונקציה יש כמה ארגומנטים, בג'אווהסקריפט אפשר להכניס לכולם ערך ברירת מחדל. הסתכלו על הדוגמה הזו:

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

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

שימו לב לדוגמה האחרונה: הארגומנט הראשון שמעבירים הוא undefined, שהוא מילה שמורה שהוזכרה בפרק על סוגי מידע. אני בעצם מצהיר שהארגומנט הראשון arg1 הוא undefined ומפעיל את ברירת המחדל. שימו לב שמדובר ב-undefined אמיתי ולא בערך ריק כמו מחרוזת ריקה או אפילו null.

שימו לב: זה נחשב לנוהג רע להכניס יותר מחמישה ארגומנטים לפונקציה. אם בפונקציה שלכם יש יותר מחמישה ארגומנטים סימן שצריך לפצל אותה לשתי פונקציות או יותר.

הפונקציה כאובייקט

פונקציה ניתנת להגדרה גם כך:

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

```
    return answer;
}

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

זה בדיוק אותו דבר כמו להגדיר פונקציה בדרך שלמדנו. שימו לב שהגדרת הפונקציה היא כמו הגדרת סוג מידע פרימיטיבי מסוג 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 לערכים הפרימיטיביים שהכרתם עד כה, פונקציה היא ערך לא פרימיטיבי ומורכב יותר. צריך לזכור שפונקציה בסופו של דבר היא מבנה נתונים שחייבים להכניס למשתנה בדיוק כמו מספר, טקסט, ערך בוליאני וחבריהם. מסיבות היסטוריות, אפשר להגדיר פונקציה בשתי הדרכים, או בשם:

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

או במשתנה:

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

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

Hoisting

כאשר מגדירים פונקציות, המנוע של ג'אווהסקריפט מזיז את כולן למעלה. לשם ההדגמה, מה לפי דעתכם תהיה תוצאת הקוד הזה?

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

משתמשים בפונקציה לפני שהיא מוגדרת, והקוד הזה עובד. למה בדיוק הוא עובד? כי המנוע של ג'אווהסקריפט, 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 משהו כזה:

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

יקבל שגיאה, כי כאמור קוראים לפונקציה לפני שמגדירים אותה.

יש לזה משמעויות כאשר עובדים בקוד מתקדם יותר ולא בהתחלה, כמובן.

closure

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

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

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

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

פה הפונקציה תדע מה הערך של foo ושל bar אף על פי שהם לא מוגדרים בה אלא מחוצה לה. כלומר, foo ו-bar לא מוגדרים בסקופ של הפונקציה אלא בסקופ הגלובלי וודאי שלא מועברים לפונקציה כארגומנטים, ועדיין הפונקציה מכירה אותם. הדיאגרמה הזו אמורה להבהיר את העניין:

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

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

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

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

מובן שה-closure הוא דו-כיווני ושאפשר להשפיע על משתנים בסקופ האב דרך סקופ הבן. למשל:

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

שימו לב שבתוך הפונקציה לא הוגדר משתנה foo באמצעות let. מהרגע שעושים את זה משנים כמובן את המשתנה שנמצא בסקופ האב.

כלומר, כל משתנה שמוגדר מבחוץ חשוף לפונקציה הפנימית. זה כלי 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 לסקופ הגלובלי. ה-let מוגבל יותר.

שימו לב: כרגע זה נראה תיאורטי למדי ואפילו מטופש, אבל יש חשיבות עליונה להבנת ה-closure. בלולאות ובפונקציות רקורסיביות, ובטח ובטח באפליקציות מורכבות יותר, יש הישענות רבה על התכונה הזו. לפיכך כדאי להשקיע זמן בלימוד שלה ועוד אחזור לנושא בהמשך.

פונקציה אנונימית ופונקציית חץ

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

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

שימו לב שיש כאן הגדרת פונקציה ללא שם, באופן די דומה למה שהכרתם: המילה השמורה function, סוגריים פותחים וסוגרים והסוגריים המסולסלים שבתוכם הפונקציה עצמה. בדוגמה הזו הפונקציה מגדירה משתנה a ומדפיסה אותו בקונסולה. אבל הדבר הזה לא ירוץ כי אף אחד לא קורא לפונקציה. איך קוראים לה? מוסיפים סוגריים פותחים וסוגרים מייד בסוף.

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

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

פונקציות אנונימיות, כלומר כאלו ללא שם, משמשות בהמון מקרים ומקומות בג'אווהסקריפט, בין שמריצים אותן מייד לאחר ההפעלה ובין שלא.

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

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

זה בדיוק אותו דבר, אבל מבחינת הסקופ יש לפונקציות חץ כמה יתרונות גדולים וכדאי להשתמש בהן.

נשאלת השאלה, למה להשתמש בכלל בפונקציה אנונימית? ויש לה כמה תשובות.

פונקציה אנונימית כמשתנה

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

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

מה שמתרחש פה הוא שיש שתי פונקציות. פונקציית returnSomething מחזירה 100. פונקציית runMe מקבלת משתנה. מה היא עושה בו? לוקחת אותו ו"מריצה" אותו באמצעות סוגריים פותחים וסוגרים. כלומר היא מניחה שהארגומנט הוא פונקציה. במקרה הזה הוגדרה פונקציה בשם returnSomething והועברה כמשתנה לפונקציית runMe. אבל אפשר להעביר לפונקציית runMe גם פונקציית חץ אנונימית:

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

ואם תריצו את זה תראו הדפסה של מה שהפונקציה האנונימית מחזירה.

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

אפשר להעביר גם ארגומנטים לקולבק. שימו לב לדוגמה הזו:

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

היא זהה כמעט לחלוטין לדוגמה הקודמת, אבל במקרה הזה מצפים שבפונקציה האנונימית שמעבירים יהיה ארגומנט אחד. במקרה הזה מעבירים 2 לארגומנט. אם העניין אבסטרקטי מדי זה בסדר, רק זכרו שקולבקים יכולים לקבל

ארגומנטים שהפונקציה הקוראת להם מעבירה. במקרה הזה פונקציית runMe, שאליה מעבירים ארגומנט שהוא פונקציה (או קולבק), מעבירה 2 כארגומנט לקולבק.

פונקציה אנונימית שמבודדת מהסקופ הגלובלי

פונקציה אנונימית שמריצה את עצמה היא נהדרת בכל מה שקשור לבידוד מהסקופ הגלובלי ולמתן אפשרות למתכנת להגדיר שכבה שרק דרכה אפשר להגיע אל רכיב הקוד שהוא כתב. זה מעט מתקדם מדי נכון לעכשיו, אבל אנסה להסביר בכל מקרה. שימו לב לפונקציה הזו:

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

כאמור, אם זה נשמע תלוש מעט זה בסדר גמור. בשלב הזה של הלימוד טוב לזכור שפונקציה אנונימית שמריצה את עצמה לגיטימית לשימוש בספריות ג'אווהסקריפט שונות (כמו ספריית jQuery) ובכל מודול שהוא, שבו רוצים לממש סקופ נפרד ופרטי. לא נדיר לראות ברשת דוגמאות של פונקציות אנונימיות שמריצות את עצמן.

תרגיל:

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

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

הסבר:

יוצרים פונקציה באמצעות המילה השמורה myFunc וסוגריים עגולים. בסוגריים המסולסלים רואים מה שקורה בתוך הפונקציה. בפונקציה לא קורה כלום והיא מחזירה רק null. מפעילים את הפונקציה באמצעות myFunc() ומכניסים את מה שהיא מחזירה, במקרה הזה null, אל המשתנה foo, שאותו מדפיסים בקונסולה בשורה הבאה.

תרגיל:

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

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

הסבר:

יוצרים פונקציה באמצעות המילה השמורה function וקוראים לה ahlaBahla. מייד אחרי שמה שמים סוגריים עגולים (). בתוך הסוגריים המסולסלים, שחייבים לשים, מתקיימת הפונקציה. במקרה הזה היא לא עושה הרבה אלא רק מחזירה 100. איך היא מחזירה משהו? באמצעות המילה השמורה return, שמחזירה את מה שכתוב אחריה, במקרה הזה 100.

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

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

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

תרגיל:

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

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

הסבר:

יוצרים פונקציה בשם yay עם המילה השמורה function. מייד אחרי ההגדרה יש סוגריים מסולסלים, שבהם מתרחש כל מה שקורה בפונקציה. במקרה הזה, דבר לא מתרחש. הפונקציה מחזירה את מחרוזת הטקסט "yay" באמצעות המילה השמורה return.

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

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

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

תרגיל:

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

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

הסבר:

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

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

כלומר, מגדירים פונקציה ובתוך הסוגריים העגולים מכניסים ארגומנט. עכשיו צריך להוסיף את ההדפסה בקונסולה:

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

הארגומנט הוא משתנה לכל דבר והוא חי בתוך הפונקציה ברגע שקוראים לה. אפשר להדפיס אותו בקונסולה, כמובן, כמו כל משתנה אחר. עכשיו צריך להוסיף לו 1 ולהחזיר את התוצאה. את זה כבר הכרתם:

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

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

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

מעבירים כארגומנט את המספר 1 לפונקציה ומכניסים את מה שהיא מחזירה לתוך המשתנה foo, שאותו מדפיסים.

תרגיל:

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

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

הסבר:

כותבים את הפונקציה כמו כל פונקציה אחרת עם ארגומנט שהיא מקבלת. בחרתי את השם number עבור הארגומנט. הארגומנט הזה הוא משתנה לכל דבר בתוך הסקופ של הפונקציה, ואפשר לכתוב עליו משפטי תנאי כפי שלמדנו בפרקים הקודמים. שימו לב שלפונקציה הזו אין return כיוון שהיא לא מחזירה ערך אלא מדפיסה בקונסולה בלבד. לפיכך גם לא צריך להכניס את מה שהיא מחזירה לתוך משתנה אלא לבצע קריאה בלבד שמפעילה אותה.

תרגיל:

כתבו פונקציה שמקבלת מספר ובודקת אם הוא מספר או סוג מידע אחר. אם הוא סוג מידע אחר היא תדפיס בקונסולה n/a ותסיים 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 האלו בפרקים הקודמים.

```
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 כרגיל, 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 n/a ומבצעים return. ה-return הזה, כפי שלמדנו, יסיים את פעולת הפונקציה סופית, אבל הוא יופעל, כמובן, רק אם התנאי יתממש.

בהנחה שהתנאי לא התממש, ניגשים לבדיקת הזוגי/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. כותבים 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 אמור להיות פונקציה. אם הוא לא פונקציה מופעלת שגיאה. אם הוא אכן פונקציה מריצים אותו ומדפיסים את התוצאה. בדקו פעמיים – עם פונקציה אנונימית שמחזירה מחרוזת טקסט ועם פונקציה אנונימית שמחזירה מספר.

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

הסבר:

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

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

אובייקטים

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

סוג מידע נוסף שאינו פרימיטיבי הוא אובייקט, ומדובר בסוג המידע החשוב ביותר בג'אווהסקריפט. המטרה של אובייקט היא ליצור סוג מידע מורכב שיכול להכיל סוגי מידע אחרים, פרימיטיביים או אפילו אובייקטים ומערכים (שעליהם אדבר בפרק הבא). איך יוצרים אובייקט? הדרך הפשוטה ביותר היא זו:

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

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

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

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

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

יוצרים אובייקט ומכניסים אותו למשתנה myObject. יצירת המפתח נעשית באמצעות החלטה על שם המפתח (במקרה הזה id) והכנסת ערך, בדיוק כמו משתנה. שימו לב לנקודה 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 נעשית 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. בדיוק-בדיוק כמו משתנים! כאן לדוגמה משנים את ה-id כמה פעמים:

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

אפשר לראות שכשמדפיסים את האובייקט, המפתחות והערכים מסודרים באופן הזה:

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 בדיוק כמו ליצור את המפתחות בדרך הקודמת. הכול מאוד-מאוד גמיש וצריך לזכור את זה. הדרך הישירה ליצירת האובייקט – כלומר יצירה שלו כבר עם כל המפתחות מראש – מקובלת יותר וגם עדיפה מבחינת ביצועים.

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

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

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

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

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

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

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

כמפתחי ג'אווהסקריפט מנוסים, אתם לא אמורים להיבהל מהקוד הזה – יש פונקציה שמקבלת שני ארגומנטים, מכניסה אותם לאובייקט ומחזירה אותו. זה משהו שאתם אמורים להכיר אחרי הפרק הקודם על פונקציות. את הקוד הזה תראו חוזר כמה וכמה פעמים במערכות מבוססות ג'אווהסקריפט, עד כדי כך שהכניסו לג'אווהסקריפט את היכולת ליצור מפתחות וערכים באופן אוטומטי לפי שמות הערכים. כך הפונקציה שלעיל תהיה שקולה 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 הזו נקראת "יצירה מקוצרת" והיא הולכת ותופסת תאוצה בשנים האחרונות. לא אשתמש בה בדוגמאות, אבל אנא זכרו שהיא קיימת כי נעשה בה שימוש רב, בעיקר על ידי מתכנתים מנוסים (כולל עבדכם הנאמן).

כאמור, אובייקט יכול להכיל כל נתון שהוא בתור ערך, כולל... אובייקטים אחרים! שימו לב לדוגמה הזו למשל:

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

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

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

אפשר לגשת אל מפתח גם באמצעות סוגריים 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 בתוך משתנה מסוג 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 בסוגריים מרובעים, ובתוכם שמים את המשתנה שמכיל את שם המפתח.

אם תנסו לעשות משהו כזה (propertyName הוא המשתנה שמכיל את שם המפתח) תקבלו undefined.

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

לפיכך לעולם לא משתמשים בנקודה אם שם המפתח נמצא בתוך משתנה, אלא בסוגריים מרובעים.

מחיקת מפתח

מחיקת המפתח נעשית באמצעות האופרטור delete:

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

אפשר להשתמש, כמובן, בנקודה או בסוגריים 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: ƒ () }
myObject.alertMe(); // hi!
```

כאן יוצרים אובייקט כפי שהיה קודם. משתמשים במפתח alertMe ומכניסים לתוכו פונקציה שכל מה שהיא עושה הוא להדפיס בקונסולה את מחרוזת הטקסט "hi!". יצירת הפונקציה זהה במאה אחוז ליצירת פונקציה רגילה והכנסתה לתוך משתנה. הקריאה לפונקציה בתוך אובייקט גם כן זהה לחלוטין לקריאה לפונקציה ונעשית כפי שקוראים לכל ערך אחר.

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

אפשר להכניס פונקציה גם ככתיב מקוצר (פונקציית חץ, שעליה למדנו בפרקים הקודמים) באופן הבא:

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

אובייקט כקבוע

אחד הדברים החשובים שצריך לזכור הוא שאם מגדירים אובייקט כקבוע, אין שום בעיה לשנות אותו. נכון, אי-אפשר לשנות את הסוג שלו (זאת אומרת, לקבוע שמעכשיו במקום אובייקט יכיל הקבוע מספר), אבל אפשר להוסיף לו תכונות או לשנות תכונות קיימות, כלומר לשנות את השדות בתוך האובייקט ולא לעשות השמה לאובייקט אחר. מדובר בהבדל משמעותי:

```
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 הערכים 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 הכי מבלבלים הוא עניין ההצבעה, שאינו כל כך מורכב כפי שחושבים. כשכתבתי על משתנים פרימיטיביים, ציינתי שאפשר להעתיק את המשתנה למשתנה אחר. אם משנים את המשתנה האחר, המשתנה המקורי נשאר כשהיה. הינה דוגמה:

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

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

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

שינוי באובייקט, שאמור להיות במשתנה השני, משפיע על האובייקט המקורי! זה קורה כי כשמעתיקים אובייקט שנמצא במשתנה objectA למשתנה object לא מבצעים העתקה אלא הפניה, וההבדל ביניהן משמעותי. הסיבה לכך היא שיקולי זיכרון. אם רוצים לבצע העתקה של אובייקט ולשנות אותו בלי שהוא ישפיע על האובייקט המקורי, צריך לעשות הליך שנקרא clone ואפשרי לביצוע באמצעות לולאות, ועל כך בפרקים הבאים.

תרגיל:

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

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

הסבר:

יוצרים משתנה ומכניסים אליו אובייקט, באובייקט יש שלושה מפתחות: id, name ו-price. כל אחד מהם קיבל ערך. ההגדרה של האובייקט נעשית באמצעות סוגריים מסולסלים, 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}
```

בעצם יוצרים כאן אובייקט ריק ואז מכניסים את המפתחות שלו בזה אחר זה. הקריאה למפתח נעשית כמו משתנה – שם האובייקט, נקודה ואז שם המפתח.

תרגיל:

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

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

תשובה:

```
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 שלמדנו. אם המספר הוא לא בין 1 ל–7 מדפיסים הודעת שגיאה ומחזירים אובייקט ריק. אם המספר הוא בין 1 ל–7 מוצאים את היום ומכניסים אותו למשתנה dayName. את dayNumber ואת dayName מכניסים לאובייקט ומחזירים אותו. זה הכול.

כיוון ששמות 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 מכניסה את שם המפתח והערך לאובייקט ולאחר מכן מחזירה את האובייקט:

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

הסבר:

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

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

אלא צריך להשתמש בסוגריים מרובעים להגדרה. כל שנותר לעשות אחרי הוספת התכונה והערך הוא להחזיר את האובייקט ולבדוק. זה הכול.

תרגיל:

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

```
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 המקבלת ארגומנט אחד ומדפיסה אותו, ומפתח returnObject שהוא פונקציה שמחזירה אובייקט. זה הכול. אחר כך כל מה שנותר לעשות הוא לקרוא לפונקציות באמצעות המפתחות.

מערכים

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

זה דבר שתמיד מבלבל מתכנתים. כשאדבר 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 יצרתי 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 היצירה של המערך ולגשת אליהם בכל שלב:

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

כאן למשל יוצרים מערך וכבר בשלב היצירה מכניסים לתוכו שלושה איברים לפי הסדר - למיקום 0, למיקום 1 ולמיקום 2. אחרי היצירה מדפיסים בקונסולה את האיבר שבמקום הראשון (מיקום 0), משנים אותו ומדפיסים אותו שוב. שימו לב: פנייה לקבלת ערך מתוך מערך נעשית על ידי סוגריים מרובעים וגם האינדקס של האיבר.

אפשר כמובן להכניס גם אובייקטים לתוך מערכים וזה אפילו מקובל:

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

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

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

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

מערך בעצם מממש מבנה נתונים שידוע בתכנות כ"מחסנית". כמו שיש מחסנית ובה כדורים, כך יש מחסנית של נתונים. הכנסה של איבר חדש נקראת "דחיפה", ואפשר לממש אותה באמצעות שימוש בפונקציה מיוחדת לסוג המידע מערך בלבד, שנקראת "push":

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

push היא פונקציה מיוחדת שעובדת אך ורק על סוגי מידע שהם מערכים, כלומר כאלו שמחזירים array באמצעות האופרטור של typeof. כשלמדנו על סוגי מידע, הראיתי כל מיני פעולות שאפשר לעשות על מחרוזות טקסט או על מספרים; ובכן, push היא פעולה שאפשר לעשות אך ורק על מערך. מכניסים לתוכה כארגומנט את המידע שרוצים להכניס למערך בתור האיבר החדש והאחרון – בין שמדובר במספר ובין שבמחרוזת טקסט, באובייקט, במערך וכו'. כל דבר ייכנס כארגומנט ב-push, וייווצר איבר חדש בסוף.

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

משיכה היא בדיוק ההפך מדחיפה – היא מאפשרת למשוך את האיבר האחרון מהמערך ולקבל אותו כמשתנה. המשיכה נעשית באמצעות פקודת 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 שינוי (מוטציה), או נכון יותר המחסנית, מתקצר באיבר אחד.

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

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

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

ולבסוף, שליפת האיבר הראשון במערך נעשית באמצעות פונקציית shift:

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

בדוגמה הזו רואים מערך שבו האיבר הראשון הוא מחרוזת טקסט a והאיבר האחרון הוא מחרוזת טקסט c. שימוש בפונקציה shift מחזיר את האיבר הראשון. כרגע במערך האיבר הראשון הוא b והאיבר האחרון הוא עדיין c. המערך קוצר משלושה איברים לשניים.

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

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

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

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

כך אפשר לראות את אורך המערך. שימו לב שמדובר במשהו טריקי. אם למשל יוצרים מערך ומכניסים אליו שני ערכים בלבד – אחד במקום ה-0 (שהוא המקום הראשון, אנו זוכרים שהמקום הראשון במערך הוא 0) והשני במקום ה-99 – האורך של המערך יהיה 100:

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

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

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

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

למה פחות 1? כי המערך מתחיל מ-0. האיבר הראשון תמיד יהיה במיקום 0.

כמו באובייקט, מחיקת איבר במערך תתבצע בעזרת האופרטור delete שבעצם לוקח את האיבר ומכניס אליו undefined:

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

שימו לב שכשרוצים למחוק את הערך השלישי, צריך לציין [2] כיוון שהמערך מתחיל תמיד מ-0.

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

כמו המתודות הקודמות שלמדנו עליהן, גם המתודה splice שייכת למערך בלבד. היא מקבלת כמה ארגומנטים. הראשון הוא מאיזה מקום במערך להתחיל והשני הוא כמה איברים למחוק.

נניח שיש מערך של אבות ורוצים להוריד את משה, כי הוא לא אחד האבות המקראיים.

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

אם מורידים את משה באמצעות delete, יהיה undefined במערך וכביכול יהיו "ארבעה אבות" והשיר "אחד מי יודע" ישתבש לגמרי. אז מה עושים? משתמשים ב-splice. מה המקום של האיבר שרוצים למחוק (משה)? המקום ה-2; 0 זה אברהם, 1 זה יצחק ו-2 משה. כמה איברים רוצים למחוק? אחד. איך תיראה המחיקה? כך:

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

מערכים ומחרוזות טקסט

כדי להגדיל את השמחה ואת הבלבול הכללי, בג'אווהסקריפט מחרוזות טקסט חולקות לא מעט תכונות עם מערכים והן מתנהגות בחלק מהמקרים כמו מערכים שבהם האיבר הראשון הוא האות הראשונה, האיבר השני הוא האות השנייה וכך הלאה. שימו לב למשל לדוגמה הזו:

```
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 האחרונה בדיוק כפי ששולפים את האיבר האחרון במערך. האופרטורים delete וגם push ו-pop לא יעבדו, אך כל שאר האופרטורים כן. כרגע אין לזה שימוש אופרטיבי, אבל כדאי לזכור את זה.

תרגיל:

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

```
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 סוגריים מרובעים ואז כל איברי המערך מופרדים בפסיק. האיברים יכולים להיות כל נתון שהוא.

תרגיל:

למערך הקודם שיצרתם, הוסיפו את המספר 11 בסוף.

```
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 לדחוף ערך בתור האיבר האחרון במערך. בתוך הסוגריים העגולים שלאחר ה-push מכניסים את הערך שרוצים שיהיה במערך.

תרגיל:

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

```
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:

הדחיפה של ערך כאיבר הראשון של המערך וההזזה של כל האיברים נעשות באמצעות הפונקציה unshift, שכמו push מקבלת את הערך שרוצים שיהיה ראשון במערך, במקרה הזה 0.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר הראשון:

```
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, שייחודית למערך, מסירה את האיבר הראשון ומזיזה את כל שאר האיברים.

תרגיל:

במערך הראשון שיצרתם, הורידו את האיבר החמישי:

```
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 הזה רציתם למחוק את האיבר החמישי ולפיכך הייתם צריכים לכתוב delete myArray[4]; למה 4? כיוון שהמערך מתחיל ב-0 והאיבר החמישי יהיה

```
myArray[4];
```

new ו-this

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

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

ראשית מגדירים פונקציה פשוטה כמו כל פונקציה אחרת. עושים את זה באמצעות הכנסתה למשתנה. את הפונקציה הזו מאתחלים באמצעות new. שימו לב שממש מאתחלים את הפונקציה. ברגע שמשתמשים 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 objMaker, כמו פונקציה רגילה.

יצירת אובייקט ריק פשוטה מאוד, אבל הגדולה של הפונקציה הזו היא ביצירת אובייקט בעל תכונות. יצירת אובייקט בעל תכונות נעשית באמצעות המילה השמורה this. בתוך פונקציה בנאית המשמעות של this היא האובייקט שייווצר כתוצאה מהפונקציה. this מצביע לאובייקט – כלומר הוא מפנה אל האובייקט הזה. אם למשל רוצים להוסיף תכונה בשם foo, עושים משהו כזה:

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

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

```
let clientObjMaker = function () {
    this.userFirstName = 'Moshe';
    this.userLastName = 'Cohen';
    this.userCity = 'Holon';
    this.userCar = 'Subaru';
};
let mosheObject = new clientObjMaker();
console.log(mosheObject);
```

כאן יוצרים אובייקט באמצעות פונקציה בנאית. הפונקציה הבנאית הכריזה על שם פרטי, על שם משפחה ועל תכונות נוספות של הלקוח. אם תציצו אל האובייקט באמצעות הקונסולה, תראו שהוא מכיל את כל הארגומנטים שהגדרתם באמצעות ה-this:

```
Object {
  userCar: "Subaru",
```

```
  userCity: "Holon",
  userFirstName: "Moshe",
  userLastName: "Cohen"
}
```

משתמשים בדרך כלל בפונקציות בונות על מנת ליצור אובייקטים קבועים. למשל, אם רוצים לייצר אובייקטי לקוח אחידים, יוצרים פונקציה בונה אחידה לאובייקט לקוח 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 הזו 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 לפעולה הבונה גם הגדרות של פונקציה שהן חלק אינהרנטי מהאובייקט שהפעולה הבונה יוצרת. לפעולות אלו קוראים "מתודות" (וביחיד "מתודה"). מדובר בפונקציה שמוצמדת לאובייקט ואפשר להפעילה 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 התכונות, אבל היא פונקציה ולא מחרוזת טקסט. במקרה הזה היא מחזירה את השם המלא של המשתמש המורכב מתכונת השם הפרטי, שם המשפחה ורווח ביניהם.

אפשר להגדיר משתנים פרטיים בתוך הפונקציה הבנאית – פרטיים במובן שאי-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 בתוך הפונקציה הבנאית. בניגוד לחבריו, המשתנה הזה לא מוצמד ל-this אלא מוגדר ממש בתוך פונקציה. האם אפשר לגשת אליו מרחוק? לא. רק מה שמצמידים 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 doesnt 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, כיוון שהיא פונקציה שמשמשת לקבלה (באנגלית get) של משתנים פרטיים שאין גישה אחרת אליהם. כמו שיש get יש 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 functuon
        id = newId;
    }
};
let mosheObject = new clientObjMaker('Moshe', 'Cohen', 'Holon', 'Subaru');
mosheObject.setId('246810')
let id = mosheObject.getId();
console.log(id); // 246810
```

נשאלת השאלה, למה בדיוק צריך set ו-get אם אפשר פשוט לחשוף את ה-id ב-this כמו שאר המשתנים? התשובה נעוצה במימוש. לפעמים רוצים לעשות ולידציה. למשל, רוצים לוודא ש-id הוא תמיד מספר. הדרך הכי טובה לעשות זאת היא לאכוף את זה באמצעות set:

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

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

בפונקציית set בודקים באמצעות האופרטור typeof את סוג הארגומנט שהועבר. אם הוא מסוג מספר מכניסים אותו ל-id, ומעכשיו המשתנה id מכיל את הערך הזה. אם לא, מחזירים שגיאה ולא משנים את ה-id הקיים.

אז מה ההבדל בין הגדרה רגילה של אובייקט לפונקציה בנאית? בדרך כלל משתמשים בפונקציה בנאית על מנת להגדיר אובייקטים שיש להם תפקיד – יחידת תוכנה שיש לה תכונות ומתודות. אדבר על כך בהמשך. בדרך כלל קוד

מודרני של ג'אווהסקריפט ארוז באובייקטים שמבוססים על פונקציה בנאית. ומה עם אובייקטים רגילים? בהם משתמשים בדרך כלל לאחסון מידע בלבד.

בהמשך תראו את התועלת שיש בפונקציה בנאית כאשר מפתחים מערכות מורכבות יותר. בינתיים, זכרו שברגע שאתם רואים new, זה סימן שתקבלו אובייקט חדש. נדבר על פונקציות בנאיות בפרק על אובייקטים מובנים בג'אווהסקריפט.

תרגיל:

צרו פונקציה בנאית לאובייקט מכונית המקבלת 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 שלה יהיה opelwhite1200. צרו לאובייקט מכונית פונקציה המחזירה את מספר הזיהוי. הדגם, הצבע ונפח המנוע הם נתונים שחשופים החוצה.

```
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:

יוצרים 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 הפרטי.

תרגיל:

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

```
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 ש-modelNumberמוגדר כמשתנה.

תבנית טקסט

תבנית טקסט מאפשרת ליצור מחרוזות טקסט בקלות ממשתנים שונים. למדנו בפרק על מחרוזות טקסט שאפשר לחבר בין מחרוזות טקסט. כלומר אם קיימים משתנה שיש בו "Hello" ומשתנה שיש בו "World", יתקבל משהו כזה אם תחברו אותם:

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

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

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

אם לדוגמה יש מספר ורוצים להצמיד אליו את התו $ (כמו במחירים) צריך לעשות משהו כזה:

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

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

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

הבעיה היא שזה מסורבל מאוד, וככל שחלף הזמן ותוכנות הג'אווהסקריפט התפתחו 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 האלו מסורבלים מאוד. אבל מאוד. לפיכך נוצרה דרך להגדיר "תבנית" בג'אווהסקריפט או, נכון יותר, דרך ליצור מחרוזת טקסט מורכבת בלי כל אופרטורי החיבור האלו. איך עושים את זה? יש גרש מסולסל (באנגלית backtick), שנמצא במקלדות סטנדרטיות משמאל לספרה 1 ומעל ה-Tab. הוא נראה כך: `

מדובר בגרש שעובד בדיוק כמו גרש רגיל ' או גרשיים כפולים " בכל מה שנוגע לטקסט. כלומר, משהו כזה:

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

בהחלט יעבוד, ו-myVar ייחשב למחרוזת טקסט לגיטימית לכל דבר. אבל לגרש המסולסל יש יכולת שאין לגרשיים הרגילים, והיא להכיל תבנית. אם רוצים להכניס משתנה מסוים לתוך מחרוזת הטקסט צריך להקיף אותו בדולר ($) ובסוגריים מסולסלים, והוא ייכנס למחרוזת הטקסט בשלמותו. הינה דוגמה:

```
const user = {
    name: 'Ran',
    localtime: 'Morning',
};
let welcomeString = `Hello, ${user.name} How are you doing? Good
${user.localtime}!`;
console.log(welcomeString); // "Hello, Ran. How are you doing? Good Morning!"
```

כלומר, הדבר 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.

תבניות מטפלות באופן נאה מאוד בריבוי שורות, ואילו מחרוזות טקסט רגילות לא מסוגלות להתמודד עם ריבוי שורות. אם נוסיף ירידת שורה בדוגמה שלעיל, הדוגמה תראה כך:

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

מה עדיף? אני חושב שהתשובה ברורה.

בשנים האחרונות יש מעבר חד-משמעי אל שימוש בתבניות טקסט בג'אווהסקריפט, ואני ממליץ גם לכם להשתמש בהן.

תרגיל:

נתון מערך שיש בו בוקר וערב:

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

הדפיסו באמצעות תבנית טקסט את המשפט "Good Morning" ללא שימוש באופרטור חיבור.

תשובה:

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

הסבר:

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

```
timeOfDay[0]
```

מקיפים אותו בסוגריים מסולסלים שבתחילתם יש $:

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

ומשבצים אותו בתוך מחרוזת הטקסט המוקפת בגרש מסולסל מכל צד. הערך של הביטוי ייכנס לתוך מחרוזת הטקסט.

תרגיל:

כתבו פונקציה שמקבלת שני ארגומנטים, מספר וסמל מטבע, ומחזירה מחרוזת טקסט של שני הארגומנטים מחוברים. למשל, אם מעבירים 30 ו-₪ הפונקציה תחזיר 30₪.

תשובה:

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

הסבר:

כותבים פונקציה רגילה שמקבלת שני ארגומנטים, amount ו-currency. שני הארגומנטים האלו נכנסים למחרוזת טקסט אחת שמוקפת בגרש מסולסל ` מכל צד. הגרש הזה מצביע על הימצאותה של תבנית טקסט שכל משתנה שנמצא בתוכה ומוקף בסוגריים מסולסלים וב-$ – ערכו ייכנס למחרוזת הטקסט. מחרוזת הטקסט פה היא פשוטה:

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

כלומר, מחרוזת הטקסט היא המספר והערך צמודים. מחזירים את מחרוזת הטקסט הזו. על מנת לבדוק אותה בודקים את הפונקציה – קוראים לה עם שני ארגומנטים, 30 וסמל המטבע של השקל (מוקף בגרשיים כיוון שהוא מחרוזת טקסט), ובודקים את התוצאה. ההדפסה תראה שצדקתם.

לולאות

לולאות מאפשרות לעבור על האיברים של מערך או של אובייקט בקלות רבה. יש לא מעט לולאות בג'אווהסקריפט. חלקן מיועדות למערכים וחלקן לאובייקטים, וכל אחת מהן משמשת למטרה אחרת עם יתרונות וחסרונות משלה. לשם הבהרה – אני משתמש במונח "לולאה" לכל איטרציה שהיא.

לולאת for

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

איך עושים את זה? כך:

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

קטע הקוד הזה נקרא "לולאת for" והוא ירוץ עשר פעמים בדיוק, מ-0 ועד 9. הבה ננתח אותו: ביטוי ה-for מורכב מהמילה השמורה for ומסוגריים עגולים שבתוכם ביטוי ה-for, המורכב משלושה חלקים:

```
        1          2     3
for (let i = 0; i < 10; i++) {
    console.log('Iteration number ' + i);
}
```

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

החלק השני הוא עד מתי תרוץ הלולאה. קובעים שהיא תרוץ כל עוד i קטן מעשר. אפשר להשתמש פה בכל אופרטור השוואתי.

החלק השלישי הוא מה שקורה למונה בכל הפעלה של הלולאה. במקרה הזה משתמשים באופרטור שמוסיף 1. כלומר בכל פעם שהלולאה רצה, i יעלה ב-1.

בתוך הסוגריים המסולסלים נמצא מה שקורה בכל פעם שהלולאה רצה.

הבה נראה מה קורה בריצה הראשונה:

ריצה ראשונה

i = 0 עומד בתנאי מעלים את i באחד

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

i שווה ל-0. נעשית בדיקה אם i קטן מ-10. כיוון שהוא קטן מ-10, מה שיש בתוך הלולאה עובד, הקונסולה מדפיסה את i ומעלים את i ב-1. מגיעים לריצה הבאה:

ריצה שניה

i = 1 עומד בתנאי מעלים את i באחד

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

פה כבר i שווה ל-1. נכון, כתוב פה let i = 0, אבל זה תקף אך ורק לריצה הראשונה של הלולאה. בריצה השנייה הלולאה תזכור שלא צריך לאתחל את i אלא רק להעלות אותו. כיוון ש-1 קטן מ-10 מריצים את מה שיש בתוך הסוגריים המסולסלים ואחרי כן מעלים את i ב-1. בריצה הבאה הוא יהיה שווה ל-2, ונמשיך לריצה הבאה.

ריצה שלישית

i = 2 עומד בתנאי מעלים את i באחד

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

פה i שווה ל-2, כיוון שהועלה ב-1 בפעם הקודמת. 2 קטן מ-10 והתנאי עדיין תקף. הלולאה מריצה את מה שיש בתוך הסוגריים ומעלה את i ב-1. בריצה הבאה הוא יהיה שווה ל-3 וכך הלאה. עד שנגיע לריצה האחרונה:

ריצה עשירית ואחרונה

i = 9 עומד בתנאי מעלים את i באחד

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

הריצה האחרונה היא הריצה העשירית. i כבר שווה ל-9. מריצים את מה שקורה בתוך הסוגריים המסולסלים ומעלים את i ב-1. עכשיו הוא שווה ל-10. בריצה הבאה התנאי לא מתקיים, 10 לא קטן מ-10 והלולאה נעצרת.

שימו לב: כל ריצה או סיבוב של הלולאה נקראים "איטרציה". זה המונח הסטנדרטי.

מובן שאפשר לעשות לולאה שיורדת. שימו לב:

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

אם מריצים את הלולאה הזו מקבלים את ההדפסות הבאות בקונסולה:

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

שלושת חלקי הלולאה הם:

1. אתחול מונה הלולאה והצבתו 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:

באיטרציה הראשונה מאתחלים את i וקובעים אותו על 5. האם הוא עומד בתנאי? כן. 5 גדול מ-0. מדפיסים את מה שיש בתוך הסוגריים המסולסלים ומורידים את i ב-1 באמצעות האופרטור --.

עוברים לאיטרציה השנייה:

ריצה שניה

באיטרציה הבאה
יהיה שווה ל 3

i = 4 עומד בתנאי

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

באיטרציה השנייה i כבר שווה ל-4. הוא עומד בתנאי, 4 גדול מ-0, לפיכך מה שכתוב בתוך הסוגריים המסולסלים קורה (יש הדפסה בקונסולה) ו-i מופחת ב-1. הלאה!

ריצה שלישית

באיטרציה הבאה
יהיה שווה ל 2

i =3 עומד בתנאי

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

באיטרציה השלישית i שווה ל-3, עדיין יש עמידה בתנאי. מריצים את מה שקורה בתוך הסוגריים המסולסלים ואז מורידים את i ב-1.

ריצה רביעית

באיטרציה הבאה
יהיה שווה ל 1

i = 2 עומד בתנאי

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

באיטרציה הרביעית i שווה ל-2. 2 עדיין גדול מ-0. מתרחשת הרצה נוספת של מה שיש בסוגריים המסולסלים והפחתה של 1 מ-i.

ריצה חמישית
ואחרונה

באיטרציה הבאה
יהיה שווה ל 0

i = 1 עומד בתנאי

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

באיטרציה החמישית והאחרונה i שווה ל-1. 1 גדול מ-0 ולפיכך מה שיש בסוגריים המסולסלים ירוץ. i יורד ב-1 ובריצה הבאה יהיה 0.

באיטרציה שלא תתקיים יש בדיקה אם i גדול מ-0. כיוון ש-i שווה ל-0, התנאי לא מתקיים. 0 לא גדול מ-0 ולפיכך האיטרציה לא תרוץ. הלולאה נעצרת:

איטרציה
שלא תתקיים

i = 0 לא עומד בתנאי

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

שימו לב: מקובל מאוד לקרוא למשתנה של הלולאה בשם i, j או index.

אמנם לולאות של הפחתה או הוספה הן הלולאות הקלאסיות, אבל אפשר באמת להתפרע. הינה למשל לולאה 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 לכתוב את הלולאה על נייר ולנסות להבין למה הפלט הוא כזה. הבה נתחיל. באיטרציה הראשונה, i מן הסתם הוגדר כך שיהיה שווה ל-0. כיוון ש-0 קטן מ-12, ההדפסה בקונסולה מתרחשת ואז מעלים את i ב-4.

איטרציה מספר 1

i = 0 ↑ ↑ i = 4 בריצה הבאה

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

באיטרציה השנייה, i שווה ל-4. מן הסתם 4 קטן מ-12 ומה שיש בתוך הסוגריים המסולסלים יקרה. i יעלה בעוד 4.

איטרציה מספר 2

i = 4 ↑ ↑ i = 8 בריצה הבאה

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

באיטרציה השלישית, i שווה ל-8. 8 עדיין קטן מ-12 והפעולה בלולאה תתקיים. i יעלה ב-4, ל-12.

איטרציה מספר 3

i = 8 ↑ ↑ i = 12 בריצה הבאה

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

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

איטרציה מספר 4 ואחרונה

בפעם הבאה, שלא תרוץ
i = 16

i = 12

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

דיברתי על סקופים בפרק על הפונקציות. גם בלולאות יש סקופ. כל מה שקורה בתוך הסוגריים המסולסלים נחשב לסקופ משלו. זו גם אחת החוזקות של let לעומת ה-var הישן יותר, שהיה נהוג בעבר. let שומר על סקופ משלו בלולאה. אם תנסו 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 ל-i מבחוץ, וזה מלכלך את הסקופ הכללי בצורה שלא תיאמן.

```
for (var i = 0; i <= 12; i = i + 4) {
    console.log('Iteration number ' + i);
}
console.log(i); //16
```

אם אתם רוצים לשמור על הסקופ הכללי שלכם נקי – ואתם רוצים, תאמינו לי שאתם רוצים – אל תשתמשו ב-var. יש בתקן החדש let, הבה נשתמש בו. צריך לזכור שהסקופ עובד גם פה, ול-for יש סקופ משלו שמתפוגג בכל איטרציה.

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

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

אבל זה ארוך ומעצבן, במיוחד אם יש הרבה איברים. במקום זה אפשר לכתוב לולאת for פשוטה:

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

```
    console.log(myArray[i]);
}
```

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

הבה נבחן את האיטרציות:

איטרציה מספר 1

באיטרציה הראשונה i מאותחל ושווה ל-0. מן הסתם 0 קטן מ-4, ומה שיש בתוך הסוגריים המסולסלים מתקיים.

כיוון ש-i=0 אפשר להשתמש בו. זה בדיוק כמו לכתוב:

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

פשוט במקרה הזה i כבר קיים. לא צריך להגדיר אותו. מה שיודפס בקונסולה הוא first. עוברים לאיטרציה הבאה.

איטרציה מספר 2

באיטרציה הבאה i כבר שווה ל-1. התנאי מתקיים כי 1 קטן מ-4, ומה שיש בתוך הסוגריים המסולסלים רץ. אפשר להדפיס את האיבר הראשון (שהוא האיבר השני, כי במערכים מתחילים מ-0). עוברים לאיטרציה הבאה, שבה i עולה ל-2.

איטרציה מספר 3

גם פה, כמו באיטרציות הקודמות, התנאי מתקיים. 2 קטן מ-4. אפשר להשתמש ב-i, שכרגע הוא 2, להדפיס את האיבר השלישי במערך ולעבור לאיטרציה הבאה, שבה i עולה שוב, הפעם ל-3.

איטרציה מספר 4

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

הבעיה בגישה הזו היא שמניחים שגודל המערך הוא 4. אבל מה אם לא יודעים מה גודל המערך? זוכרים שאפשר לחלץ את גודל המערך באמצעות שימוש בתכונה המיוחדת למערך length? אפשר להשתמש בה בלולאה:

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

myArray.length הוא 4, בדיוק 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 הזו היא שהיא לא תעצור לעולם. למה? כי היא תתקיים כל עוד i גדול מ-0, ו-i גדל ב-1 כל הזמן ולעולם לא יהיה קטן מ-0. הלולאה הזו, שלא תסתיים לעולם, נקראת לולאה אינסופית. בדרך כלל, דברים כאלו יקרסו אחרי המון איטרציות. אם מריצים את זה בדפדפן, מתישהו הלשונית תהיה לא יציבה ותקרוס. כשכותבים לולאה (ולא משנה מאיזה סוג) והדפדפן קופא, סביר להניח שזאת הסיבה.

לולאת while

לולאה נוספת היא לולאת while. היא פשוטה יותר מלולאת for ומכילה תנאי אחד שכל עוד הוא נכון הלולאה מתקיימת. למשל:

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

כל עוד התנאי מתקיים, הלולאה רצה.

כל עוד התנאי הזה שווה ל true
הלולאה תרוץ

```
let i = 0;

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

בלולאות כאלו, כמובן, חייבים להגדיר את i מחוץ ללולאה, אחרת בכל ריצה של הלולאה יתאפס משתנה ה-i.

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

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

היא עובדת כל עוד n חיובי ו-i קטן מ-100. ברגע שאחד משניהם לא נכון, היא תעצור. קצת מורכב ומבלבל, אבל כדאי לזכור את זה. אחד השימושים שלה הוא אם (מסיבה מסוימת) רוצים ליצור לולאה אינסופית, ואז כותבים משהו כזה:

```
while (true) {
}
```

אבל צריכה להיות לכם סיבה טובה מאוד ליצור לולאה אינסופית.

לולאת do while

עוד סוג של לולאת while הוא לולאת do while. סוג הלולאה הזה מאוד לא שכיח אבל כדאי להכיר אותו. הלולאה הזו ייחודית כי בניגוד ללולאת for וללולאת while, היא תמיד מבצעת פעולה אחת ואז בודקת את התנאי, במקום לבדוק את התנאי ואז לרוץ. כלומר, תמיד הלולאה רצה לפחות פעם אחת.

הינה דוגמה:

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

יש כאן מילה שמורה, do, ומייד אחריה סוגריים מסולסלים. מה שיש בתוכם תמיד יקרה, ורק אחר כך יגיע ה-while שבודק את התנאי. כאן, למשל, הלולאה תתקיים כל עוד i הוא שלילי. לפני התנאי קובעים את i על 100. לפי כל הלולאות שלמדנו זה אומר שהלולאה לא תרוץ לעולם, אבל כיוון שמשתמשים ב-do while, הלולאה תרוץ לפחות פעם אחת.

לולאת forEach

הלולאות שלמדנו עד כה הן לולאות שעובדות עם תנאים, אבל בעולם הג'אווהסקריפט מקובלות יותר לולאות שעובדות עם אובייקטים או עם מערכים באופן שלם יותר. למה צריך את זה? באחד התרגילים בפרק על מערכים ראיתם 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 דרכים אלגנטיות יותר לעבור על כל האיברים בו. הדרך הראשונה והידועה מכולן היא שימוש ב-forEach. מדובר במתודה שקיימת אך ורק במבנה נתונים מסוג מערך (לא באובייקט) ומקבלת פונקציה אנונימית עם שני ארגומנטים – value שהוא האיבר של המערך ו-key שהוא המספר של המפתח. איך זה עובד? ככה:

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

```
});
// "value: first, key: 0"
// "value: second, key: 1"
// "value: third, key: 2"
```

יש כאן מערך בשם myArray ובו שלושה איברים. האיבר במיקום 0 הוא מחרוזת טקסט first. האיבר במיקום 1 הוא מחרוזת טקסט second והאיבר האחרון, במיקום 2, הוא מחרוזת טקסט third. כיוון שמדובר במערך, אפשר להצמיד לו את מתודת forEach:

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

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

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

זה כמו לכתוב משהו כזה:

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

הקוד שלעיל זהה כמעט לחלוטין לקוד של פונקציה אנונימית, אבל פה יוצרים עוד פונקציה ו"מלכלכים" את הסקופ. מקובל מאוד להשתמש בפונקציה אנונימית וזה מה שעושים כאן. הפונקציה האנונימית רצה על כל איבר. מאיפה באים הארגומנטים value ו-key בלי שקראו להם? זה הכוח של לולאת forEach, היא שותלת אותם שם.

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

הבה נבחן את ההרצה. בפעם הראשונה, האיבר הראשון מגיע לפונקציה:

מה שיש בפונקציה מופעל. זה הכוח של forEach. אל תתבלבלו מהסינטקס המוזר עם החץ, בסופו של דבר זו פונקציה. מה שיש בתוך הסוגריים המסולסלים רץ ורק הארגומנטים מתחלפים. בריצה הראשונה של forEach, תור האיבר הראשון להיכנס לפונקציה האנונימית. יש value שהוא האיבר של המערך ויש key שהוא המקום של האיבר במערך. למה 0? כי מערך תמיד מתחיל מ-0.

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

הפונקציה רצה 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 הזה - מחרוזת טקסט. ה-key הוא המקום של האיבר השני במערך. למה 1? כי מערכים מתחילים מ-0.

בפעם השלישית והאחרונה, האיבר השלישי ייכנס לפונקציה האנונימית:

שמות הארגומנטים כמובן נתונים לבחירה. גם אם תקראו להם arg1 ו-arg2 הם יעבדו:

```
let myArray = ['first', 'second', 'third'];
myArray.forEach((arg1, arg2) => {
    console.log(`value: ${arg1}, key: ${arg2}`);
});
// "value: first, key: 0"
```

```
// "value: second, key: 1"
// "value: third, key: 2"
```

מה החיסרון של הלולאה? עם forEach אי-אפשר לשבור את הלולאה. היא תרוץ לכל אורך איברי המערך.

למה רוצים לשבור את הלולאה? בגלל אלף ואחת סיבות. בדרך כלל אם מחפשים משהו ומוצאים אותו, אין צורך בהרצת כל הלולאה. בדיוק בשביל זה יש את לולאת for of.

לולאת for of

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

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

לא צריך להיבהל. מדובר במערך שבמקום 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 שתעבור 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, שאותה הכרנו בלולאת for. אך במקום הסינטקס המוכר יותר, יש הגדרת משתנה שהוא בעצם האיבר התורן במערך, המילה השמורה of ושם המערך. בתוך הסוגריים המסולסלים ירוץ בכל פעם האיבר התורן במערך:

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

משתנה לפי האיבר במערך

הבה נראה את האיטרציה הראשונה:

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

  { name: 'pen', price: 11 },
```

באיטרציה 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."

בהרצה השנייה, האיבר השני נכנס אל תוך המשתנה הזה ומה שיש בסוגריים המסולסלים רץ שוב:

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

  { name: 'pencil', price: 5 },
```

מה שרואים בהדפסה הוא:

// "The price of pencil is 5."

בהרצה האחרונה, האיבר השלישי ייכנס אל תוך המשתנה order בדיוק כמו האיברים שלפניו.

עכשיו נעבור לשבירת לולאה. בואו נניח שהדרישה היא להציג את המוצר הראשון שהמחיר שלו נמוך מ-10. לא את כל המוצרים אלא את המוצר הראשון. דרישה כזו יכולה כמובן להיות אמיתית לחלוטין. איך שוברים את הלולאה? באמצעות המילה השמורה break:

```
let orderArray = [
    { name: 'pen', price: 11 },
    { name: 'pencil', price: 5 },
    { name: 'TV', price: 2345 }
];
let answer;
for (let order of orderArray) {
    console.log(`I am passing ${order.name}.`);
    if (order.price < 10) {
        answer = order;
        break;
    }
}
// "I am passing pen."
// "I am passing pencil."
```

מה יש פה? תנאי שאומר שברגע שמחיר של מוצר כלשהו קטן מ-10, מכניסים את המוצר למשתנה answer ושוברים את הלולאה. אפשר לראות באמצעות ההערות שהלולאה אכן נשברת ולא ממשיכה אל האיבר השלישי כיוון 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;
  }
}
```

{ name: 'pen', price: 11 },

לא עונה לתנאי

יש הדפסה של:

// "I am passing pen."

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

באיטרציה השנייה:

```
let orderArray = [
  { name: 'pen', price: 11 },
  { name: 'pencil', price: 5 },
  { name: 'TV', price: 2345 }
];                                  { name: 'pencil', price: 5 },
let answer;
for (let order of orderArray) {
  console.log(`I am passing ${order.name}.`);
  if (order.price < 10) {  עונה לתנאי!
    answer = order;
    break;   הלולאה נשברת
  }
}
```

יש הדפסה:

// "I am passing pencil."

כיוון שהתנאי מתקיים, המשפט break מופעל והלולאה לא עוברת לאיטרציה השלישית. זה היתרון הגדול של for of – שהיא ממזגת את היכולת של while מבחינת שבירה ואת האלגנטיות של forEach.

לולאת map

פעמים רבות רוצים לשנות את המערך עצמו. למשל, הבה נניח שלקוח שמגיע מישראל רואה את המחירים ורוצים להציג לו אותם בשקלים, כלומר להכפיל את המחיר פי ארבעה. איך עושים את זה? אפשר להשתמש בלולאת for, בלולאת while, בלולאת forEach או בלולאת for of – אך אם רוצים לשנות את המערך מקובל מאוד להשתמש ב-map, שמאחורי הקלעים לוקחת את המערך הקיים והופכת אותו למערך חדש. לולאת map מיועדת לשינוי מערכים והיא פועלת באופן דומה ל-forEach, אלא שהפונקציה האנונימית יכולה להחזיר את הערך החדש שייכנס במקום הערך הקיים, כלומר לעשות לו מוטציה. בואו נבדוק את הדוגמה:

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

```
];
orderArray.map((value) => {
    let newPrice = value.price * 4;
    value.price = newPrice;
    return value;
});
console.log(orderArray);
/*
[
  { name: 'pen', price: 44 },
  { name: 'pencil', price: 20 },
  { name: 'TV', price: 9380 }
 }
]
*/
```

זה דומה מאוד ללולאת forEach שעליה כבר למדנו. ה-value שהפונקציה הפנימית מקבלת הוא האיבר שיש במערך. אבל אפשר לראות שכאן הפונקציה מחזירה משהו. מה היא מחזירה? את האיבר החדש שעבר שינוי. האיבר החדש ייכנס במקום האיבר הקודם, כולל כל השינויים שהכנסתם אליו:

אפשר להחזיר מה שרוצים, אפילו מחרוזת או אובייקט מורכב יותר. כל מה שמכניסים ייכנס במקום האיבר הראשון של המערך. באיטרציה השנייה משנים את האיבר השני, ומה שמחזירים נכנס במקום האיבר השני. הינה:

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

נכנס למקום
השני
במערך
המקורי

```
{ name: 'pencil', price: 20 }
```

ולסיום, באיטרציה האחרונה מקבלים את האיבר השלישי ומה שמחזירים ייכנס למקום השלישי במערך המקורי:

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

נכנס למקום
השלישי והאחרון
במערך המקורי

```
{ name: 'TV', price: 9380 }
```

וזהו. מה שיש במערך המקורי הוא לא מה שהיה פעם אלא האיברים החדשים.

לולאת filter

הלולאה האחרונה היא לולאה שמוציאה איברים מהמערך. בעוד map רק משנה איברים, באמצעות לולאת filter אפשר לצמצם את המערך. הבה נסתכל על המערך הזה, למשל:

```
let writingArray = [
    { name: 'pen', price: 11 },
```

```
    { name: 'pencil', price: 5 },
    { name: 'paper', price: 10 },
    { name: 'brush', price: 12 },
    { name: 'ink', price: 6 },
];
```

נניח שמבקשים להציג ללקוח רק דברים שהמחיר שלהם גבוה מ-7. אפשר לעשות משהו כזה:

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

כלומר, לקחת את המערך ולעבור עליו עם לולאה. בכל איטרציה (מעבר) של הלולאה בודקים את המחיר של האיבר הנוכחי. אם הוא קטן מ-7 או שווה לו, מוחקים אותו. האיבר הראשון, למשל, הוא pen, שה-price שלו הוא 11. התנאי לא יתקיים כי value.price הוא 11 והוא לא קטן מ-7 או שווה לו. האיבר השני הוא pencil וה-price שלו הוא 5. במקרה הזה התנאי מתקיים. כיוון שלולאת ה-forEach נותנת גם את ה-key, הלוא הוא מספר האיבר, שבמקרה של האיבר השני הוא 1 (זוכרים שמערך מתחיל מ-0, נכון?), אפשר להשתמש באופרטור delete, שעליו למדנו בפרק על מערכים, כדי למחוק את האיבר הזה, וכך הלאה. מה שנקבל בסוף הוא מערך עם שני undefined, היכן שהיו כל האיברים שמחירם נמוך מ-7.

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

זה קצת מבאס, ואפשר לעקוף את זה בכל מיני דרכים או באמצעות filter. מדובר בלולאה שזהה ל-map אך מחזירה רק true או false. אם היא מחזירה true האיבר נשאר במערך. אם היא מחזירה false האיבר מתפוגג מהמערך כאילו לא היה שם. בניגוד ללולאת map, הלולאה הזו לא משנה את המערך המקורי וצריך להחזיר את התוצאה למשתנה אחר.

הינה דוגמה:

```
const writingArray = [
    { name: 'pen', price: 11 },
    { name: 'pencil', price: 5 },
```

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

writingArray הוא מערך המכיל אובייקטים של מוצרי כתיבה. לכל אחד מהם יש שם ומחיר. כדי לסנן את כל האיברים שהמחירים שלהם נמוך מ-7, משתמשים בפונקציה ייחודית למערך שנקראת filter. הפונקציה הזו מקבלת פונקציה אנונימית שאותה כותבים כפונקציית חץ. אני רק מזכיר - פונקציית חץ היא דרך מסוימת לכתוב פונקציה אנונימית (כלומר פונקציה שאין לה שם אלא עוברת כארגומנט). הפונקציה האנונימית מחזירה true או false. אם היא מחזירה true האיבר נשאר במערך. אם היא מחזירה false הוא עף מהמערך. פונקציית filter מחזירה את המערך החדש.

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

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

באיטרציה השנייה מגיע תור האיבר השני:

```
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;
  }
});
```

התנאי מתקיים price = 5

הפונקציה מחזירה false

האיבר הזה לא זוכה להיכנס לאיבר החדש

הוא כבר לא זוכה להיכנס, כיוון שהמחיר שלו הוא 5 והתנאי הוא שכל מה שקטן מ-7 או שווה לו מקבל false. וכך הלאה. כמו כל לולאה אחרת, הלולאה הזו עוברת על כל המערך. הפונקציה שהעברנו מחזירה true או false לגבי כל איבר. האיברים שמקבלים true זוכים להיכנס למערך 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 filteredArray = writingArray.filter((value) => {
  if (value.price <= 7) {
    return false;
  } else {
    return true;
  }
});
```

לולאת sort

לולאה חשובה נוספת משמשת למיון מערכים. הלולאה הזו לא משנה איברים ולא את גודל המערך, אלא פשוט מסדרת את האיברים במערך. בניגוד לאיטרציות אחרת, לא חייבים להעביר פונקציה אנונימית שתעשה את הסדר. כברירת מחדל האיטרציה הזו מסדרת לפי סדר הא'-ב' – היא ממירה את הערכים למחרוזות טקסט ועורכת השוואה ביניהם, וכך מבצעת את הסדר. הינה דוגמה פשוטה מאוד שמדגימה את זה:

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

מובן שהעניינים מתחילים להסתבך אם במערך יש אובייקטים ולא רק מספרים או מחרוזות טקסט, אבל לא נכסה את איטרציית sort במלואה בספר הזה.

לולאות על אובייקטים

כל הלולאות שלמדנו עד כה משחקות יפה עם מערכים ובאמצעותן אפשר לעבור על כל איבר ואיבר במערך. אך נשאלת השאלה – איך לעבור על אובייקט? הכוונה למשהו בסגנון הזה:

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

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

לולאות for in

מדובר בלולאה שעוברת על כל תכונות האובייקט בדומה ל-for of. כך היא נראית:

```
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 ה-for in מכילה כמה חלקים. ראשית המילה השמורה for, ומייד אחריה סוגריים עגולים. בסוגריים העגולים מגדירים משתנה שהוא המפתח. בתוך הלולאה יקבל המשתנה הזה את שם המפתח התורן. אחר כך מכניסים את המילה השמורה in ואת המשתנה שבתוכו נמצא המערך.

בתוך הסוגריים המסולסלים מתחוללת האיטרציה של כל מפתח. ברגע שיש את המפתח key, אפשר לגשת אל הערך שלו באמצעות סוגריים מרובעים כפי שלמדנו בפרק על האובייקטים. המפתח של האובייקט הוא המשתנה key והערך הוא:

userObject[key]

בואו 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]}`);
}
```

באיטרציה הראשונה שמתרחשת, key הוא המפתח הראשון שיש באובייקט, הלוא הוא מחרוזת הטקסט userId. באמצעות המפתח אפשר לחלץ את הערך. איך? למדנו בפרק על אובייקטים שאם המפתח נמצא בשם של משתנה אפשר לחלץ אותו באמצעות שימוש בסוגריים מרובעים, וזה בדיוק מה שעושים. את המפתח ואת הערך מדפיסים באמצעות שימוש בתבנית טקסט פשוטה, שגם עליה למדנו:

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

key = userId

userObject[userId] = 12

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

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

key = userName

userObject[userName] = 'barzik'

באיטרציה האחרונה, key הוא ה-age.

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

key = age

userObject[age] = 40

הלולאה הזו ניתנת לשבירה על ידי break;.

בסך הכול פשוט ונחמד, נכון? אבל האמת היא שנדיר למצוא מתכנת שמשתמש בשיטה הזו, מפני שלולאת for in מביאה נתונים גם מאובייקטים אחרים שהם פרוטוטייפ – אב טיפוס - של האובייקט שלנו. כרגע זה נשמע סינית ועדיין לא דיברנו על זה, אבל על קצה המזלג אציין שבדרך כלל אובייקטים לא מגיעים כלוח חלק והם העתק של אובייקטים אחרים שיש להם מפתחות נוספים. לרוב לא מעוניינים במפתחות של האובייקטים האחרים, ולפיכך לולאת for in עלולה להיות הרת אסון, וזו הסיבה שבקושי משתמשים בה. אם כבר משתמשים בה, מקובל מאוד לוודא שהתכונה שבודקים היא התכונה של האובייקט, באמצעות שימוש בפונקציה הייחודית לאובייקטים שנקראת hasOwnProperty, באופן הבא:

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

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

Object.keys

הדרך השנייה והפופולרית ביותר היא להשתמש בפונקציה שנקראת Object.keys לחילוץ כל המפתחות של האובייקט כמערך, ואז לעבור על המערך הזה באיזו לולאה שרוצים, כי זה מערך. איך ניגשים לערך באובייקט? כמו בלולאת for in; ברגע שיש את המפתח יש גם את הערך. שימו לב לדוגמה הבאה:

```
const userObject = {
    userId: 12,
    userName: 'barzik',
    age: 40,
};
const userObjectKeys = Object.keys(userObject); //["userId", "userName",
"age"]
userObjectKeys.forEach((key) => {
    console.log(`key: ${key}, value: ${userObject[key]}`);
});
/*
"key: userId, value: 12"
"key: userName, value: barzik"
```

```
"key: age, value: 40"
*/
```

מה קורה פה? יש כאן את אותו אובייקט מהדוגמה הקודמת. באמצעות שימוש בפונקציית Object.keys מקבלים את כל המפתחות כמערך שנכנס ל-userObjectKeys, ועליו עושים לולאת forEach. כיוון שמדובר במערך, אפשר לעשות עליו forEach. נהדר ואלגנטי.

נסכם בטבלה המסבירה על כל סוגי הלולאות:

שם לולאה	**יתרונות**	**חסרונות**
לולאות למערכים		
לולאת for	פשוטה למימוש ניתנת לשבירה באמצעות break	ללא תנאים מורכבים צריך לדעת את גודל המערך קל להיכנס ללולאה אינסופית מגיעים גם לאיברי undefined
לולאת while	פשוטה למימוש ניתנת לשבירה באמצעות break אפשר לכתוב תנאים מורכבים	צריך לדעת את גודל המערך קל להיכנס ללולאה אינסופית מגיעים גם לאיברי undefined
לולאת forEach	לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים	צריך לכתוב פונקציית חץ לא ניתנת לשבירה
לולאת for of	פשוטה למימוש ניתנת לשבירה באמצעות break לא צריך לדעת את גודל המערך אין לולאה אינסופית לא נכנסים לאיברים ריקים	מקבלים רק את הערך של אברי המערך וצריכים לחלץ את המפתח לבד
לולאת map	לולאה שעוברת על המערך ויוצרת ממנו מערך חדש באותו גודל. אפשר לשנות את האיברים בקלות.	לא מוחקת איברים מהמערך
לולאות להמרת מערכים		
לולאת filter	מוחקת איברים	צריכים להחזיר את התוצאה למשתנה אחר לא משנה את המערך המקורי
לולאות לאובייקטים		
לולאת for in	פשוטה למימוש ניתנת לשבירה באמצעות break לא צריך לדעת את גודל האובייקט	מקבלים רק את המפתח של האובייקט וצריכים לחלץ את הערך לבד חייבים להשתמש ב-hasOwnProperty

	אין לולאה אינסופית לא נכנסים לאיברים ריקים	
שימוש ב-Object.keys	אפשר להשתמש בלולאות של המערך ואז מקבלים את כל היתרונות של הלולאה שבחרתם	מקבלים רק את המפתח של האובייקט וצריכים לחלץ את הערך לבד אפשר להשתמש בלולאות של המערך ואז מקבלים את כל החסרונות של הלולאה שבחרתם

תרגיל:

באמצעות לולאות for, הדפיסו שלוש פעמים על המסך "I know how to use for loop".

```
for (let i = 0; i < 3; i++) {
    console.log('I know how to use for loop');
}
```

הסבר:

זו לולאת for פשוטה שמתחילה 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 המדפיסה את שמות הלקוחות.

```
const customersArray = ['Moshe', 'Yaakov', 'Yossi', 'Baruch', 'Yael'];
for (let i = 0; i < customersArray.length; i++) {
    console.log(customersArray[i]);
}
```

הסבר:

משתמשים בלולאת for. המונה הוא i ששווה ל-0 והתנאי הוא שהלולאה תרוץ כל עוד המספר קטן מאורך המערך, במקרה הזה 5. משתמשים במונה i שזמין על מנת להדפיס את האיבר המתאים במערך. למשל, באיטרציה הראשונה i שווה ל-0. התנאי מתקיים כי 0 קטן מ-5 (אורך המערך). משתמשים ב-i על מנת להדפיס את האיבר שנמצא במיקום 0. באיטרציה השנייה, i שווה ל-1, התנאי מתקיים כי 1 קטן מ-5 (אורך המערך) ומשתמשים ב-i כדי להדפיס את האיבר שנמצא במיקום 1, וכך הלאה.

תרגיל:

נתון מערך של אובייקטי הזמנות:

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

הדפיסו את סכום ההזמנות במערך:

```
const invoices = [{ id: 1, price: 10 }, { id: 2, price: 32 }, { id: 3, price:
40 }];
let amount = 0;
for (let i = 0; i < invoices.length; i++) {
    amount = amount + invoices[i].price;
}
console.log(amount); //82
```

הסבר:

יוצרים לולאה שעוברת על המערך. המערך הוא לא של ערכים פרימיטיביים אלא של אובייקטים, אבל גם אובייקטים אפשר לשלוף באמצעות [i], כש-i משתנה לפי מספר האיטרציה. לפני שהלולאה רצה, יוצרים משתנה בשם amount ששווה 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:

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

הסבר:

לולאת while פשוטה שעובדת כל עוד i גדול מ-0 או שווה לו. בלולאה עצמה מדפיסים את המספר ואז מורידים 1 מ-i באמצעות כתיב מקוצר.

תרגיל:

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

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

הסבר:

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

תרגיל:

נתון מערך:

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

מדובר בפרופילים של משתמשים באתר היכרויות. כתבו פונקציית forEach שתדפיס בקונסולה אך ורק את המשתמשים שהם בני פחות מ-30.

```
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 אנונימית מסוג חץ. הפונקציה הזו נראית כך:

```
(value, index) => {
    if (value.age < 30) {
        console.log(value.userName);
    }
}
```

לא צריך להיבהל ממנה. זו פונקציה רגילה שרצה על כל איבר במערך באופן אוטומטי. יש ערך שבחרתי לקרוא לו value והמספר של האיבר במערך. מה שמעניין הוא הערך. הערך הוא בעצם האיבר התורן במערך. בכל איטרציה יש איבר אחר. באיטרציה הראשונה מדובר באיבר הראשון, באיטרציה השנייה מדובר באיבר השני ובאיטרציה השלישית מדובר באיבר האחרון. זה הכול. בודקים את האובייקט שיש באיבר או, נכון יותר, את תכונת ה-age שלו, ומדפיסים את מי שגילו פחות מ-30.

תרגיל:

נתון מערך (אותו מערך כמו בתרגיל הקודם):

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

צרו פונקציה שמקבלת מספר ומערך ומחזירה אך ורק מערך משתמשים שהגיל שלהם שווה למספר הזה או גדול ממנו.

```
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:

יוצרים פונקציה שמקבלת שני ארגומנטים – מספר מסוים ומערך. משתמשים בפונקציית filter על מנת לסנן את המערך ולהשאיר שם רק את המשתמשים שרוצים. איך נראית פונקציית filter? בדיוק 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 בבושת פנים. 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 ורואים שמקבלים משתמש אחד שהוא בן 40, כיוון ש-40 לא קטן מ-40, הוא מקבל true בפונקציית הפילטר.

ג'אווהסקריפט בסביבת דפדפן

עד כה, כל התרגולים היו עם הקונסולה בלבד. סביבת העבודה היא בדפדפן ועם עורך טקסט, אבל בגדול הדבר היחידי שעשיתם בדפדפן היה בקונסולה. הגיע הזמן להתחיל לממש את הידע שלמדנו ולראות מה אפשר לעשות בעזרתו במציאות.

כיום ג'אווהסקריפט רצה במגוון אדיר של סביבות, אבל בעבר הסביבה היחידה שבה היא רצה הייתה הדפדפן. כלומר, באתרי אינטרנט השתמשו בה כדי ליצור התנהגויות אינטראקטיביות עם המשתמש, אנימציות, אפקטים, וידיאו ועוד. באופן עקרוני, כל הידע שלמדנו עד עכשיו הוא הבסיס הנדרש ליצירת כל הדברים האלו בדפדפן. יכול להיות שלולאות, אובייקטים ופונקציות נראים לא מעניינים לעומת אנימציות וסרטונים, אבל הם מה שפועל מאחורי הקלעים כדי לאפשר את כל הדברים היפים בדפדפן.

הספר הזה מלמד ג'אווהסקריפט ולא HTML, שהוא עולם ומלואו. בפרק הזה נלמד HTML באופן בסיסי בלבד, רק מספיק כדי להבין איך ג'אווהסקריפט קשורה לכך. לשפת ג'אווהסקריפט יש קשר הדוק עם ה-HTML של הדף.

הסבר כללי על HTML

ראשי התיבות של HTML הם Hyper Text Markup Language והוא בעצם הטקסט המרכיב את כל דפי האינטרנט שאתם מכירים. כשנכנסים אל דף כלשהו ברשת, ולא משנה אם מדובר באתר חדשות או באתר משחקים או סרטים – רואים דף HTML. דף כזה הוא בעצם דף טקסט. בדיוק כמו קוד ג'אווהסקריפט שאפשר לכתוב ב-

notepad, גם קוד HTML הוא קוד שאפשר לכתוב בכל מקום. אם אתם רוצים לראות את קוד ה-HTML של כל דף, לחצו על הכפתור הימני בעכבר ואז על "הראה מקור" או "view source"'. תראו מייד המון-המון טקסט עם סימנים כאלו < וכאלו >. הדברים שנמצאים בתוך החצים נקראים "אלמנטים של HTML" ויכולים להכיל טקסט או אלמנטים אחרים. אלמנט בסיסי של HTML נראה ככה:

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

אלמנט מורכב מתגית פתיחה עם סוג האלמנט, תוכן האלמנט, במקרה הזה טקסט פשוט, ותגית סגירה שמורכבת מלוכסן ומשם האלמנט.

אם מדובר באלמנט שנקרא "אלמנט שנסגר מאליו", הוא נראה כך:

```
<elementName />
```

בגדול, כל דבר שרואים 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. הקפתי כמה אלמנטים בריבועים. למשל הלוגו של ויקיפדיה הוא אלמנט מסוג תמונה. הטקסט "ברוכים הבאים לוויקיפדיה!" הוא אלמנט מסוג h1, כלומר כותרת ראשית. יש גם אלמנטים בתוך אלמנטים. טקסט הפתיחה הוא אלמנט ויש לו אלמנטים בנים של קישורים (סימנתי כמה מהם). כאמור אלמנט HTML נראה כך:

```
<elementName attributeName="attributeValue" id="elementId"
class="elementClass">
</elementName>
```

ה-elementName הוא שם האלמנט. למשל img הוא תמונה, h1 הוא כותרת ראשית, h2 הוא כותרת משנית, p הוא פסקה, a הוא קישור, div הוא אלמנט כללי וכך הלאה. יש עשרות סוגי אלמנטים.

לכל אלמנט יכולות להיות תכונות (attributes). למשל, אם יש קישור, אחת התכונות של האלמנט היא המקום שאליו הקישור מפנה. התכונה הזו נקראת href. תכונה אחרת שיכולה להיות לקישור היא title – הטקסט שמופיע אם מציבים את העכבר מעל הקישור. הינה דוגמה למה שרואים אם כותבים בתוך מסמך HTML אלמנט של קישור:

```
<a href="https://google.com" title="Link to google">Google.com</a>
```

אפשר לראות איך כל תכונה של האלמנט יכולה לשנות משהו בהתנהגותו.

יש אלמנטים שמכילים טקסט כמו קישור או פסקה:

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

ויש כאלו שלא, כמו תמונה. למשל:

```
<img src="https://internet-israel.com/wp-content/uploads/2016/05/me.jpg" />
```

ההבדל הוא שהאלמנטים ללא הטקסט נסגרים בלוכסן בסופם ולא בשם האלמנט. כלומר ממש לא כך:

```
<img src="https://internet-israel.com/wp-
content/uploads/2016/05/me.jpg"></img>
```

כאמור, יכולים להיות אלמנטים בתוך אלמנטים. הינה למשל פסקה שמכילה שני אלמנטים של קישור:

```
<p>There are several search engines like <a href="https://google.com"
title="Link to google">Google</a> and <a
    href="https://duckduckgo.com/" title="Link to DuckDuckGo">DuckDuckGo</a>
</p>
```

במקרה הזה המבנה של המסמך הוא:

כלומר ה-HTML עצמו שנחשב אבא, תגית ה-body, ואז אלמנט הפסקה שהוא אלמנט האב. לאלמנט הפסקה (האב) יש שני ילדים שהם ה-a – שני קישורים, אחד לגוגל ואחד ל-Duckduckgo. בכל דף HTML ממוצע יש מאות אלמנטים כאלו מסוגים רבים ושונים ועל כולם אפשר להשפיע באמצעות ג'אווהסקריפט.

מזהים של תגיות HTML

לאלמנטים של HTML יכולים להיות מזהים, משהו שבעזרתו אפשר לזהות את האלמנט לצרכים שונים. בדיוק כמו בני אדם שיש להם שם פרטי ושם משפחה, גם לאלמנטים יש שני מזהים. המזהה הראשון הוא כללי וקוראים לו class. לא להתבלבל עם קלאס של ג'אווהסקריפט. במקרה של HTML, class הוא מקביל לשם משפחה, ובדיוק כפי שיש כמה אנשים בעולם עם שם המשפחה שלי, אפשר לתת את אותו class לכמה וכמה אלמנטים שונים. הם יכולים להיות בקשר אב–ילד או לא. אין הגבלה! אפשר גם לתת כמה שמות משפחה לאותו אלמנט. המזהה השני נקרא id והוא מזהה ייחודי. אסור לתת את אותו id לשני אלמנטים באותו דף. ה-id הוא ייחודי ומיוחד לכל אלמנט, ואפשר לתת id אחד בלבד לכל אלמנט.

אפשר כמובן להצמיד גם מזהה class וגם מזהה id לאלמנט אחד.

במה משתמשים? תלוי במה שרוצים לבצע ובמבנה הדף. לאלמנטים מיוחדים שרוצים להגיע אליהם בקלות באמצעות ג'אווהסקריפט נותנים id בלבד.

גישה אל תגיות באמצעות ג'אווהסקריפט

לתרגול, צרו דף HTML לדוגמה ותציבו בו ג'אווהסקריפט. הדף הזה זהה לדף שהסברתי עליו בפרק על בניית סביבת העבודה, אבל חשוב להקפיד על דבר אחד – דפדפן מריץ את קוד הג'אווהסקריפט ומרנדר את האלמנטים בסדר מקבילי מלמעלה למטה. לצורך התרגול חשוב מאוד שקובצי הג'אווהסקריפט והקוד של הג'אווהסקריפט יהיו בתחתית הדף, כיוון שאם הקוד יחפש את האלמנטים של ה-HTML או יתייחס אליהם, הדפדפן יהיה חייב להכיר

אותם לפני שהוא מריץ את הקוד – כלומר קובצי הג'אווהסקריפט שמתייחסים לאלמנטים חייבים להיות מתחת לאלמנטים. זו הסיבה שבגללה יש להקפיד, לפחות כרגע, שקוד הג'אווהסקריפט יהיה מתחת לאלמנטים.

```
<!doctype html>
<html>

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

<body>
  <!-- HTML elements will be here -->
  <script src="./source.js"></script>
</body>

</html>
```

בקוד יש הערת HTML. כל מה שמוקף ב-HTML בתגיות <!-- --> נחשב להערה ולא מודפס, בדומה להערת ג'אווהסקריפט. היכן שיש הערת HTML – זה המקום שבו כותבים את האלמנטים. תגית הסקריפט תכיל קישור לקובץ הג'אווהסקריפט, ולפיכך היא חייבת להיות מתחת ל-HTML.

אלמנטים של HTML מתורגמים לאובייקטים של ג'אווהסקריפט

כל אלמנט שיש ב-HTML יכול להיות מתורגם לאובייקט ג'אווהסקריפטי כשר למהדרין. כשקוד ג'אווהסקריפט חי בתוך HTML הוא יכול לגשת אל ה-HTML הזה באופן ישיר בלי צורך בטעינה. ה-HTML מתורגם לסוג של ישות שנקראת DOM, ראשי התיבות של Document Object Model. הישות הזאת נמצאת בזיכרון וזמינה דרך קוד הג'אווהסקריפט. נשמע אבסטרקטי מדי? הבה נראה. אחרי שמציבים את הג'אווהסקריפט בתוך ה-HTML, יוצרים בתוכו אובייקט h1, משהו בסגנון הזה:

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

בקובץ של הג'אווהסקריפט מכניסים את הקוד הבא:

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

אם טוענים את העמוד אפשר לראות שהטקסט שבכותרת הוא לא "Headline" אלא "Hello world". איך עושים את זה? ראשית משתמשים באובייקט הגלובלי document. האובייקט הזה קיים בכל ג'אווהסקריפט שנמצאת 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-id הוא הערך שהושם ב-atrribute מסוג id של האלמנט h1. המתודה מחזירה את הייצוג של האלמנט ב-DOM עם כל המתודות שלו. ומעכשיו? חגיגה שלמה. אחד המאפיינים הוא innerHTML, שמאפשר לשנות את תוכן האלמנט.

אפשר גם ליצור אלמנטים חדשים לחלוטין ולהכניס אותם למסמך:

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

כאן למשל משתמשים במתודה createElement כדי ליצור אלמנט מסוג פסקה או תגית <p/><p>, להכניס אליה תוכן ואז להצמיד אותה אל ה-body של המסמך באמצעות appendChild. עד שמשתמשים 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 שמתרחשים בדף. אירוע יכול להיות קליק, מעבר של העכבר על אלמנט והמון דברים אחרים. בדוגמאות כאן אתמקד באירוע של קליק. הבה ניצור ב-HTML כפתור שיש לו id:

```
<!doctype html>
<html>

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

<body>
  <button id="myButton" type="button">Click me!</button>
  <script src="./source.js"></script>
</body>

</html>
```

אם תפתחו את הדף, תראו שנוצר כפתור עם הכיתוב "Click Me!"

בקובץ הג'אווהסקריפט תופסים את אלמנט הכפתור כרגיל עם document.getElementById ומכניסים אותו למשתנה:

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

עכשיו נרצה שבכל פעם שיקליקו על הכפתור יראו משהו בקונסולה. את זה עושים בעזרת אירוע DOM. אירוע DOM מופעל על ידי אינטרקציה עם המשתמש שלוחץ על הכפתור, והטיפול באירוע מתרחש באמצעות הפונקציה

addEventListener שמגדירים אותה כמטפלת באירוע. מתודת addEventListener זמינה בכל אלמנט. היא מקבלת שני ארגומנטים. הראשון הוא סוג האירוע, ובמקרה הזה: click. השני הוא פונקציה שפועלת בכל פעם שיש קליק. משתמשים במקרה הזה בפונקציית חץ 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 תריצו את זה תוכלו לראות שבכל פעם שלוחצים על הכפתור, בקונסולה רואים "קליק". אבל למה להסתפק רק בקונסולה? אפשר ליצור אלמנט שכתוב 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 והפלט. הוא הרינדור של האלמנט. אפשר ליצור קלט משוכלל 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. תופסים את האלמנט באמצעות getElementById ואז ניגשים אל תכונת ה-value שלו:

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

הצעד הבא יהיה לקחת 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);
});
```

אסביר שוב על הקוד המלא: ב-HTML יש שני אלמנטים, אלמנט אחד שיש לו id בשם myButton והוא כפתור, ואלמנט נוסף של שדה טקסט שה-id שלו הוא myInput. שניהם נכנסים אחר כבוד למשתנים המתאימים כדי שיתאפשר להתייחס אליהם. בשלב הבא מצמידים אירוע קליק לכפתור. האירוע עובד בכל פעם שיש לחיצה. מדובר בפונקציה טהורה, כלומר כזו שהיא ג'אווהסקריפט בלבד, שבמקרה הזה יוצרת אלמנט מסוג פסקה. ב-HTML פסקה היא תגית <p>. היא לוקחת את מה שהמשתמש הכניס לתוך שדה הטקסט, מכניסה אותו לתכונת innerHTML של אלמנט הפסקה ומצמידה את אלמנט הפסקה אל גוף ה-HTML, וכך אפשר לראות אותו.

מה שחשוב להבין הוא שמדובר כאן בג'אווהסקריפט לכל דבר אך בשילוב ה-DOM – כלומר אובייקטים ואירועים שמגיעים מהאלמנטים שיש ב-HTML. ברגע שמשתמשים ב-document כדי ליצור אלמנט או לקרוא לו, הוא מקבל גם תכונות כמו innerHTML ואפשר להצמיד אליו אירועים מיוחדים כמו "קליק" באמצעות addEventListener.

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

בכל הדוגמאות עד כה הראיתי איך מצמידים 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 להצמיד אלמנטים לכל אלמנט אחר!

אדגים באמצעות תגיות <ul> ו-<li>. 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 ובו רשימת קניות שהמשתמש יכול להכניס אליה פריטים. ה-HTML יכלול חלון של שדה טקסט, כדי שהמשתמש יוכל להכניס פריטים, כפתור של "הכנס" וכמובן שלד של הרשימה:

```
<!doctype html>
<html>

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

```
</head>

<body>
  <button id="myButton" type="button">Click me!</button>
  <input id="myInput" />
  <ul id="myList"></ul>
  <script src="./source.js"></script>
</body>

</html>
```

קוד הג'אווהסקריפט יהיה פשוט. בדיוק כמו בדוגמה הקודמת, מגדירים את שלושת השחקנים במשחק: הכפתור שמפעיל את כל הפונקציה של ההוספה, שדה הטקסט שבו המשתמש 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 לכפתור. את זה עושים באמצעות המתודה addEventListener. מעבירים שני ארגומנטים. הראשון הוא סוג האירוע (קליק), והשני הוא ארגומנט של הפונקציה שתעבוד כשהארגומנט יפעל. מעבירים אותה כפונקציית חץ:

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

הקוד שבתוך 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 ללמוד CSS, ובגדול אפשר לשלוט 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) px10, שמשמעותו 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 משתמשים בתכונת style, שקיימת בכל אלמנט DOM. לשם ההדגמה אצור אלמנט HTML מסוג div עם id מסוים ב-HTML. אלמנטים מסוג div הם אלמנטים בסיסיים שאין להם עיצוב:

```
<!doctype html>
<html>

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

<body>
  <div id="myDivId"></div>
  <script src=" ./source.js"> </script>

</body>
</html>
```

אם תציגו את הדף הזה באמצעות הדפדפן לא תראו כלום. הבה נכניס קצת צבע לאלמנט הזה באמצעות ג'אווהסקריפט:

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

יוצרים תכונות של גובה, של רוחב ושל צבע רקע על מנת ליצור ריבוע אדום. זה ממש נחמד. אפשר להכניס, לשנות או לקרוא כל תכונה של style באופן חופשי כמובן. שימו לב שמדובר בהשפעה על תגית ה-style של האלמנט ולא בשינוי קובץ ה-CSS שקשור למסמך. לרוב לא מקובל לשנות צבעים באופן ישיר בג'אווהסקריפט, אלא ליצור class של CSS שמכיל את התכונות המבוקשות ואז להוסיף את הקלאסים האלו לאלמנטים או להוריד אותם מהם באמצעות ג'אווהסקריפט.

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.add מוסיף class ו-classList.remove מסיר אותו מהאלמנט שמחזיקים במשתנה. אם כל מה שכתוב לעיל נשמע לכם ג'יבריש – זה הזמן ללמוד CSS. כאמור, אני לא מכסה HTML או CSS בספר זה.

סלקטורים של DOM

סלקטור הוא שם כללי למתודות 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 id הוא ייחודי כי יוצאים מנקודת הנחה שתמיד יש אלמנט אחד שיש לו את ה-id הזה. במקרים שיש סלקטורים המבוססים על class או על תכונה אחרת, יתקבל מערך של תוצאות שאפשר להתייחס אליו כאל כל מערך, אלא שבמערך הזה יהיו אלמנטים של DOM.

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

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

(שימו לב שלשם 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 הזה ולצבוע אותם באדום (לצורך העניין) עושים משהו כזה:

```
const recs = document.getElementsByClassName('rectangle'); //[div.rectangle,
div.rectangle]
for (let el of recs) {
    el.style.backgroundColor = 'red';
    el.style.height = '100px';
    el.style.width = '100px';
};
```

מקבלים מערך של כל האלמנטים שיש להם class בשם rectangle באמצעות השורה:

```
const recs = document.getElementsByClassName('rectangle');
```

עוברים על המערך הזה בדיוק כמו על כל מערך רגיל בג'אווהסקריפט, במקרה הזה באמצעות for of, אבל אפשר להשתמש בכל לולאה שהיא. למדנו על לולאת for of בפרק על לולאות, אז אתם אמורים להבין מה קורה פה.

עוברים על כל איבר בלולאה ונותנים לו גובה, רוחב וצבע. אם רוצים לתת אותם רק לאלמנט הראשון, צריך לעשות משהו כזה:

```
recs[0].style.backgroundColor = 'red';
recs[0].style.height = '100px';
recs[0].style.width = '100px';
```

חשוב להבין שאף על פי שמדובר באלמנטי DOM, אנחנו בג'אווהסקריפט. כל מה שלמדנו עד עכשיו – אובייקטים, לולאות, מערכים, סוגי מידע – הכול רלוונטי גם לג'אווהסקריפט בסביבת דפדפן. כשמריצים ג'אווהסקריפט בסביבת דפדפן – גם ה-DOM וגם הג'אווהסקריפט חיים בתוך הדפדפן והוא מנהל את עץ ה-DOM ואת המנוע של ה-JS ודואג לחבר בין השניים. למען האמת, מפני שאתם כבר מכירים את היסודות התיאורטיים של השפה, אתם אמורים לשלוט גם בג'אווהסקריפט בסביבת דפדפן. מדובר באותה שפת ג'אווהסקריפט בדיוק.

אפשר להעביר כארגומנט רשימה של קלאסים עם הפרש של רווח ביניהם.

אפשר גם לבחור אלמנטים לפי שם התגית. למשל, הקוד הזה בוחר את כל התגיות שהן מסוג div וצובע אותן באדום. העיקרון הוא אותו 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. כאן נדרשת הבנה 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 שלהם הוא rectangle והם ילדים של good_parent, אפשר להשתמש בסלקטור של CSS שהוא:

.good_parent div.rectangle

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

הקוד 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 שמבינים שיש סלקטורים שמחזירים מערך של אלמנטי DOM במקום אלמנט DOM אחד, ושהמערך הזה הוא מערך ג'אווהסקריפטי לכל דבר ועניין אלא שהוא מכיל אלמנט DOM – נגמר הסיפור.

ל-querySelectorAll יש גם גרסה של פונקציה בשם querySelector שמחזירה רק את התוצאה הראשונה של הסלקטור ולא מערך 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 לחלוטין ל:

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

לשם הנוחות אצרף רשימה של סלקטורים נפוצים ותפקידם:

שם הסלקטור	תפקיד
document.getElementById	בוחר אלמנט אחד ויחיד לפי ה-ID מחזיר אלמנט אחד
document.getElementsByClassName	בוחר כמה אלמנטים לפי קלאס אחד או יותר (הקלאסים מועברים כמחרוזת טקסט עם רווח)

	מחזיר מערך של אלמנטים
document.getElementsByTagName	בוחר כמה אלמנטים לפי שם האלמנט - סימן לכל האלמנטים ב-OM מחזיר מערך של אלמנטים
document.getElementsByName	בוחר כמה אלמנטים לפי תכונת השם – name מחזיר מערך של אלמנטים
document.querySelectorAll	בוחר כמה אלמנטים לפי סלקטור של CSS אפשר להכניס כמה סלקטורים עם פסיק ביניהם מחזיר מערך של אלמנטים
document.querySelector	בוחר את האלמנט הראשון שמציית לסלקטור של ה-CSS מחזיר אלמנט אחד

אירועים נוספים

עד כה למדנו רק על אירוע קליק, אך יש אירועים רבים נוספים הקשורים לאלמנטים של DOM. למשל hover, המופעל בכל פעם שהעכבר עובר מעל אלמנט מסוים. למשל, ל-HTML הזה:

```
<div id="rectangle">Rec 1</div>
```

אפשר לקבוע את הצבע ואת הגודל באמצעים שכבר למדנו:

```
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
```

אבל אפשר גם לגרום לו לשנות את הצבע באמצעות אירוע, אם העכבר יעלה עליו. את זה עושים באמצעות הצמדת אירוע בשם mouseover. הטכניקה זהה לטכניקה שכבר למדנו:

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('mouseover', (event) => {
    div.style.backgroundColor = 'yellow';
});
```

addEventListener מקבל שני ארגומנטים. הראשון הוא שם האירוע והשני הוא הפונקציה שעובדת כשהאירוע מופעל. משתמשים בפונקציית חץ, בדיוק 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'אווהסקריפט לשם התרגול וההסבר. במציאות אפקטים כאלו נעשים באמצעות CSS בלבד.

אני רוצה להתעכב קצת על הארגומנט שפונקציית החץ מקבלת, והוא ה-event. כשלמדנו על פונקציות חץ ועל קולבקים בפרק על הפונקציות, הסברתי שכל קולבק יכול לקבל כמה ארגומנטים שמי שמפעיל אותו מעביר. במקרה הזה הארגומנט שמקבלים הוא אובייקט האירוע עצמו, שמכיל עשרות פרטים. למשל, אם הכפתורים 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 הכפתור 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 בכל שם כמובן) הוא אובייקט כמו כל אובייקט. אפשר לעשות עליו console.log כדי לתהות על קנקנו ואפילו מומלץ להביט בו. אפשר לעשות איתו דברים מעניינים מאוד.

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

```
const div = document.getElementById('rectangle');
div.style.backgroundColor = 'red';
div.style.height = '100px';
div.style.width = '100px';
div.addEventListener('mouseenter', (event) => {
    div.style.backgroundColor = 'yellow';
});
div.addEventListener('mouseleave', (event) => {
    div.style.backgroundColor = 'red';
});
```

שימו לב שיש משמעות לאירועים. למה לא השתמשתי ב-mouseover? כיוון שהוא מופעל בכל פעם שהעכבר נמצא מעל האלמנט, כלומר המון פעמים – בכל פעם שהעכבר זז והתזוזה שלו היא מעל האלמנט. האירוע mouseeneter מופעל רק פעם אחת – כשסמן העכבר נכנס לתוך האלמנט. זה הכול.

יש המון אירועים, מאירועי מקלדת ועד אירועי עכבר. יש גם אירועים של אלמנטים (למשל אירוע שמופעל אם מישהו משנה את ה-CSS של האלמנט) ואירועים הקשורים ל-clipboard. הינה רשימה של כמה אירועים עיקריים:

אירוע	הסבר
click	אירוע של הקלקה באמצעות העכבר
mouseenter	סמן העכבר נכנס אל תחום האלמנט
mouseleave	סמן העכבר יוצא מתחום האלמנט
mousedown	המשתמש לוחץ על כפתור העכבר בתוך האלמנט
mouseup	המשתמש שחרר את כפתור העכבר בתוך האלמנט
wheel	המשתמש מגלגל את גלגלת העכבר כאשר הסמן נמצא על האלמנט
keydown	לחיצה על כפתור במקלדת
keyup	שחרור של כפתור במקלדת
focus	המשתמש מבצע פוקוס על האלמנט (למשל באמצעות הטאב או העכבר)
blur	המשתמש יוצא מפוקוס על האלמנט (למשל באמצעות הטאב או הקלקה על אלמנט אחר)

פעפוע של אירועים

פעפוע (באנגלית bubbling) הוא שם של תופעה שחשוב להכיר באירועים, והיא מתרחשת כאשר יש יותר מכמה אלמנטים. למשל כמו בדוגמה הזו:

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

כאן יש שלושה אלמנטים. אלמנט div אחד עם id של parent שבתוכו יש כפתור עם id של button. בתחתית יש div שכתוב בו result.

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

```
const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
```

התוצאה צריכה להיות משהו כזה:

מה קורה אם מצמידים לאלמנט האב אירוע? תזכורת: אלמנט האב הוא ה-div שמכיל את הכפתור. בדוגמה הזו לאלמנט האב יש id 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 הכפתור? אותו הדבר! למה? כי האירוע מפעפע גם לאלמנטים של הילדים. האירוע מתחיל בכפתור ומפעפע למעלה עד שיש אלמנט שמטפל בו (ובמקרה הזה האב). כלומר, אם אני אלמנט ויש אירוע על האלמנט שמכיל אותי (או על

האלמנט שמכיל את האלמנט שמכיל אותי וכו'), אז האירוע יפעפע כלפי מעלה עד שיפגע באירוע כלשהו (או עד שיגיע ל-body). לוחצים על הכפתור, אין בו אירוע לחיצה. האירוע עולה למעלה. האם באלמנט האב יש אירוע? כן! אז האירוע מופעל וממשיך לעלות למעלה, והוא מפעיל את כל האירועים באלמנטי האב השונים.

זה נחמד ושימושי, אבל העסק מתחיל להסתבך 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 כשילחצו על הכפתור? האירוע של הכפתור יתרחש, ואכן ב-div של התוצאה יודפס Button click detected, אבל מייד אחר כך, האירוע יפעפע למעלה ויפעיל את האירוע של parent, שבתורו ידפיס מייד 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 בלחיצה על הכפתור Button click detected, ורק כשלוחצים על ה-div של האב לקבל Parent click detected.

מה יקרה אם לא רוצים שהאירוע יפעפע למעלה? באירוע שבו רוצים לעצור את הפעפוע חייבים להפעיל את:

event.stopPropagation();

שזמין באירוע עצמו, לצד שאר תכונות האירוע. קחו למשל הקוד הזה:

```
const parent = document.getElementById('parent');
const result = document.getElementById('result');
const button = document.getElementById('button');
parent.style.backgroundColor = 'gray';
parent.style.height = '200px';
parent.style.width = '200px';
parent.addEventListener('click', (event) => {
    result.innerHTML = 'Parent click detected!';
});
button.addEventListener('click', (event) => {
    result.innerHTML = 'Button click detected!';
    event.stopPropagation();
});
```

הוא יעשה בדיוק את מה שרוצים שהוא יעשה. לחיצה על האב תדפיס שלחצתם על האב. לחיצה על הכפתור תפעיל את האירוע של הכפתור, אך האירוע לא יחלחל הלאה 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 משהו כזה:

```
<img src="hello.png" onclick="alert('Hi');" />
```

יצירת alert ו-prompt

עוד דרך מיושנת לתקשר עם משתמשים היא באמצעות alert. מדובר בפונקציה גלובלית שזמינה בכל ג'אווהסקריפט שחיה בסביבת דפדפן. משתמשים בה כך:

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

מה שהמשתמש יראה הוא חלון לא גרפי שיקפוץ עם ההודעה.

עיצוב החלון תלוי בדפדפן, בעיקר מפני שמדובר בחלון של מערכת ההפעלה. כאמור, מדובר בדרך מיושנת להצגת הודעות למשתמש וכיום ממעטים להשתמש בה. מומלץ מאוד להציג למשתמש הודעות בדרך אחרת, כפי שהראיתי לכם – הקצאת div מיוחד והדפסת הודעות עליו.

דרך נוספת לתקשר עם המשתמש היא באמצעות הפונקציה הגלובלית prompt, שגם היא זמינה בכל ג'אווהסקריפט שחיה בסביבת דפדפן. הפונקציה הזו מקבלת שני ארגומנטים. הארגומנט הראשון מכיל את המחרוזת למשתמש. כשמפעילים אותה, מוצגת למשתמש המחרוזת שהוכנסה בארגומנט הראשון עם שדה טקסט שבו הוא יכול להכניס מחרוזת טקסט משלו, ואותה פונקציית ה-prompt מחזירה. הארגומנט השני הוא ערך ברירת המחדל שמוצג למשתמש.

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

הקוד הזה למשל יציג את השאלה:

אם המשתמש יכניס שם, מייד אחר כך יוקפץ alert עם השם שהוא הכניס.

גם כאן מדובר בדרך מיושנת מאוד ועדיף לא להשתמש בה. אם רוצים לתקשר עם המשתמש, עושים זאת ב-input, כפי שראינו בדוגמאות הקודמות.

תרגיל:

צרו אלמנט מסוג span עם תוכן של "אני יודע קוד" והכניסו אותו ל-HTML.

הסבר:

יוצרים HTML בסיסי עם קישור לקובץ ג'אווהסקריפט, שיהיה 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 הג'אווהסקריפט מכניסים את הקוד הבא:

```
let myObj = document.createElement('span');
myObj.innerHTML = 'אני יודע קוד';
document.body.appendChild(myObj);
```

הקוד עצמו פשוט למדי. יוצרים אלמנט באמצעות מתודת createElement שנמצאת באובייקט document. האובייקט זמין בכל קובץ ג'אווהסקריפט שנמצא ב-HTML. את התוצאה של המתודה מכניסים למשתנה myObj. מעכשיו יש אלמנט DOM שיש לו מתודות משלו, וכל אחת מהן משפיעה עליו. אחת החשובות שבהן היא innerHTML, שמכניסה 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 ממוספרת באורך של המספר. למשל, אם המשתמש הכניס 3, תתקבל הרשימה:

- 1
- 2
- 3

פתרון:

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 וכפתור להפעלת הפונקציה שתיקח את המספר ותכניס פריטים לרשימה. ב-HTML יהיה קישור לקובץ ג'אווהסקריפט שבו יהיה הקוד. אלו שלושת השחקנים.

בקוד הג'אווהסקריפט מגדירים את שלושת השחקנים באמצעות document.getElementById. מרגע זה מקבלים גישה לתכונות שלהם ב-DOM.

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

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

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

בפונקציה עצמה מגדירים את המספר שמקבלים מהמשתמש:

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

ברגע שיש מספר, מפעילים לולאת for שתרוץ לפיו:

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

בתוך הלולאה יוצרים אלמנט li (li הוא אלמנט של אייטם ברשימה) ומכניסים אותו לרשימה. מאכלסים את תוכנו באמצעות i +1. זה הכול.

תרגיל:

צרו כפתור ואלמנט מסוג div. לחיצה על הכפתור תהפוך את האלמנט לצהוב, ברוחב של 100 פיקסלים ובגובה של 100 פיקסלים.

הסבר:

קוד ה-HTML:

```
<!doctype html>
```

```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'אווהסקריפט:

```javascript
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'אווהסקריפט.

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

תרגיל:

בהמשך לתרגיל הקודם, כתבו קוד שיגרום לכך שלחיצה על הכפתור תצבע את האלמנט בצהוב, לחיצה נוספת תצבע אותו באדום, לחיצה נוספת תצבע אותו שוב בצהוב וכך הלאה.

הסבר:

קוד ה-HTML:

```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'אווהסקריפט:

```
const button = document.getElementById('myButton');
const div = document.getElementById('myDiv');
let activated = true;
div.style.height = '100px';
div.style.width = '100px';
div.style.backgroundColor = 'yellow';

button.addEventListener('click', (event) => {
    if (activated === true) {
        div.style.backgroundColor = 'yellow';
        activated = false;
    } else {
        div.style.backgroundColor = 'red';
        activated = true;
    }

});
```

קוד ה-HTML אינו שונה מהקוד בתרגיל הקודם. שני השחקנים העיקריים הם div ו-button.

בקוד הג'אווהסקריפט מגדירים את שני השחקנים כאלמנטים של DOM באמצעות getElementById. מגדירים משתנה בוליאני בשם activated ומגדירים אותו כ-true. נוסף על כך מאתחלים את הריבוע הצהוב. קובעים את הרוחב, את הגובה ואת הצבע ההתחלתיים.

מצמידים אירוע קליק לכפתור באמצעות addEventListener, שכבר הכרנו בתרגילים קודמים. הפונקציה ש-addEventListener מפעיל בכל קליק מורכבת מעט. ראשית בודקים ש-activated הוא true. אם כן, צובעים את האלמנט בצהוב ומשנים את ה-activated ל-false.

אם activated הוא false, הוא נצבע באדום ומשנים את ה-activated ל-true.

דיבאגינג

בכל כלי מפתחים שהוא – בין שמדובר בכרום ובין שבפיירפוקס או באדג' – יש אפשרות להפעיל כלי שנקרא "דיבאגר". כלי מפתחים הוא רכיב תוכנה שנמצא בכל דפדפן ללא צורך בהורדה או בהפעלה. מדובר בשם כללי לכלי שמסייע למפתחים לבחון את הקוד שלהם. בדפדפן הכלי הזה עוזר לפתור תקלות בג'אווהסקריפט ולהבין מה קורה בנבכי הקוד. עד כה השתמשנו בעיקר ב-console.log על מנת להדפיס דברים בקונסולה. הכלי הזה חוסך את התהליך של הדפסת פלט בקונסולה ומאפשר לראות בדיוק מה קורה בג'אווהסקריפט.

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

```
<!doctype html>
<html>

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

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

</html>
```

אפשר לראות שיש כאן קישור ל-source.js, שנמצא באותה תיקייה של קובץ ה-HTML. התוכן ב-source יהיה משהו 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);
}
```

כדאי להריץ את הפרויקט פעם אחת בדפדפן על מנת לוודא תקינות. אם הכול תקין, הבה נפתח את הדיבאגר. בכרום, לחצו על F12 אם אתם עובדים בחלונות או על Cmd + Shift + i אם אתם במק. עכשיו מִצאו את לשונית ה-sources:

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

במקרה הזה העץ מצומצם מאוד כיוון שהוא כולל תיקייה אחת בלבד. אם העץ מורכב, אפשר ללחוץ על Ctrl + p (או Cmd + p במק) ולהקליד את שם הקובץ.

הדיבאגר עובד באופן עקרוני עם נקודות עצירה (באנגלית breakpoints). אפשר לבחור נקודה (או כמה נקודות) בסקריפט שבה הוא יעצור ויאפשר לבחון את מה שקורה. הבה נראה! נלחץ על השורה השלישית, ממש היכן שנמצאת לולאת for of. מייד יופיע סימון כחול המאשר שנקודת העצירה קיימת:

עכשיו חייבים להפעיל את הסקריפט מחדש. את זה עושים על ידי טעינה של הדף שוב. שימו לב שבהרבה מקרים לא צריך להפעיל את הסקריפט מחדש. אם מציבים את נקודת העצירה באירוע שמופעל בלחיצה על קליק, אין צורך

בהפעלה מחדש. כאן מדובר במקרה נדיר כיוון שהסקריפט הזה כבר רץ ולא ירוץ אם לא נטען את הדף מחדש. טעינה מחדש של הדף לא תציג שוב את התוכן כרגיל, אלא יופיע לפתע דף אפור ובו אינדיקציה שהדיבאגר מופעל:

בדיבאגר עצמו תוכלו לראות שעצרתם בנקודת העצירה:

מצד ימין אפשר לראות פרטים על המצב הנוכחי:

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

אם עוברים לקונסולה, אפשר להקליד פקודות שיעבדו בנקודת הזמן הזו של הסקריפט.

מעל דיווח המצב יש שורת פקדים שבהם אפשר להפעיל את הדיבאגר:

בלחצן הראשון אפשר להמשיך להריץ את הסקריפט עד נקודת העצירה הבאה, אם יש כזו. אם אין כזו, הסקריפט ירוץ עד הסיום. בלחצן השני אפשר לעשות step over – כלומר להריץ את הסקריפט לשורה הבאה. הלחצן השלישי והרביעי, stepin\out, שמורים לכניסה אל פונקציות ולא נדון 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:

עכשיו יש בסקופ שני חלקים – גם זה של הבלוק, כלומר של ה-for of. אפשר לראות שיש את ה-el, אבל הוא עדיין לא מוגדר. הוא יהיה מוגדר רק בשורה הבאה. בואו נתקדם אליה באמצעות לחיצה שנייה על הפקד step over או על F10:

כעת אפשר לראות שה-el הוא אלמנט מסוג p. אם לוחצים על החץ הקטן שלצד ה-el אפשר לראות את האובייקט של ה-DOM במלוא תפארתו, לצד כל התכונות המיוחדות של ה-DOM והתכונות שבאות יחד עם הגדרתו כאובייקט, וזה המון.

בגדול, אין פה יותר מדי, אך מדובר באחד הכלים החשובים ביותר למתכנתים וכדאי מאוד ללמוד להשתמש בו – ואז להשתמש בו - כבר בהתחלה. היכולת המרהיבה של הדיבאגר לסייע במציאת תקלות היא חשובה מאוד, והוא גם נותן תמונה מלאה על מה שמתרחש בקוד – כולל המשתנים המופיעים בסקופ או בקלוז'רים השונים (ואם שכחתם מה זה, חזרו לפרק על הפונקציות).

שימו לב: מטבע הדברים עיצוב של דפדפנים משתנה כל הזמן וייתכן שצילומי המסך המופיעים פה לא מעודכנים מספיק. אם כן, לא צריך להילחץ – העיקרון הוא אותו עיקרון גם אם ללשונית קוראים בשם אחר או אם האייקון נראה קצת אחרת.

אובייקטים גלובליים ואובייקטים מובנים

הראיתי בכמה הזדמנויות שקיימים אובייקטים שיש גישה אליהם. הדוגמה הטובה ביותר היא document. על אף ש-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 הזמינים לג'אווהסקריפט. ב"אובייקטים גלובליים" אני מתכוון לאובייקטים שמוגדרים ברמה הגלובלית ומשם מחלחלים לכל סקופ:

```
console.log('I am global');
(() => {
    console.log('I am inside my scope');
    const myArray = [1, 2, 3, 4];
    for (member of myArray) {
        console.log(member);
    }
})();
```

בדוגמת הקוד הזו, למשל, console הוא אובייקט שזמין מייד ברמה הגלובלית, בסקופ של הפונקציה האנונימית המריצה את עצמה (שלמדנו עליה בפרק על הפונקציות) וגם בסקופ של לולאת for of. אם דורסים את console ברמה הגלובלית ויוצרים אותו מחדש כאובייקט עם פונקציית log ריקה, ה-console.log לא יעבוד:

```
console.log('I am global'); //Only this will work

(() => {
    let console = {
        log: () => { };
    }
    console.log('I am inside my scope');
    const myArray = [1, 2, 3, 4];
    for (member of myArray) {
```

```
        console.log(member);
    }
})();
```

בדוגמה שלעיל, למשל, דורסים בתוך הסקופ של הפונקציה האנונימית את console. כיוון שלא מדובר במילה שמורה אלא באובייקט גלובלי, אפשר לעשות את זה בקלות. זה אומר שכל console.log בסקופ של הפונקציה האנונימית וכל ה-console.log שהוצבו באותו סקופ באמצעות closure (שגם עליו למדנו בפרק על הפונקציות), לא יעבודו כיוון שפונקציית log תהיה ריקה ב-console. כלומר, הטקסט שנעביר ב-console.log פשוט לא יוצג.
כל ההקדמה הארוכה הזו נועדה להסביר שיש הבדל משמעותי בין מילה שמורה, שהיא עצם מעצמותיה של ג'אווהסקריפט, לבין אובייקטים גלובליים שהם מכובדים וחשובים, אך עדיין רק אובייקטים שמצייתים לכללי השפה.
יש אובייקטים גלובליים שזמינים אך ורק לג'אווהסקריפט שעובד בסביבת HTML, כמו document שהכרנו קודם, window, Screen, URL וכו', ויש כאלו שזמינים אך ורק לג'אווהסקריפט שעובד בסביבת שרת, כמו process למשל. אובייקטים גלובליים רבים זמינים לכל ג'אווהסקריפט שהוא, כמו console. כל האובייקטים האלו נקראים Web APIs.

לכל אובייקט גלובלי יש תכונות או מתודות. על מנת לבחון אותן, אפשר לחפש אותן בתיעוד ברשת או פשוט להסתכל בקונסולה. בדוגמה הבאה פותחים את כלי המפתחים בכרום, עוברים לקונסולה, מקלידים console ולוחצים על Enter. בבת אחת רואים את כל המתודות שיש לקונסולה.

```
> console
<· ▼Object {debug: function, error: function, info: function, log: function, warn: function…}
    ▶assert: function assert()
    ▶clear: function clear()
    ▶count: function count()
    ▶debug: function debug()
    ▶dir: function dir()
    ▶dirxml: function dirxml()
    ▶error: function error()
    ▶group: function group()
    ▶groupCollapsed: function groupCollapsed()
    ▶groupEnd: function groupEnd()
    ▶info: function info()
    ▶log: function log()
    ▶markTimeline: function markTimeline()
     memory: (...)
    ▶profile: function profile()
    ▶profileEnd: function profileEnd()
    ▶table: function table()
    ▶time: function time()
    ▶timeEnd: function timeEnd()
    ▶timeStamp: function timeStamp()
```

אפשר לראות שיש שם error, למשל. ואכן, אם כותבים בקוד:

```
console.error('This is error!');
```

רואים שהקונסולה שולחת את הטקסט על רקע אדום כשגיאה. אם משתמשים ב-console.info רואים שמתקבלות התרעות בגוון אחר.

כאמור, ה-Web API מאפשר לקבל המון מידע מהתקנים שונים. אחד האובייקטים השימושיים ביותר הוא אובייקט window, שיש לו תכונות ומתודות 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 כאלו, וצריך רק לזכור שהם לא מגיעים משום מקום אלא מוזרקים 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: Mozilla Developer Network (MDN), שם יש פירוט מקיף ביותר על כל האובייקטים. עם כל שינוי והתקדמות של הדפדפנים נוספים אובייקטים. 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 בפיד המגיע ממנה 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 שכאן האובייקטים לא מוזרקים מהדפדפן או מסביבת העבודה אלא מוגדרים בתוכניות של השפה עצמה, כלומר הם חלק מהשפה. נשמע אולי שמדובר בהבדל דק, אך הוא משמעותי מאוד. אובייקט document מגיע מהדפדפן. הוא לא חלק רשמי מהשפה אלא סוג של API שהדפדפן מיישם. אובייקט מובנה מסוג Math, למשל, הוא אובייקט שמגיע מהשפה עצמה ומופיע בהגדרות שלה. בעוד אובייקט גלובלי כמו console.error יכול להיות מיושם בדפדפן כרום בצורה אחת, בפיירפוקס בצורה אחרת ובסביבת שרת בצורה שונה, אובייקט מובנה Math.PI שמחזיר את ערך הפאי מתנהג באופן זהה לחלוטין בלי שום קשר לדפדפן או אם ג'אווהסקריפט רצה בסביבת שרת. חלק מהאובייקטים הם אובייקטים של ממש וחלקם ממומשים כפונקציה (שגם היא אובייקט).

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

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

בהמשך הפרק נעבור על כמה אובייקטים מובנים חשובים.

parseInt

הפונקציה הגלובלית parseInt ממירה מחרוזת טקסט למספר. הפונקציה מקבלת שני ארגומנטים. הראשון הוא מחרוזת הטקסט והשני הוא הבסיס. ההמלצה הרשמית היא להשתמש בשני הארגומנטים, אבל בפועל צריך רק ארגומנט אחד. כך זה עובד:

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

אפשר לראות שהארגומנט הראשון הוא מחרוזת טקסט, אבל מייד אחרי שמעבירים אותו ב-parseInt רואים בקונסולה ש-convertedNumber הוא מספר. הפונקציה תעבוד גם אם יש המון רווחים:

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

אבל אם היא מקבלת משהו שהיא לא יכולה להתמודד איתו, 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 parseInt דווקא עובדת. אם היא נתקלת במספר שיש אחריו טקסט, היא נותנת 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 הזה ומריצה אותו כאילו מדובר בג'אווהסקריפט. כלומר היא מעריכה אותו (evaluation) כאילו הוא ג'אווהסקריפט ומחזירה את התוצאה.

בדוגמה הזו למשל לוקחים את מחרוזת הטקסט 1 + 4 ומריצים אותה כאילו היא ג'אווהסקריפט. התוצאה של כזה דבר היא 5, וזה בדיוק מה שמתקבל:

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

אפשר לקחת את זה רחוק יותר ולהכניס לתוך מחרוזת הטקסט הגדרות של משתנים ממש:

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

פה למשל מציבים במחרוזת הטקסט מספר לתוך משתנה a שהוגדר בסקופ 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, ירוץ כמו ג'אווהסקריפט. כתיבה של משהו כזה:

```
let a;
let result = eval('a = "Hello";');
console.log(a); // Hello
```

דומה לכתיבה של משהו כזה:

```
let a;
a = "Hello";
console.log(a); // Hello
```

אחד המשפטים הכי נפוצים בקרב מתכנתים הוא eval is evil. באופן עקרוני צריך להשתמש ב-eval רק אם אתם יודעים באמת מה אתם עושים. הסיבה היא ענייני אבטחת מידע – בדרך כלל מחרוזת הטקסט מגיעה בדרך זו או אחרת ממשתמש, ומתן אפשרות למשתמש להריץ מה שהוא רוצה בקוד הוא פתח אדיר לבעיות אבטחה ולבעיות אחרות. לפיכך למדנו כאן את eval, אבל רק לצורך ההסבר מדוע מומלץ לא להשתמש בה לעולם...

Math

Math הוא אובייקט גלובלי שמאפשר לבצע פעולות הקשורות ל... מתמטיקה. אל תיפלו מהכיסא פה. יש לאובייקט לא מעט מתודות ותכונות, שאחת מהן (pi) הכרנו בתחילת הפרק. רוב המתכנתים פוגשים את האובייקט הזה כאשר הם רוצים ליצור מספר רנדומלי, למשל כשמציעים למשתמש שם משתמש חדש או כאשר מדמים משתמש, משחקים, אנימציות וכו'.

באמצעות math.random יוצרים מספר רנדומלי מ-0 (כולל 0) עד 1 (לא כולל 1). המספר הוא שבר עם 16 ספרות לאחר הנקודה. אתם מוזמנים להריץ את הקוד הזה ולבדוק:

```
let result = Math.random();
console.log(result);
```

האם המספר הזה רנדומלי? לא ממש, כיוון שהוא מבוסס על הזמן ועל האלגוריתם שלוקח את חלקיק הזמן ויוצר מספר רנדומלי. אבל התהליך הזה רנדומלי מספיק כדי שיתקבל מספר אקראי. נשאלת השאלה, איך לוקחים את המספר הזה והופכים אותו למספר הרנדומלי שרוצים לקבל – למשל מספר מ-1 עד 10. האמת היא שזה 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 ליצירת עיגול, שנקראת round. כך:

```
let result = Math.random();
result *= 10;
result = Math.round(result);
console.log(result);
```

בגרסאות ישנות יותר של ג'אווהסקריפט, לפני שהיה קיים האופרטור ** 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'אווהסקריפט מחושבים כמילישניות (אלפיות השנייה) ומותאמים לשעון של המחשב. בעידן המודרני, שבו השעונים מסונכרנים באופן אוטומטי עם הרשת, התאריך מדויק למדי כך. התאריך הוא מספר המילישניות שעברו מ-1.1.1970 והוא מוסכם על כל מערכות המחשוב כ-epoch time, הזמן שבו כביכול מערכת החלה היוניקס הראשונה לפעול. כמעט כל שפות התכנות המוכרות משתמשות במוסכמה הזו. חלקן סופרות את השניות שעברו וחלקן את המילישניות. בג'אווהסקריפט סופרים מילישניות.

כשנדרשים לחשב תאריכים בג'אווהסקריפט, צריך לחשב אותם עם Date. בניגוד לאובייקטים הגלובליים שלמדנו עד כה, Date הוא פונקציה בנאית שמאתחלים עם המילה השמורה new. על פונקציה בנאית ועל new למדנו בפרק על this ועל new. לשם תזכורת, new מחזיר אובייקט שלם. הפונקציה הבנאית Date מקבלת ארגומנט של הזמן שמעוניינים לחקור. כאשר אין ארגומנט, הזמן הוא ה-epoch time. ברגע שמאתחלים את Date, נוצר אובייקט שמכוון לתאריך האתחול של האובייקט – כלומר היום, ברגע הזה בדיוק.

הבה נדגים:

```
const dateObject = new Date();
let epochElapsedTime = dateObject.getTime();
console.log(epochElapsedTime);
```

כאן יוצרים date ללא ארגומנט, כלומר תאריך האתחול הוא 1.1.1970. משתמשים במתודה getTime כדי לקבל את מספר המילישניות שעברו מהתאריך שבו אותחל ה-date (במקרה הזה 1.1.1970) עד היום. נסו ותראו!

אם רוצים לאתחל את אובייקט ה-date לתאריך מסוים, אפשר להכניס ארגומנטים של תאריך. הארגומנט הראשון הוא שנה, השני חודש, השלישי יום, הרביעי שעה, החמישי דקה, השישי שנייה והשביעי והאחרון מילישנייה. אפשר גם להכניס רק ארגומנט של שנה וחודש, ואז האובייקט יאותחל ביום הראשון של החודש. שנה וחודש הם המינימום הנדרש, אבל אפשר להוסיף עוד ארגומנטים, עד המילישנייה. כאן למשל מאתחלים את האובייקט בתאריך המלא 16.09.1977, שעה 14:30, 20 שניות ו-500 מילישניות. אחרי שמאתחלים את האובייקט אפשר להשתמש ב-getTime כדי לבדוק כמה זמן עבר מהתאריך של ה-epoch time עד התאריך הזה. שזה 245,849,420,500 מילישניות בדיוק.

```
const dateObject = new Date(1977, 9, 16, 14, 30, 20, 500);
let myBirthdayElapsedTime = dateObject.getTime();
console.log(myBirthdayElapsedTime);
```

זה נראה מעט אבסטרקטי. הבה נראה שימוש אמיתי באפשרות הזאת. נניח שמקבלים את גילו של לקוח ורוצים לדעת בן כמה הוא בדיוק. ראשית יוצרים אובייקט עם יום הולדתו, ואז יוצרים אובייקט של התאריך הנוכחי ומחסירים ממנו את הזמן של יום הולדתו. זה אפשרי כי הזמן הוא במספר, מילישניות, משנת 1970:

```
const customerDateObject = new Date(1977, 9, 16);
const currentDaObject = new Date();
let gap = currentDaObject - customerDateObject;
console.log(gap);
```

יש כאן את אובייקט התאריך של הלקוח ואת אובייקט התאריך הנוכחי, ופשוט מחסירים אחד מהאחר. אם השנה עכשיו היא 2018 ונולדתם בשנת 1978, חישוב הגיל הוא פשוט: 2018-1978 = 40. כאן במקום בשנים סופרים במילישניות, אבל קשה לומר לבן אדם, "היי! אתה בן 1,305,505,055 מילישניות!" – צריך להשתמש בפורמט של בני אדם. פה נכנסת המתמטיקה לסיפור, ונעשה חישוב פשוט שבו מחלקים את הפער במילישניות במספר המילישניות שיש בשנה:

```
const customerDateObject = new Date(1977, 9, 16);
const currentDateObject = new Date();
let gap = currentDateObject - customerDateObject;
console.log(gap);
const years = gap / (365 * 24 * 60 * 60 * 1000);
console.log(`You are ${years} old!`);
```

הינה דרך החישוב:

1000	מילישניות בשנייה
60	שניות בדקה

60	דקות בשעה
24	שעות ביממה
365	ימים בשנה

מספר המילישניות בשנה הוא לפיכך: 1000X60X60X24X365, כלומר 31,536,000,000 מילישניות, שזה המון. אבל אם מחלקים את מספר המילישניות במספר הזה, מגלים את מספר השנים. מובן שיצא מספר עם הרבה ספרות אחרי הנקודה, אך בדיוק לשם כך קיים Math.round.

נהוג לשמור תאריכים כמעט תמיד כמספר המילישניות (או מספר השניות, ואז להכפיל באלף כשמגיעים 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 משרת ולעבוד איתם או לטעון או לשמור קובץ אם עובדים בג'אווהסקריפט בצד השרת. ג'אווהסקריפט בנויה לעבוד היטב עם פורמט נתונים שנקרא JSON, ראשי תיבות של JavaScript Object Notation. בגדול, הפורמט הזה הוא אובייקט ג'אווהסקריפטי שעליו למדנו כבר בעבר, אך בשינויים קלים:

1. מירכאות כפולות בלבד.
2. מירכאות סביב כל מחרוזת טקסט, גם סביב התכונות.
3. אין פסיק בסוף התכונה האחרונה.
4. אסור לכתוב הערות בתוך JSON.
5. אין מתודות.

למשל, הקוד הבא הוא קוד ג'אווהסקריפטי 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 מחרוזת טקסט וגם סביב כל התכונות ואין פסיק בסוף התכונה האחרונה.

כאמור, JSON אמור להיות אינטואיטיבי לכל מי שמכיר ג'אווהסקריפט מספיק זמן. למעט השינויים הקלים האלו, מדובר בפורמט פשוט וקל, וכמעט כל השפות בשרתים השונים עובדות איתו, מה שהופך אותו לממש נוח. אבל כדי לעבוד איתו צריך להמיר אובייקט ג'אווהסקריפטי ל-JSON ולהפך. עושים את זה בקלות באמצעות אובייקט JSON הגלובלי, שיש לו שתי מתודות – האחת הופכת אובייקט ג'אווהסקריפט ל-JSON תקין והאחרת ממירה JSON תקין לאובייקט ג'אווהסקריפטי 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 מחרוזת, ומכאן השם של המתודה: stringify – להפוך למחרוזת. את האובייקט הזה אפשר לשגר לשרת באמצעות פקודת AJAX, לשמור על השרת או להעביר אותו בכל צורה אחרת. הוא נחשב מחרוזת טקסט לכל דבר. בצד השרת, כל שפה יודעת להתמודד עם JSON, שהפך דה פקטו לפורמט המרכזי של הרשת.

שימו לב שאובייקט JSON הוא מחרוזת טקסט לכל דבר. כלומר דבר כזה:

```
let someObject = {
    name: 'Ran',
    birthdate: 245849420500
}
let JSONObject = JSON.stringify(someObject);
JSONObject.name = 'Moshe'; // Will get Uncaught SyntaxError: Unexpected token
o in JSON at position 1
```

לא יעבוד ויזרוק הודעת שגיאה. על מנת לעבוד עם מחרוזת טקסט שהיא אובייקט JSON כשר למהדרין, צריך להמיר אותה לג'אווהסקריפט. את זה עושים באמצעות JSON.parse באופן הבא:

```
const JSONObject = '{"name":"Ran","birthdate":245849420500}';
let regularObject = JSON.parse(JSONObject);
regularObject.name = 'Moshe';
console.log(regularObject); // Object {name: "Moshe", birthdate: 245849420500}
```

כאן היה אובייקט JSON שהוא בעצם מחרוזת טקסט. במקרה הזה הוא הוקלד ידנית, אבל במציאות הוא יגיע משרת, ממסד נתונים או בכל צורה אחרת. באמצעות JSON.parse הוא הומר לאובייקט רגיל שעליו אפשר לעבוד כרגיל, למשל להחליף את שם הלקוח ל-Moshe כפי שנעשה כאן.

setTimeout

מדובר בפונקציה גלובלית שמאפשרת לעכב את הקוד או לתזמן אותו. בדרך כלל בקוד רגיל אין בה שימוש, אבל בלא מעט דוגמאות משתמשים בקוד הזה. כמו כן, בקוד ג'אווהסקריפט שבודק UI יש חשיבות רבה לתזמון או לעיכוב, אבל בדרך כלל השימוש בפונקציה הזו נחשב לתכנות גרוע. אם התוכנה שלכם צריכה עיכוב (timeout), סימן שיש בעיה בסקריפט. אז למה לומדים את הפונקציה הזאת? בעיקר כי צריך אותה לשם לימוד קוד אסינכרוני בהמשך. נוסף על כך, זו פונקציה שכאמור נמצאת בדוגמאות רבות ברשת בכל מה שקשור לאספקטים מתקדמים יותר של ג'אווהסקריפט (כמו גנרטורים – שהם חלק מתקדם בשפה שלא נגיע אליו בספר הזה, אבל אפשר לגשת ל-MDN שהוזכר קודם כדי לבדוק), אחרת לא הייתה שום סיבה לכתוב עליה במיוחד. אז מומלץ להתוודע לסינטקס שלה ולהכיר אותה, אבל מאוד לא מומלץ להשתמש בה בקוד אמיתי בלי סיבה ממש-ממש טובה (סיבה טובה היא יצירת אנימציות או משחקים, למשל).

חמושים באזהרות האלו, בואו נלמד את הפונקציה של timeout. מדובר בפונקציה גלובלית שזמינה כמעט בכל הסביבות – דפדפן או סביבת שרת. איך היא עובדת? זו פונקציה שמקבלת שני ארגומנטים, פונקציה אנונימית (המועברת כפונקציית חץ) שרצה מייד אחרי זמן העיכוב, וזמן העיכוב במילישניות.

הינה הפונקציה:

```
console.log('start the code');
setTimeout(() => {
    console.log('After 5 seconds');
}, 5000);
console.log('end the code');
```

פונקציית החץ, שמועברת כארגומנט הראשון, תרוץ מייד לאחר שזמן העיכוב, שמועבר כארגומנט השני, יסתיים. הקוד לפיכך ידפיס את מה שיש בקונסולה בשורה הראשונה – "start the code", ויריץ את פונקציית העיכוב שתחכה חמש שניות. הסקריפט לא יעצור ויריץ את השורה האחרונה "end of code". רק לאחר חמש שניות, שהן 5,000 מילישניות, יוצג "After 5 seconds".

```
start the code
end the code
undefined
After 5 seconds
```

מה שחשוב להבין הוא ששאר הסקריפט רץ, וברגע שהזמן שהוקצב יסתיים, מה שהועבר בפונקציית החץ ירוץ מייד. לא כל הסקריפט עוצר; ההרצה של שאר הסקריפט מתקיימת כרגיל.

פונקציית setTimeout מחזירה מספר חיובי שבו אפשר להשתמש כדי לזהות אותה בהמשך הקוד. לאיזה צורך זה שימושי? אם רוצים מסיבה כלשהי בהמשך להשמיד את ה-setTimeout. ההשמדה של setTimeout נעשית באמצעות פונקציה מובנית אחרת שנקראת clearTimeout, והיא מקבלת כארגומנט את המספר המזהה של ה-setTimeout שאותו 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 בדיוק כמו בדוגמה הקודמת, אבל הפעם מקבלים את ה-ID שלה ומשתמשים בו כדי להרוג מיידית את ה-setTimeout, שלעולם לא יתבצע.

תרגיל:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד 100 (לא כולל 100):

```
function getRandom() {
    let result = Math.random();
    result *= 100;
    result = Math.round(result);
    return result;
}
const randomNumber = getRandom();
```

```
console.log(randomNumber);
```

הסבר:

יוצרים פונקציה עם שם כלשהו. ראשית יוצרים מספר רנדומלי באמצעות Math.random. המספר הוא בין 0 ל-1 (לא כולל 1). מכפילים את התוצאה ב-100. כך למשל אם המספר הרנדומלי שיצא הוא 0.512, התוצאה תהיה 51.2. כיוון שחייבים להחזיר מספרים שלמים, לוקחים את המספר, מעגלים אותו באמצעות Math.round ומחזירים אותו כמו כל פונקציה רגילה.

תרגיל:

צרו פונקציה שמחזירה מספר רנדומלי, עגול, מ-1 עד הארגומנט שהפונקציה מקבלת:

```
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 שהטווח 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:

```
function verify21(costumerYear, costumerMonth) {
    const currentDate = new Date();
    const costumDate = new Date(costumerYear, costumerMonth);
    const age = currentDate - costumDate;
    const years = age / (1000 * 60 * 60 * 24 * 365);
    if (years > 21) {
        return true;
    } else {
        return false;
    }
}
console.log(verify21(1977, 9)); // true
console.log(verify21(2017, 1)); // false
```

הסבר:

יוצרים פונקציה שמקבלת שני ארגומנטים, הראשון של שנה והשני של חודש. ראשית יוצרים אובייקט תאריך של התאריך הנוכחי, ללא ארגומנט. שנית יוצרים אובייקט תאריך של שנת הלידה והחודש. כל מה שנותר לעשות הוא לחשב את הפער. הוא מתקבל במילישניות, וכדי להסב אותו לשנים מחלקים אותו במספר המילישניות שיש בשנה. עכשיו יש את מספר השנים שהוא הפער בין גיל המשתמש לתאריך הנוכחי. אם מספר השנים גדול מ-21 מחזירים true, ואם מספר השנים קטן מ-21 מחזירים false. איזה יופי.

תרגיל:

צרו פונקציה שמקבלת מספר X ומדפיסה בקונסולה את המשפט "רצתי לאחר X שניות".

הסבר:

```
function delayFunction(int) {
    const delaySeconds = int * 1000;
    setTimeout(() => {
        console.log(`After ${int} seconds`);
    }, delaySeconds);
}
delayFunction(2);
```

יוצרים פונקציה בשם delayFunction שיש לה ארגומנט אחד. את הארגומנט הזה מכפילים ב-1,000 כדי לקבל מספר במילישניות ומכניסים אותו לקבוע delaySeconds. קוראים לפונקציה הגלובלית setTimeout. הפונקציה הזו קיימת בכל ג'אווהסקריפט כחלק מהשפה, לא צריך ליצור אותה. היא מקבלת שני ארגומנטים. הראשון הוא פונקציה שרצה אחרי מספר המילישניות שמעבירים בארגומנט השני. בארגומנט הראשון מעבירים פונקציה אנונימית מסוג חץ, שעליה למדנו בפרק על הפונקציות, שכל מה שהיא עושה הוא להדפיס בקונסולה "אחרי X שניות". שימו לב שההדפסה נעשית עם תבנית שגם עליה כבר למדנו, בפרק על תבנית טקסט.

בארגומנט השני שהיא מקבלת מעבירים את delaySeconds – מספר המילישניות. זה הכול. 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 לתוך 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 אימייל, אפשר להשתמש בביטוי רגולרי פשוט

כדי להבין אם מדובר באימייל או לא. הדרך הכי טובה למצוא מילים, משפטים או כל דבר אחר בטקסט היא באמצעות ביטוי רגולרי.

קשה לתכנת בכל שפה ובמיוחד בג'אווהסקריפט ולא להיתקל בביטוי רגולרי. למשל, אם משתמש בוחר סיסמה שלא מצייתת לכלל מסוים או לחיפוש והחלפה של טקסט. בכל פעם שעושים פעולה מורכבת בטקסט או מנתחים אותו, צריך להשתמש בביטוי רגולרי. ביטויים רגולריים נמצאים בהמון שפות ולא רק בג'אווהסקריפט.

בג'אווהסקריפט, ביטוי רגולרי הוא מוקף בסלאשים (לוכסנים) בלבד, אף על פי שהוא מסוג אובייקט, וככה אפשר לזהות אותו. ביטוי רגולרי יכול להיראות כך:

```
const regularExpression = /abc/;
```

כל מה שיש בין הסלאשים הוא הביטוי הרגולרי. מייד אחרי הסלאש האחרון יש אופציות (modifiers). למשל:

```
const regularExpression = /abc/ig;
```

מקובל מאוד להציב מייד אחרי הביטוי הרגולרי אופציות. במקרה הזה האותיות i ו-g מסמלות אופציות – וגם הביטוי הזה תקין לחלוטין. יש לא מעט אפשרויות שונות שאפשר להצמיד. הינה טבלה קצרה של אפשרויות לדוגמה:

התו המייצג את ה-modifier	מה הוא עושה
i	הביטוי הרגולרי לא יתייחס להבדל בין אותיות ראשיות לרגילות. abc יהיה זהה ל-ABC או ל-ABc וכך הלאה
m	הביטוי הרגולרי יתחשב בריבוי שורות בסימון של התחלה או סוף
g	כאשר יש כמה התאמות בחיפוש, הביטוי הרגולרי יחזיר את כולן במקום אחת בלבד

אפשר ליצור ביטוי רגולרי גם באמצעות פונקציה בנאית:

```
const regularExpression = new RegExp('abc', 'ig');
```

אם משתמשים בפוקנציה בנאית, אפשר להשתמש במשתנים בביטוי הרגולרי.

כל ההדגמות להלן ייעשו בדרך הראשונה והפשוטה יותר, אבל אתם מוזמנים לנסות גם את הדרך השנייה, באמצעות הפונקציה הבנאית, כאשר אתם מתרגלים.

הביטוי הרגולרי הראשון הוא /abc/ שאומר פשוטו כמשמעו – כל ביטוי שכולל את האותיות abc ללא רווח ביניהן וללא אותיות אחרות, כלומר המחרוזת הפשוטה abc. נראה איך ג'אווהסקריפט עובדת עם ביטויים רגולריים ואחר

כך נעמיק לתוך הביטויים ומשמעותם, ולשם כך הביטוי הבסיסי הזה יספיק. כאשר יוצרים ביטוי רגולרי, יש לו באופן אוטומטי את המתודות של הביטוי הרגולרי (בדיוק כמו שלמערך יש את המתודות של forEach למשל).

המתודה הראשונה שנלמד היא test, שמאפשרת לבדוק שהביטוי הרגולרי נמצא במחרוזת טקסט כלשהי. נניח שרוצים לבדוק אם יש בביטוי abcdefg את הביטוי הרגולרי /abc/:

```
const regularExpression = /abc/;
const textString = 'abcdefg';
const result = regularExpression.test(textString);
console.log(result); //true
```

ראשית, יוצרים משתנה המכיל ביטוי רגולרי. הביטוי הרגולרי מוקף בסלאשים בלבד, ללא מירכאות. זה מראה שהוא ביטוי רגולרי. שנית, יוצרים מחרוזת טקסט שאותה בודקים. בחיים האמיתיים, מן הסתם, מחרוזת הטקסט הזו תועבר על ידי המשתמש או על ידי שירות חיצוני כלשהו, אבל כאן, לשם הדוגמה, נכתוב אותה לתוך משתנה קבוע.

כיוון שהמשתנה regularExpression הוא ביטוי רגולרי, יהיו לו מתודות של ביטוי כזה. מתודת test מקבלת כארגומנט מחרוזת טקסט ואם הביטוי הרגולרי נמצא במחרוזת הטקסט, מחזירה true. זה בדיוק מה שהיא עושה כאן, כיוון שיש abc ב-**abc**defg.

אם ננסה את זה במחרוזת אחרת, שלא כוללת את הצירוף abc, יתקבל false:

```
const regularExpression = /abc/;
const textString = 'Nothing is here';
const result = regularExpression.test(textString);
console.log(result); //false
```

מובן שביטוי רגולרי יכול להיות הרבה יותר מורכב מזה. אפשר לסמן, למשל, שרוצים שהביטוי יהיה נכון רק אם מחרוזת הטקסט מתחילה ב-abc. לדוגמה, הביטוי abcdef יעבור את הבדיקה, אבל הביטוי defabc לא יעבור. את זה עושים באמצעות התו ^ שמשמעותו "התחלה", אם הוא לא נמצא בתוך סוגריים מרובעים. הביטוי /^abc/ יהיה נכון רק לגבי ביטויים כאלו:

```
const regularExpression = /^abc/;
const textString = 'abcdef';
const result = regularExpression.test(textString);
console.log(result); //true
```

אתם מוזמנים לקחת את הדוגמה הזו ולנסות לשנות את מחרוזת הטקסט לכל דבר אחר שמכיל abc אבל לא בתחילת המחרוזת. הביטוי הרגולרי לא יהיה נכון ויתקבל false:

```
const regularExpression = /^abc/;
const textString = 'defabcdef';
const result = regularExpression.test(textString);
console.log(result); //false
```

כמו שיש התחלה, יש גם סוף. התו $ מסמל את סוף המשפט שהביטוי הרגולרי מסתיים בו:

```
const regularExpression = /abc$/;
const textString = 'defabc';
const result = regularExpression.test(textString);
console.log(result); //true
```

הביטוי הרגולרי הזה 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); //false
```

כל ביטוי אחר לא יתאים, כי תחמנו את abc בהתחלה ובסוף.

אפשר גם לציין אותיות חופשיות (wildcards). למשל, אם רוצים ביטוי שמתחיל ב-a ומסתיים ב-bc ובאמצע יש רק אות אחת, אפשר להשתמש בביטוי "." (נקודה), שמשמעותו תו אחד 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. 1 או יותר ממה? ממה שמגיע לפניו. למשל:

+a – הכוונה היא לאות a אחת או יותר.

+. – הכוונה היא לכל סימן אחד או יותר.

כלומר הביטוי:

^a.+bc$

תקף גם ל-a1bc וגם ל-a12bc וגם ל-a123bc וכך הלאה. ולא רק מספרים נכללים כאן, אלא גם אותיות, רווחים, סימנים מיוחדים וכל דבר אחר. נקודה (.) היא סוג תו כללי.

אפשר גם לציין את סוג התו. למשל, אם רוצים רק מספרים, ללא אותיות או סימנים מיוחדים, לא משתמשים בביטוי שמשמעותו "כל אות" אלא בביטוי:

^a[0-9]+bc$

מה הביטוי הזה מציין? שהטקסט חייב להתחיל ב-a ולהסתיים ב-bc. באמצע יש מספר ([0-9]) אחד או יותר (זו משמעות ה-+).

אפשר להשתמש גם בתווים. למשל, כאן מאפשרים גם אותיות קטנות וגם מספרים:

^a[0-9a-z]+bc$

הביטוי הזה מתחיל ב-a ומייד אחריו מופיע הביטוי [0-9a-z], שמשמעותו כל אות בין 0 ל-9 או בין a ל-z, כלומר אותיות קטנות ומספרים. כמה פעמים? "+" – כלומר פעם אחת או יותר.

אפשר להשתמש באיזה טווח שרוצים, למשל Z-A או 0-5.

לא חייבים להשתמש ב-+, יש עוד כמה ביטויי כמות:

* – כוכבית, 0 עד אינסוף.

? – סימן שאלה, 0 עד 1.

אפשר לסכם את הביטויים הרגולריים בטבלה הבאה:

סימן	**פירוש הסימן**
^	התחלה – מה שאמור להגיע בתחילת הביטוי
$	סוף – מה שיש בסוף הביטוי
.	כל תו
[]	בתוך הסוגריים המרובעים יהיו טווחים של אותיות
+	אחד לפחות מהביטוי המופיע לפניו
?	אפס או ביטוי אחד מהביטוי המופיע לפניו
*	אפס או יותר מהביטוי המופיע לפניו

תחום הביטויים הרגולריים הוא עולם ומלואו, וספרים שלמים מוקדשים לו. חשוב לדעת שיש דבר כזה ושמשתמשים בו לניתוח טקסט. שליטה בביטויים רגולריים היא מעבר לתחומו של ספר זה, אבל אתם חייבים לדעת שיש דבר כזה ולשלוט בו באופן בסיסי ביותר, מכיוון שמדובר בחלק משפת JS שמתכנתים משתמשים בו.

בעברית הביטויים הרגולריים קשים יותר. אי-אפשר לכתוב טקסט בעברית אלא צריך להשתמש בייצוג היוניקודי שלו. יוניקוד הוא הקידוד שבו מקובל להשתמש בשנים האחרונות והוא מונע את כל הג'יבריש. כל צורות הכתב בעולם נמצאות ביוניקוד, וכל אות ואות בכל מערכת כתב מיוצגת על ידי מספר סידורי. כך למשל האות א' ביוניקוד היא U+05D0, וזה מה שצריך לכתוב, וכן, יש גם יוניקוד ל... ניקוד. כאמור, הנושא הזה לא מכוסה בספר הזה.

תרגיל:

נתונה מחרוזת הטקסט "ahla bahla". כתבו ביטוי רגולרי הבודק אם היא כוללת את המילה ahla.

```
const regularExpression = /ahla/;
const textString = 'ahla bahla';
const result = regularExpression.test(textString);
console.log(result); //true
```

הסבר:

יוצרים קבוע בשם regularExpression ומכניסים אליו את הביטוי הרגולרי ahla, שמשמעותו מחרוזת טקסט של ahla. הביטוי הרגולרי מוקף ב-/. כך מסמנים ביטויים רגולריים. לאחר מכן יוצרים קבוע שבו יש את מחרוזת הטקסט שאותה בודקים.

השלב הבא הוא להשתמש במתודת test. כיוון שלכל ביטוי רגולרי יש אותה באופן אוטומטי, יש את המתודה הזו גם ל-regularExpression שמכיל ביטוי רגולרי. המתודה הזו מקבלת טקסט כארגומנט, וזה בדיוק מה שהועבר לה.

תרגיל:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אות גדולה אחת לפחות.

```
const regularExpression = /[A-Z]+/;
const textString = 'passworD';
const result = regularExpression.test(textString);
console.log(result); //true
```

הסבר:

הביטוי הרגולרי הוא מה שחשוב פה. הסוגריים המרובעים מציינים טווח: A-Z. [A-Z] משמעו כל אות גדולה שהיא. כמה אותיות גדולות אנחנו רוצים? אחת או יותר. לפיכך משתשים ב-+. הביטוי:

[A-Z]+

משמעו אות אחת לפחות מהטווח Z-A, כלומר 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 ביטוי רגולרי, על מחרוזת הטקסט הזו. המתודה מחזירה true אם המחרוזת מצייתת לביטוי הרגולרי. כיוון שיש בה אות אחת גדולה, היא מצייתת ויתקבל true.

תרגיל:

נתונה סיסמה. כתבו ביטוי רגולרי הבודק אם היא כוללת אות גדולה אחת לפחות ומספר אחד לפחות. כשהאות היא לפני המספר. למשל passworD1.

```
const regularExpression = /[A-Z] [0-9] /;
const textString = 'passworD1';
const result = regularExpression.test(textString);
console.log(result); //true
```

הסבר:

כמו בתרגיל הקודם, עיקר הבעיה פה הוא לנסח את הביטוי הרגולרי. במקרה הזה:

[A-Z]

הטווח הוא Z-A (כלומר אות גדולה).

[0-9]

הטווח הוא 9-0 (כלומר מספר).

הביטוי המשולב משמעו מחרוזת טקסט שיש בה אות גדולה אחת לפחות ומייד אחריה מספר אחד. אחרי שמנסחים את הביטוי הרגולרי, יוצרים אותו בג'אווהסקריפט בין // ומכניסים אותו לקבוע regularExpression. לקבוע זה, מהרגע שיש בו ביטוי רגולרי, יש את מתודת test שמקבלת כארגומנט מחרוזת טקסט, במקרה הזה את הסיסמה שהוזנה, ובודקת אם היא מצייתת לתנאי.

טיפול בשגיאות

כמתכנתי ג'אווהסקריפט מנוסים, בוודאי יצא לכם לראות שגיאות בקונסולה במהלך העבודה. השגיאות האלו נוצרות בדרך כלל אם טועים בסינטקס, למשל בקריאה למשתנה שלא ממש הוגדר. אם למשל כותבים:

```
console.log(myVar);
```

ולא טורחים להגדיר קודם את myVar, מקבלים שגיאה כזו:

Uncaught ReferenceError: myVar is not defined

אם משתמשים בדפדפן, רואים את השגיאה בקונסולה באופן הבא:

```
▶Uncaught ReferenceError: myVar is not defined
```

הודעות השגיאה מכילה מידע גם על השגיאה וגם על המקום שלה.

עד עכשיו התייחסתי אל שגיאות כאל משהו בלתי רצוי, אבל הרבה פעמים כן רוצים שגיאות. שגיאות והטיפול בהן הם חלק מכל מערכת ופונקציה חשובה. אם למשל כותבים פונקציה שבודקת סיסמה והמשתמש מכניס סיסמה בעברית בלבד, זו שגיאה שחייבים לתפוס ולטפל בה. אם כותבים פונקציה שקוראת קובץ אבל הקובץ לא שם, חייבים לייצר שגיאה עבור מי שכתב את הפונקציה.

יצירת שגיאה היא ממש פשוטה. שגיאה היא אובייקט שנוצר מפונקציה בנאית ויוצרים אותו כך:

```
const myError = new Error('Oy Vey');
throw myError;
```

ראשית יוצרים את השגיאה עם הטקסט המיוחד לה. במקרה הזה "Oy Vey". אחרי שיש אובייקט שגיאה אפשר בכל שלב של הסקריפט לזרוק אותו. ברגע שיש שגיאה, כל קוד שירוץ אחריה יפסיק לרוץ מיידית, ממש כמו שגיאת הרצה מהסוג המוכר (והאהוב?), כלומר הקוד הזה:

```
function testMe() {
    const myError = new Error('Oy Vey');
    throw myError;
}
testMe();
console.log('Hello'); // Will never run
```

ההדפסה של Hello בקונסולה לא תרוץ לעולם, אלא אם כן מדובר במקרה נדיר שבו חייבים להפיל מערכת. בדרך כלל יוצרים שגיאה כחלק מהניסיון לנהל משהו. שגיאה שמפילה את כל הסקריפט לא ממש תעזור כאן. בגלל זה לרוב זורקים את השגיאות לתוך בלוק שנקרא try-catch.

בלוק של try-catch הוא בעצם דרך לזרוק שגיאות ולטפל בהן בלי השבתה של הסקריפט כולו. בגדול הוא מורכב מהמילה השמורה try שאחריה סוגריים מסולסלים, ובהם הקוד רץ. אם לא תהיה שגיאה – טוב ויפה. אם תהיה שגיאה, קוד השגיאה ייכנס לבלוק catch, שבו יש את ארגומנט השגיאה:

```
function testMe() {
    const myError = new Error('Oy Vey');
    throw myError;
}

try {
```

```
    testMe();
} catch (error) {
    console.log(error);
}
console.log('Hello');
```

בבלוק של ה-try קוראים לפונקציה. אם הפונקציה רצה ללא שגיאה, הכול תקין והקוד מתנהל כרגיל. אם יש שגיאה, הבלוק של catch נכנס לפעולה. כל קוד אחר שמחוץ ל-try-catch רץ כרגיל והסקריפט לא קורס.

במקרה הזה, testMe תמיד זורקת שגיאה. בחרתי להדפיס את השגיאה בבלוק ה-catch. ההדפסה של ה-Hello, מחוץ ל-try-catch, תודפס כרגיל.

כל קוד שעטוף ב-try ויזרוק שגיאה, אפילו שגיאת קומפילציה, לא יפריע למהלך התקין של הקוד ויהיה אפשר לנהל את דרך הטיפול בשגיאה באמצעות ה-catch. האם תדפיסו הודעה למשתמש? האם תנסו להריץ מחדש את הפונקציה? הכול אפשרי.

בדרך כלל מקובל שמודולים זורקים שגיאות וסומכים על מי שמפעיל אותם שיטפל בהן לפי מיטב הבנתו. נניח למשל שיש אובייקט כזה:

```
const passwordTester = function (password) {
    if (/[A-Z]+/.test(password) === false) {
        const error = new Error('No capital in password');
        throw error;
    }
    if (password.length < 8) {
        const error = new Error('Password is shorter than 8');
        throw error;
    }
    this.password = password;
```

```
};
```

האובייקט בודק סיסמאות והוא אובייקט מורכב שנבנה בעזרת פונקציה בנאית, מהסוג שכבר למדנו עליו. במקרה הזה, בתוך האובייקט עצמו בודקים את הסיסמה באמצעות ביטוי רגולרי. אם הסיסמה לא מכילה אות גדולה, זורקים שגיאה מסוג אחד. אם הסיסמה קצרה מדי, זורקים שגיאה מסוג אחר. כל זריקה של הסיסמה עוצרת את הסקריפט בתוך האובייקט.

אם מיישמים את האובייקט כמו שצריך, צריך לעטוף אותו ב-try וב-catch:

```
try {
    new passwordTester('12345678');
} catch (error) {
    console.log(error);
}
```

בתוך ה-catch אפשר לנהל את השגיאה, כלומר להדפיס אותה למשתמש, למשל. מקבלים את האובייקט של השגיאה, והטקסט מופיע בתוך אובייקט השגיאה תחת השדה message. אפשר להדפיס את הטקסט של השגיאה, כמובן, או לעשות בו מה שרוצים.

כמעט לכל מודול חיצוני או אפילו פנימי בג'אווהסקריפט יש מדיניות שגיאות שאי-אפשר לתפוס ולנהל. לא את כל השגיאות אפשר לנהל בצורה ממש טובה, והבחירה מה לעשות היא של המתכנת; לפעמים מעדיפים להשבית את כל הסקריפט, לפעמים מציגים הודעת 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. ככה רק מונעים הדפסת שגיאות, אבל מפספסים את הנקודות החשובות של ניהול שגיאות – ניהול השגיאה הוא חשוב אם יודעים מאיפה היא מגיעה ומה לעשות איתה. אם הסקריפט מורכב, בשלב העליון (try-catch) אי-אפשר לעשות כלום עם השגיאות וקשה לדעת מהיכן הן הגיעו. לפיכך, מומלץ שתעטפו את חלקי הקוד שלכם ב-try-catch רק אם יש לכם תוכנית מסודרת בנוגע למה שצריך לעשות במקרה של קריסה או במקומות שאתם צופים שבהם יעופו שגיאות.

finally

כתוספת ל-try-catch יש את finally. זהו בלוק קוד נוסף שאנו יכולים להוסיף לביטוי ה-try-catch והוא תמיד ירוץ אחריו, בין שה-try יעבוד ובין שה-catch:

```
const passwordTester = function (password) {
    if (/[A-Z]+/.test(password) === false) {
        const error = new Error('No capital in password');
```

```
        throw error;
    }
    if (password.length < 8) {
        const error = new Error('Password is shorter than 8');
        throw error;
    }
    this.password = password;
};
try {
    new passwordTester('12345678');
} catch (error) {
    console.log(error);
} finally {
    console.log('Try again!');
}
```

אם תריצו את הקוד הזה, תראו ש-Try again תמיד 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 אייקון 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:

```
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 להכביר מילים. יש כאן פונקציה שמקבלת שני ארגומנטים ובודקת במשפט תנאי פשוט אם אחד מהם הוא לא מספר. אם הוא לא מספר, יוצרים שגיאה וזורקים אותה. שתי השורות האלו 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 קומפילציה. ג'אווהסקריפט בהחלט יכולה לחבר מחרוזת טקסט ומספר (היא פשוט תמיר את המספר לטקסט), אבל במקרה הזה, הפונקציה לא תאפשר את 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:

```
function checkNegative(arg1) {
    if (arg1 < 0) {
        const numberError = new Error('Negative Number');
        throw numberError;
    }
}
let number = -1;
try {
    checkNegative(number);
} catch (e) {
    number = 0;
}
console.log(number); // 0
```

הסבר:

הפונקציה פשוטה למדי, היא מקבלת ארגומנט. אם הארגומנט הוא שלילי, זורקים שגיאה באמצעות:

```
const numberError = new Error('Negative Number');
throw numberError;
```

כדי לנהל את השגיאה, עוטפים את הקוד ב-try-catch. אם יש שגיאה, מאפסים את המשתנה. זה הכול. מה שיש בתוך try יזרוק את השגיאה אך לא יפריע למהלך התקין של שאר הסקריפט. מה שיש בתוך catch ינהל את השגיאה. במקרה הזה דרך ניהול השגיאה היא לאפס את המשתנה.

מבני נתונים מסוג Map ו-Set

בפרק על מערכים למדנו שאפשר לאחסן בהם נתונים. אבל יש בעיה – במערכים, המפתח הוא מספר ואי-אפשר לקבוע אותו. אפשר לאחסן מידע באובייקטים, אך לא נוח לעשות זאת כיוון שקשה לעבוד עם איטרציות באובייקטים, קשה למחוק ערך לפי המפתח שלו וקשה לספור את מספר הערכים. ג'אווהסקריפט מספקת שני מבני נתונים מתוחכמים יותר לאחסון נתונים, וכך אפשר ליהנות מהיתרונות של אובייקטים אבל גם מהפונקציות של המערכים.

Map

מבנה הנתונים הראשון מאפשר להכניס ולאחזר ערכים בפורמט key value פשוט שבפשוטים. אין ערכים כפולים. בגדול, הוא בדיוק כמו מערך, אלא שבמקום מספרים יש שם מפתח וערך. על מנת ליצור Map צריכים להשתמש בפונקציה הבנאית new Map, שמחזירה אובייקט שיש לו את התכונות ואת המתודות של Map.

למשל, כך יוצרים Map בסיסי עם המפתחות username, city ו-phone:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
```

הפונקציה set מאפשרת להכניס ערכים ל-Map והערכים יכולים להיות, כמובן, כל דבר, לא רק מחרוזות טקסט כמו פה אלא גם מספרים, מערכים ואפילו Map אחר. בזמן היצירה אפשר ליצור את ה-Map עם מפתחות. למשל:

```
let myMap = new Map([['userName', 'Ran'], ['city', 'Petah Tiqwa']]);
```

איך מאחזרים את השמות? באמצעות מתודת get שיש בה ארגומנט של המפתח המתאים. ממש כך:

```
const phone = myMap.get('phone');
console.log(phone); // 6382020
```

אם רוצים לדעת כמה איברים יש במערך, משתמשים בתכונת size. שימו לב שבשונה ממערכים, כאן משתמשים ב-size ולא ב-length. הינה דוגמה:

```
const size = myMap.size;
console.log(size); // 3
```

מחיקה נעשית באמצעות המתודה delete, שמקבלת כארגומנט את המפתח של הערך שרוצים למחוק. למשל:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
console.log(myMap.size); // 3
myMap.delete('city');
console.log(myMap.size); // 2
```

שימו לב שבניגוד למערך, שם אם מוחקים איבר האורך שלו נותר כשהיה, כאן האורך משתנה בהתאם לערכים.

על מנת למחוק את כל המפה, צריכים להשתמש במתודה clear שלא מקבלת שום ארגומנט:

```
console.log(myMap.size); // 3
myMap.clear();
console.log(myMap.size); // 0
```

אפשר להשתמש בלולאת forEach ממש פשוטה על מנת לעבור על כל ה-Map, ממש כמו מערך, אלא שכאן המפתחות הם לא מספרים:

```
let myMap = new Map();
myMap.set('userName', 'Ran');
myMap.set('city', 'Petah Tiqwa');
myMap.set('phone', '6382020');
myMap.forEach((value, key) => {
    console.log(key);
    console.log(value);
})
```

גם לולאת for of עובדת יפה עם Map.

בגדול, מבנה הנתונים Map אמור להכיל כל מבנה נתונים שהמפתחות שלו הם לא מספרים, ויש המון כאלו, מאובייקט של משתמש, דרך אובייקט של פעולה שהמשתמש עושה ועד כל דבר בעצם. היתרון האדיר של Map הוא שאפשר להיות בטוחים שיש רק מפתח אחד מכל סוג. אין חזרות ולא צריך לנהל את הדופליקציות. נוסף על כך מרוויחים יופי של איטרציות, ממש כמו במערכים.

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, שבהם יש מפתחות שיכולים להיות מחרוזת טקסט רגילה, ב-Set יש רק ערכים - כלומר אין ערכים כפולים. אם מנסים להכניס ערך ל-Set והוא כבר קיים, הוא פשוט "יידרס". Set נוח מאוד לאחסון נתונים כמו תגיות, רשימת מצרכים וכל נתון שמה שחשוב 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. שימו לב שהכנסתי את ה-Set לתוך קבוע. מהנקודה הזו אי-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 בדיוק אותן האיטרציות שיש למערכים ול-Map, כלומר אפשר בנחת להשתמש ב-forEach. בפונקציית הקולבק של ה-forEach יש כמובן רק את הערך, כי אין מפתחות ב-Set:

```
tags.forEach((value) => {
    console.log(value); // tag1, tag2, tag3
});
```

נחמד, נכון? אפשר לבדוק אם יש ערך מסוים ב-Set בעזרת מתודת has, שמחזירה true או false:

```
tags.has('tag1'); // true
tags.has('Not here'); // false
```

כמו ב-Map, גם כאן מוחקים ערכים במתודת delete ובודקים את גודל ה-Set בעזרת תכונת ה-size. מובן שאין מפתח אלא משתמשים בערך בלבד:

```
console.log(tags.size); // 3
tags.delete('tag1');
console.log(tags.size); // 2
```

כל ה-Set מתנקה אוטומטית בעזרת מתודת clear. כן, גם כאן יש דמיון ל-Map:

```
tags.clear();
console.log(tags.size); // 0
```

כאמור, Set שימושי מאוד לאחסנת ערכים שאין צורך במפתחות שלהם, כגון רכיבי מתכון, תגיות, שמות וכו'. Set יכול להכיל כל נתון, ולא רק מחרוזות טקסט.

תרגיל:

צרו Map שיכיל עיר, רחוב, מספר בית ומיקוד.

```
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:

יוצרים קבוע בשם address מסוג Map. שימו 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 בדיוק מה שעושים – ברגע שהקבוע address הוגדר כ-Map, אפשר להשתמש במתודה set על מנת להכניס לתוכו מפתחות וערכים. הדפסה של האובייקט בקונסולה תראה את ה-Map במלואו.

תרגיל:

צרו Set של רשימת משתמשים שמכילה את המשתמשים moshe, yaakov ו-itzhak. הציגו גם את הגודל שלו.

פתרון:

```
const names = new Set();
names.add('moshe');
names.add('yaakov');
names.add('itzhak');
console.log(names.size); // 3
```

הסבר:

יצירת ה-Set נעשית בעזרת פונקציה בנאית מסוג Set. מכניסים את ה-Set החדש לתוך קבוע. זה לא אומר שאי-אפשר להכניס דברים ל-Set או למחוק דברים מתוכו, אבל אי-אפשר לשנות את names לסוג אחר. קל ונחמד.

משתמשים במתודת add, שקיימת בכל Set להכנסת ערכים בלבד. אתם זוכרים שב-Set יש אך ורק ערכים ולא מפתחות? זו הייחודיות של Set לעומת Map. הצגת מספר הערכים נעשית באמצעות תכונת size. זה הכול.

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. לטעון את מערך א'.
2. לטעון את מערך ב'.

3. למיין את מערך א'.
4. למיין את מערך ב'.

מה קורה אם שלב 1 ושלב 2 (טעינת המערכים) הם ארוכים? בכל הדוגמאות, מנוע הג'אווהסקריפט 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. הרי אין היגיון לחכות לטעינת מערך ב' אם מערך א' כבר נמצא בזיכרון ואפשר למיין אותו בזמן שמערך ב' נטען. תוכנה אסינכרונית נראית כך:

1. טוענים את מערך א'. כשהוא נטען (אם אין פעולה אחרת שמתבצעת) – ממיינים אותו.
2. טוענים את מערך ב'. כשהוא נטען (אם אין פעולה אחרת שמתבצעת) – ממיינים אותו.

קל לראות שהתהליך יעיל בהרבה. טוענים את מערך א' ואת מערך ב'. ברגע שהראשון נטען, ממיינים אותו. זה לוקח הרבה פחות זמן. לא מבצעים כלום במקביל – אבל בזמן שממתינים למשהו שייטען, יכולים לעשות דברים אחרים. כדאי לשים לב שזה שונה מלעשות דברים במקביל.

תכנות אסינכרוני הוא אחד מהפיצ'רים החזקים של הפלטפורמות השונות שמריצות ג'אווהסקריפט (מדפדפן ועד שרת), וג'אווהסקריפט תומכת באסינכרוניות באופן מושלם, בטח ובטח בשימושים מודרניים. כשכותבים סקריפט, יש המון פעולות שמריצים והתוצאה שלהן מגיעה מאוחר יותר. הבה נדגים זאת ונשתמש שוב בדוגמה של תוכנה פשוטה של המרת מטבעות. הסקריפט הפשוט הזה מקבל קלט מהמשתמש (למשל סוג המטבע וכמותו), ניגש לשירות חיצוני כלשהו באמצעות האינטרנט, על מנת לקבל את שער ההמרה, מכפיל את שער ההמרה בכמות המטבע ומציג את התוצאה למשתמש.

סדר פעולה	מה קורה בפעולה
1	קבלת קלט מהמשתמש – שם המטבע וכמותו
2	גישה לשרת חיצוני כדי לקבל את שער ההמרה
3	קבלת הנתון מהשרת החיצוני, חישוב הסכום והצגה למשתמש
4	מוכנים לקלט נוסף מהמשתמש

פעולה 2 דורשת זמן. בעוד פעולה 1 ופעולה 3 הן מיידיות, עבור פעולה 2 צריכים לבצע קריאה לשרת מרוחק ולחכות עד שתגיע תשובה. גם אם השרת המרוחק הוא מאוד-מאוד מהיר והרשת של המשתמש סופר-יעילה, עדיין הפעולה הזו תיקח כמה מילישניות לפחות, ואז יש בעיה. אי-אפשר לעבור לשלב 4. האם עוצרים את פעולת הסקריפט

והמשתמש "ננעל"? או שממשיכים את פעולת הסקריפט לדברים אחרים? למשל, המשתמש אולי רוצה גם לבצע המרה 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 עכשיו. בקוד הבא:

```
console.log('1');
console.log('2');
console.log('3');
console.log('4');
```

2 יודפס אך ורק אחרי 1 ו-3 תמיד יודפס אחרי 2. הקוד רץ לפי סדר הכתיבה/קריאה. זה נקרא סדר סינכרוני. כל פעולה בג'אווהסקריפט היא סינכרונית וקוד תמיד ירוץ מההתחלה עד הסוף.

האפשרות האסינכרונית היא מעט מורכבת יותר. אם שלב 2 חיוני רק לשלב 3, אפשר לומר לסקריפט, "שמע נא, מר בחור, שלח את הקריאה לשרת. ברגע שהשרת יחזור עם תשובה, תציג את הפלט למשתמש. אבל עד שזה יגיע, אל תחכה לי. רוץ הלאה לשלב 4 ולשלבים שאחריו". כלומר הקוד האסינכרוני לא חוסם ולא בולם את המשך הרצת הסקריפט. הוא כן חוסם את מה שצריך.

קוד אסינכרוני מאפיין סקריפטים מודרניים. באפליקציות או באתרים כיום יש המון פעולות כאלו. למשל, רישום לדיסק קשיח, קריאה למשאב חיצוני, הפעלת מצלמת וידיאו או מיקרופון. במקום לנעול את המשתמש, מאפשרים לו לעבור הלאה. בדיוק כמו שבגוגל דוקס, למשל, השמירה נעשית ברקע ואפשר לכתוב מסמך בלי הפרעה או "נעילה". בדיוק כפי שבפייסבוק, לדוגמה, אם רוצים ליזום שיחת וידיאו, עד שהמשתמש השני מקבל את הבקשה אפשר לגלול בפיד או לעשות לייק. כל מה שכתבתי 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 ללמוד ולהבין, ולמדנו עליה בפרק על הפונקציות. בואו נחזור ונסביר מה זה קולבק.

על מנת להמחיש את השיטה ניצור הדמיה של פונקציה שקוראת לשירות. הפונקציה הזו 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 עיכוב של מילישנייה אחת. אפשר להעמיד פנים שהתשובה הזו הגיעה משרת מרוחק. בפונקציה אמיתית, לא יהיה setTimeout אלא תהיה קריאה אחרת, אבל בכל מקרה הקריאה הזו תיקח זמן. אפילו מילישנייה אחת (שזה מעולה) היא עדיין זמן.

אם מנסים לקרוא לפונקציה הזו, מקבלים undefined. למה? כי הפונקציה עדיין לא רצה בכלל, אף על פי שמדובר בעיכוב של מילישנייה אחת בלבד!

כלומר, כשנמצאים בשלב 3, שלב 2 עדיין לא הספיק להתרחש.

יש דרכים סינכרוניות לטפל בכך, אבל הבה נטפל בכך באופן אסינכרוני. אם רוצים ששלב 3 יתרחש רק לאחר ששלב 2 יושלם, הדרך לעשות זאת היא להכניס את שלב 3 לפונקציה ולגרום לפונקציה בשלב 2 לקרוא לה באופן אוטומטי. הדרך הזו נקראת "קולבק".

איך גורמים לפונקציה getService להריץ פונקציה אחרת? מעבירים לה ארגומנט שהוא בעצם פונקציה וקוראים לפונקציה הזו עם מה שמגיע מהשרת המרוחק:

```
function getService(cb) {
    setTimeout(() => {
        cb('Result from remote service');
    }, 1);
}
```

למדנו את זה לעומק בפרק על הפונקציות, אבל ארחיב שוב. ארגומנטים יכולים להיות פונקציות. בג'אווהסקריפט פונקציות הן אובייקטים. כמו שאפשר להעביר מחרוזת טקסט, מספר, מערך וכל דבר אחר, אפשר להעביר פונקציה. ברגע שהיא בתוך משתנה, קוראים לה כמו פונקציה רגילה ומעבירים לה איזה ארגומנט שרוצים. במקרה הזה, הארגומנט הוא סתם מחרוזת טקסט מומצאת. בקריאה לשרת הוא יכול להיות מחרוזת טקסט משמעותית או אובייקט מידע.

הפונקציה מקבלת ארגומנט של פונקציה וקוראת לו אך ורק כשהפעולה מסתיימת, ובמקרה הזה, אחרי מילישנייה אחת:

עכשיו מפעילים את הפונקציה. צריך רק לקרוא ל-getService, כשהארגומנט הוא הפונקציה שרוצים שתעבוד מייד לאחר שהפעולה של getService תושלם. מעבירים כארגומנט פונקציה אנונימית מסוג חץ שתדפיס את מה שמעבירים אליה, זה הכול. שימו לב שה-cb פה הוא הארגומנט שאותו מעבירים ושהוא אמור להיות פונקציה, כמובן:

```
function getService(cb) {
    setTimeout(() => {
        cb('Result from remote service');
    }, 1);
```

```
}
getService(
    (answer) => {
        console.log(answer)
    }
);
```

ככה עובד קוד אסינכרוני עם קולבקים. הבעיה מתחילה כשיוצאים ממחוזות הדוגמה אל מחוזות המציאות. נניח שיש אפליקציה שבה הלקוח מכניס מספר בשקלים ובודק כמה מניות של חברה זרה הוא יכול לקנות. צריך לעשות את הפעולות הבאות:

1. לקבל את הקלט מהלקוח – מחיר בשקלים וסוג מניה.
2. לראות מה שער הדולר באמצעות קריאה לשירות חיצוני ולבדוק כמה דולרים השקלים האלו שווים.
3. לקבל את מחיר המניה באמצעות שירות חיצוני אחר ולראות כמה מניות אפשר לקנות.
4. להציג את המידע ללקוח.

כלומר, אם הלקוח רוצה לרכוש ב-40 שקל מניות של חברת אינטל, קוראים לשירות מסוים שיראה מה שער הדולר. נניח שהוא 4, וללקוח יש 10 דולרים. אחרי כן קוראים לשירות אחר שיראה את מחיר המניה, נניח שהוא 5 דולרים. הפלט שיוצג ללקוח הוא 2 מניות. 40 לחלק ל-4 לחלק ל-5, או **בתרגיל:** 40:4:5.

איך מממשים את זה? הינה השירותים שמדמים קריאה לשער הדולר וקריאה למחיר מניה:

```
function getCurrencyRate(cb) {
    setTimeout(() => {
        cb(4);
    }, 1000);
}
```

```
function getStockPrice(stockSymbol, cb) {
    setTimeout(() => {
        cb(5);
    }, 1000);
}
```

השירות המזויף הראשון getCurrencyRate דומה לזה שתרגלנו – הוא קורא לקולבק שמעבירים לו ומחזיר 4. השירות המזויף השני הוא getStockPrice. מעבירים לו שני ארגומנטים – הארגומנט הראשון הוא סימן המניה והשני הוא קולבק. השירות המזויף 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
        });
    }
);
```

אז מבצעים קריאה ל-getCurrencyRate כשמעבירים פונקציית חץ כארגומנט. פונקציית החץ תקבל את שער הדולר מהשירות המזויף ותקרא מייד לשירות שמחזיר את מחיר המניה. שער הדולר ומחיר המניה יסייעו לחשב כמה מניות אפשר לרכוש. במקרה הזה – שתיים.

אפשר לראות שהמבנה הזה מסובך למדי. קולבק בתוך קולבק.

```
function getCurrencyRate(cb) {
  setTimeout(() => {
    cb(4);
  }, 1000);
}

function getStockPrice(cb) {
  setTimeout(() => {
    cb(5);
  }, 1000);
}

const amount = 40;

getCurrencyRate(
  (rate) => {
    getStockPrice((price) => {
      const finalResult = amount/rate/price;
      console.log(finalResult); //2
    });
  }
);
```

קולבק ראשון

קולבק שני

המצב הזה עוד יכול להסתבך ככל שהסקריפט מורכב יותר, ואז 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 פונקציית getCurrencyRate, שמדמה החזרה של שער הדולר באמצעות שרת חיצוני:

```
function getCurrencyRate(cb) {
    setTimeout(() => {
        cb(4);
    }, 1000);
}
```

קראו לפונקציה הזו והחזירו את התוצאה שלה בקונסולה.

```
getCurrencyRate(
    (answer) => {
        console.log(answer)
    }
);
```

הסבר:

קוראים לפונקציה getCurrencyRate עם ארגומנט אחד. איזה ארגומנט? פונקציה אנונימית מסוג חץ שמקבלת ארגומנט. הארגומנט הזה יודפס. איך זה עובד בפועל? פונקציית getCurrencyRate מצפה לקבל ארגומנט שהוא קולבק. היא מדמה קריאה לשרת ומפעילה את הקולבק עם ארגומנט שהוא התשובה. הפונקציה האנונימית שיצרתם מקבלת את התשובה ומחזירה אותה.

תרגיל:

צרו שירות מזויף המדמה קריאה לדיסק הקשיח וקריאת קובץ. השירות המזויף יקבל שם קובץ וקולבק. הוא תמיד יחזיר טקסט שבו כתוב "this is from filename", ושם הקובץ יהיה השם שמועבר לו. קראו לשירות המזויף והדפיסו את התוצאה בקונסולה.

```
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 לקולבק.

עכשיו רק צריך לקרוא לשירות ולהעביר אליו שם קובץ מזויף ופונקציית חץ אנונימית שתופעל בסיום הפעולה. בפונקציית החץ האנונימית רק מבצעים הדפסה של התוצאה:

```
function readFile(fileName, cb) {
  setTimeout(() => {
    cb(`This is from ${fileName}`);
  }, 1000);
}

readFile(
  'somename.txt',
  (result) => {
    console.log(result); // "This is from somename.txt"
  }
);
```

הקריאה

ארגומנט ראשון

ארגומנט שני

Promises

כפי שראינו, קולבקים הם שיטה יעילה מאוד לעבוד עם קוד אסינכרוני, אבל יש להם חסרונות, ובראשם הסרבול הגדול מאוד של הקוד או "גיהינום הקולבקים", וגם חסרונות אחרים – כמו למשל חוסר היכולת לתפוס שגיאות (את עניין ה-try-catch למדנו באחד הפרקים הקודמים). כדי לנהל את השגיאות קיימים פרומיסים (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. בקוד שקורא לפונקציה אפשר להחליט מה קורה אם ההחלטה מתקיימת ומה קורה אם לא.

איך יוצרים promises? הבה נשתמש בשירות המזויף מהפרק הקודם. השירות מחכה שנייה ואז מחזיר 4. הוא מדמה שירות שמחשב שערי מטבעות וקורא לשרת חיצוני. הקוד הוא פשוט מאוד:

```
setTimeout(() => {
    // Return 4
}, 1000);
```

מובן שזו יכולה להיות קריאה לשרת אחר, קריאה למאגר נתונים או לכל מקום שהוא. בסופו של דבר, מדובר בפקודה שמקבלת קולבק שפועל אחרי 1,000 מילישניות. למדנו על כך בפרק על setTimeout וגם 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. יצירת ההבטחה נראית כך:

```
let myPromise = new Promise((resolve, reject) => {
});
```

כאן ממש אפשר לראות את יצירת ה-Promise באמצעות פונקציה בנאית שמקבלת כארגומנט פונקציית חץ אנונימית, שלה יש שני ארגומנטים שהם בעצם קולבקים.

כל מה שנשאר לעשות הוא להכניס את השירות המזויף ולקרוא ל-resolve שבתוכו. כאן רוצים לממש את ההבטחה עם 4:

```
function readFile() {
    let myPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(4);
        }, 1000);
    });
    return myPromise;
}
```

אפשר לראות שהפונקציה הבנאית מקבלת פונקציית חץ אנונימית שיש לה שני ארגומנטים – reject ו-resolve. מדובר בשני קולבקים שאפשר להפעיל. הראשון הוא בעצם ההפעלה במקרה של הצלחה, וזה בדיוק מה שעושים בשירות המזויף.

השירות המזויף מחכה שנייה אחת (1,000 מילישניות) ואז קורא ל-resolve ומעביר את הארגומנט 4. הכול ארוז בתוך פונקציית readFile. הפונקציה מחזירה הבטחה. אפשר לקרוא לפונקציה הזו ולקבל את ההבטחה.

מה עושים עם ההבטחה? אפשר בעצם לבוא ולומר, "תשמעי, הבטחה, ברגע שתתמלאי, תריצי את הקוד הזה". איך עושים את זה? לכל הבטחה יש מתודת then שמקבלת כארגומנט ראשון פונקציה שמופעלת כשההבטחה מתממשת:

```
readFile().then((result) => { console.log(result) });
```

הקוד הזה רץ בכל פעם שההבטחה הנמצאת ב-readFile מתממשת.

בואו נמחיש את זה באמצעות דוגמה נוספת, הפעם אפילו בלי setTimeout: יוצרים פונקציה שמחזירה הבטחה. שמה readFile, והיא לא תקבל שום ארגומנט. מה שהיא עושה הוא ליצור הבטחה. בתוך ההבטחה, במקום לקרוא לשירות זה או אחר, פשוט עושים resolve ומקיימים את ההבטחה.

כשקוראים לפונקציה, משתמשים ב-then ומעבירים כארגומנט פונקציה שמאזינה להבטחה. ברגע שהיא תתקיים, היא תפעל. כך נראה הקוד:

```
function readFile() {
    let myPromise = new Promise(
        (resolve, reject) => {
            resolve('Promise resolved!')
        }
    );
    return myPromise;
}
readFile().then(
    (result) => {
        console.log(result);
    }
);
```

השרטוט הבא ימחיש טוב יותר את העניין. בפונקציית readFile יוצרים את ההבטחה. ההבטחה מקבלת ארגומנט של פונקציית חץ שבתוכו מקיימים אותה ומחזירים תגובה. בחלק השני קוראים לפונקציה:

```
function readFile() {
  let myPromise = new Promise(
    (resolve,reject) => {
      resolve('Promise resolved!')  כאן אני מקיים את ההבטחה ומחזיר תגובה
    }
  );                                                          ההבטחה שיצרתי
  return myPromise;  החזרת ההבטחה למי שקורא לפונקציה
}
                          קריאה לפונקציה המחזירה הבטחה
                          הפונקציה הראשונה המועברת כארגומנט תיקרא
readFile().then(                                  כאשר ההבטחה תתקיים
  (result) => {
    console.log(result);  ירוץ כאשר ההבטחה תתקיים
  }
);
```

מה שחשוב להבין הוא שיש לשימוש בהבטחות שני חלקים – החלק הראשון הוא יצירת הפונקציה שבה יש את ההבטחה וכתיבת ההבטחה, כולל החלק שבו מבצעים resolve או מקיימים את ההבטחה. החלק השני הוא ההאזנה, באמצעות then, לפונקציה שמחזירה הבטחה.

כמו שאפשר לקיים הבטחה, אפשר גם להפר אותה. למשל אם השירות מחזיר שגיאה כי השרת לא נמצא, כי מסד הנתונים נפל או מכל סיבה שהיא. הפרת ההבטחה נעשית באמצעות המתודה reject. זה נראה כך:

```
function waitTwoSeconds() {
    let secondsPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('ERROR! ERROR!');
        }, 2000);
    });
    return secondsPromise;
}
waitTwoSeconds().then(
    (result) => { console.log(result) }, // Will never happen
    (error) => { console.log("Bad error:" + error) } //"Bad error:ERROR!
ERROR!"
);
```

כאן יש את ה-setTimeout שראינו קודם, אבל במקום לקיים את ההבטחה אחרי שתי שניות, מפרים אותה בגסות. שימו לב שהכול נראה אותו דבר: בניית הפונקציה שעוטפת את ההבטחה והשימוש ב-setTimeout. השוני העיקרי כאן הוא השימוש ב-reject. כך מפרים את ההבטחה. גם פה אפשר לשלוח ארגומנט שייתפס בפונקציה שקולטת את ה-reject.

הפונקציה שתופסת את הפרת ההבטחה נכנסת כארגומנט השני של פונקציית then.

הערה: בקוד מורכב מומלץ לעשות reject עם אובייקט שגיאה מסודר, אחרת לא מקבלים לוג שגיאות מסודר וזה עלול לבלבל.

```
function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      reject('ERROR! ERROR!');
    }, 2000);
  });
  return secondsPromise;
}

waitTwoSeconds().then(
  (result) => {console.log(result)}, // Will never happen
  (error) => {console.log("Bad error:" + error)} //"Bad error:ERROR! ERROR!"
);
```

הפונקציה הראשונה תרוץ במקרה וההבטחה תקוים

הפונקציה השני תרוץ במקרה וההבטחה תופר

הפונקציה הראשונה מתממשת רק כשההבטחה מקוימת. במקרה הזה – זה לעולם לא יקרה.

הפונקציה השנייה היא החידוש פה: מעבירים אותה כארגומנט השני של ה-then והיא תפעל אך ורק כאשר ההבטחה תופר. במקרה הזה היא מדפיסה את מה שהיא מקבלת בקונסולה בתוספת המילים Bad error. כיוון שבהפרת ההבטחה מחזירים את המילים ERROR! ERROR!, מקבלים בקונסולה את המשפט:

"Bad error: ERROR! ERROR!"

בשימוש בהבטחות צריך תמיד לזכור לסגור את כל המקרים, כדי שלא יקרה שהקוד פשוט יפסיק לרוץ מפני ששכחתם להשתמש ב-resolve/reject באחד ממשפטי התנאי.

שרשור הבטחות

אחד הפיצ'רים החזקים ביותר בהבטחות הוא היכולת לשרשר אותן – כלומר ה-then יכול להחזיר promise שייתפס על ידי then שמשורשר אליו. ראשית נשאלת השאלה – למה צריך את זה? התשובה פשוטה: פעמים רבות פונים לשרת, ואת התגובה שמקבלים ממנו שולחים לשרת אחר או ששומרים במסד הנתונים או כל דבר אחר. שרשור הבטחות או פונקציות אסינכרוניות הוא נפוץ מאוד.

בדוגמה הבאה אראה איך עושים את זה. יש שתי פונקציות שמממשות הבטחה. אחת היא getNumber שתמיד מקיימת את ההבטחה ומחזירה 100, והאחרת היא writeNumber שבודקת את המספר. אם המספר הוא 100, היא מקיימת את ההבטחה ומחזירה OK. כאן אפשר לראות את התלות – הפונקציה writeNumber תלויה בפונקציית getNumber שתחזיר לה את המספר. הן ממומשות כמו כל הבטחה שכבר ראינו, אבל מה שמיוחד בשרשור ההבטחות הוא מי שקורא לפונקציות האלו:

```
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 שמממשים את הקריאה ל-getNumber כרגיל ומשתמשים ב-then כדי לתפוס את הקריאה הראשונה. מה שמיוחד פה הוא שלוקחים את התוצאה של קיום ההבטחה הראשונה (שתהיה מן הסתם 100) ובמקום להחזיר אותה לפונקציה, מבצעים קריאה של הפונקציה השנייה עם התוצאה (שכאמור היא 100) ומחזירים אותה! זו הבטחה לכל דבר ועניין ואפשר לתפוס אותה עם then, שם מדפיסים את התוצאה, שמן הסתם היא OK.

אמחיש 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 ההבטחה יותר מפעם אחת.

ואיך תופסים בשיטה הזו הבטחה שנכשלת? בעזרת מתודת catch שמקבלת לתוכה את מה ששולחים ב-reject, בדיוק כמו במתודה השנייה ב-then הרגיל שלמדנו עליו בסעיף הקודם. הינה דוגמה ל-catch:

```
function myPromise() {
    let promise = new Promise(function (resolve, reject) {
        reject('promise rejected!');
    });
    return promise;
}
myPromise().then((data) => {
    return data + ' 1 ';
})
    .then((data) => {
        return data + ' 2 ';
    })
    .then((data) => {
```

```
        console.log(data);
    })
    .catch((error) => {
        console.log(error); //promise rejected!
    });
```

הדוגמה הזו זהה לדוגמה הקודמת למעט שני דברים: נוסף catch שתופס ומדפיס כל error, ואת ה-promise מחזירים כ-reject. הוספת ה-catch מאפשרת לתפוס שגיאות בכל רגע נתון כשמשרשרים then.

קיבוץ הבטחות

אם יש כמה הבטחות שאינן תלויות זו בזו ורוצים לשלוח 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 לשרשר לה then שיטפל במצב שבו כולן עוברות או שאחת מהן לפחות נכשלת, אף על פי שסדר השליחה וההפעלה אינו מובטח.

כלומר אם יש שלוש הבטחות, אפשר להאזין לקיום של שלושתן בבת אחת. אם כולן עוברות, הפונקציה הראשונה ב-then מופעלת. אם אחת נכשלת, הפונקציה השנייה מופעלת, ממש כמו בהאזנה להבטחה בודדת, ואין זה משנה אם שאר ההבטחות קוימו 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 ההבטחות שלהן. שימו לב שבמערך קוראים לפונקציות כדי לקבל את ההבטחות שלהן. המערך הוא של הבטחות ולא של פונקציות. את המערך מכניסים ל-Promise.all ואפשר להשתמש עכשיו ב-then. כיוון שיש כמה הבטחות וכמה תוצאות לקיום ההבטחות, הכול נכנס למערך של results. זה השוני היחידי.

תרגיל:

צרו שירות, באמצעות setTimeout, שמחזיר לאחר שתי שניות תגובה עם המילים Two seconds passed. הכניסו אותו להבטחה והכניסו את ההבטחה לתוך פונקציה שתחזיר אותה.

בדקו שהפונקציה עובדת באמצעות האזנה עם then לקיום ההבטחה:

```
function waitTwoSeconds() {
    let secondsPromise = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Two seconds passed!');
        }, 2000);
    });
    return secondsPromise;
}
waitTwoSeconds().then((result) => { console.log(result) });
```

הסבר:

החלק הראשון הוא יצירת ה-setTimeout, שמקבל שני ארגומנטים. הראשון הוא מה שיש לעשות אחרי פרק הזמן שהוגדר לו והשני הוא פרק הזמן במילישניות. 2,000 מילישניות הן שתי שניות.

החלק השני הוא יצירת פונקציה שמחזירה הבטחה. יוצרים פונקציה ששמה waitTwoSeconds. בתוכה יש הבטחה ששמה secondsPromise ואותה מחזירים.

החלק האחרון והקשה ביותר הוא ליצוק תוכן בתוך ההבטחה. ההבטחה נוצרת באמצעות פונקציה בנאית של Promise. היא מקבלת ארגומנט אחד שהוא פונקציית קולבק עם שני ארגומנטים, resolve ו-reject. בתוך הקולבק מכניסים את ה-setTimeout וקובעים שההבטחה תקוים ברגע ששתי השניות יעברו.

הבדיקה פשוטה יותר. קוראים לפונקציה ואז מאזינים עם then. היא מקבלת ארגומנט אחד, פונקציה אחת שפועלת ברגע שההבטחה מתקיימת. כל מה שצריך לעשות הוא להאזין למה שמחזירים ב-result ולהדפיס אותו בקונסולה:

```
function waitTwoSeconds() {
  let secondsPromise = new Promise((resolve,reject) => {
    setTimeout(() => {
      resolve('Two seconds passed!');
    }, 2000);
  });
  return secondsPromise;
}

waitTwoSeconds().then((result) => {console.log(result)});
```

מה שאני מחזיר בקיום ההבטחה מועבר לכאן

כאשר ההבטחה מתקיימת, הפונקציה הזו מופעלת

תרגיל:

צרו פונקציה checkIfNumberOK שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזור הבטחה מקוימת עם מחרוזת הטקסט Number OK. אם המספר קטן מ-10 או שווה לו, תחזור הבטחה מופרת עם מחרוזת הטקסט Number bad.

תשובה:

```
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:

הפונקציה checkIfNumberOK היא פונקציה רגילה המקבלת מספר כארגומנט. בתוך הפונקציה יוצרים הבטחה כאובייקט Promise עם הפונקציה הבנאית. הפונקציה מקבלת ארגומנט שהוא פונקציה אנונימית מסוג חץ, ובה יש

שני ארגומנטים, resolve ו-reject. משתמשים בשניהם. מפעילים משפט תנאי פשוט ובודקים: אם המספר גדול מ-10, ההבטחה תקוים באמצעות resolve, ואם המספר קטן מ-10 היא תופר באמצעות reject - כמובן עם הטקסטים הרלוונטיים.

הבדיקות יהיו פשוטות למדי – הרצה של הפונקציה עם המספר 11 ותפיסת ההצלחה או השגיאה באמצעות then. כיוון ש-checkIfNumberOK מחזירה הבטחה, אפשר להשתמש ב-then שמקבל שני ארגומנטים, ארגומנט של הפרה וארגומנט של הצלחה. מן הסתם, כשמעבירים 11, ההבטחה מקוימת והפונקציה הראשונה שמעבירים ב-then תופעל. כשמעבירים מספר קטן יותר מ-10, ההבטחה מופרת והפונקציה השנייה שמעבירים ב-then תופעל.

פונקציית async

השיטה השלישית לתכנות אסינכרוני לא שונה כל כך מהשיטה של הבטחה. למען האמת מדובר בהרחבה שלה. פונקציית async מאפשרת לטפל בהבטחות ללא ה-then או ליתר דיוק מאפשרת לכתוב קוד אסינכרוני בצורה סינכרונית. אם יש שירות שמחזיר promise, אפשר לבחור אם לעבוד עם then כמו בפרק הקודם או בפונקציה אסינכרונית עם המילה השמורה async.

לשם הדגמה אשתמש בדוגמה של הקריאה המזויפת שהשתמשנו בה קודם. הקריאה הזו 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 מזויפת של 4 אחרי שנייה אחת. שימו לב שהיא מחזירה promise. אם היינו רוצים להתחבר אליה, היינו צריכים לקרוא לה ואז להשתמש ב-then כדי לתפוס את ההצלחה. אבל אפשר לעשות את זה עם async באופן פשוט. ראשית מגדירים פונקציה אסינכרונית, כלומר הפונקציה הזו מריצה קוד שיכול לרוץ בזמן אחר. ההגדרה נעשית באמצעות המילה השמורה async. בתוך הפונקציה הזו אפשר לקרוא לפונקציות המחזירות הבטחה כפי שרוצים. פונקציות המחזירות הבטחה יקבלו await ליד הקריאה שלהן, והסקריפט בתוך ה-async יעצור וימתין להן. קוד מחוץ לפונקציה ירוץ כרגיל. כך מרוויחים אסינכרוניות אבל עם קוד ברור יותר.
למשל:

```
async function main() {
    let result = await readFile();
```

```
    console.log('result' + result);
}
main();
```

בדוגמה יוצרים פונקציה שמוגדרת כאסינכרונית. שימו לב למילה השמורה async לפניfunction main. בתוך הפונקציה הזו, שיש בה async, אפשר להשתמש במילה השמורה await שקוראת לפונקציה המחזירה promise. שימו לב ש-readFile לא שונה מפונקציה אחרת שמחזירה הבטחה, בדיוק 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 הזו לא חוזרת
הפונקציה לא ממשיכה

לא ירוץ עד שה await
לא יושלם

כפי שב-then יש שני ארגומנטים – אחד שמתקיים כאשר ההבטחה מקוימת ואחד שמתקיים כאשר ההבטחה נכשלת – גם ב-async אפשר לתפוס את הכישלון. איך? בעזרת try-catch רגיל, שעליו למדנו בפרק על טיפול בשגיאות. בדוגמה הבאה יוצרים שירות שתמיד 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 occured: ' + error); // An error occured: error
:(
    }
}
main();
```

שירות readFile הוא שירות שתמיד נכשל. הוא מחכה שנייה ועושה reject. היישום שלו הוא בדיוק כמו כל שירות אחר שמחזיר הבטחה ומקיים אותה או לא מקיים אותה. מה שמעניין הוא שה-async שנקרא main זהה לדוגמה הקודמת, למעט ה-try-catch. ה-try יקרה כאשר ההבטחה תקוים (ובדוגמה הזו לעולם לא) וה-catch יתקיים אם ההבטחה לא תקוים (בדוגמה הזו תמיד כי תמיד עושים reject). מה שקורה הוא שמודפסת שגיאה בקונסולה. אפשר כמובן לקבוע כל התנהגות אחרת.

מובן שאין בעיה לקבוע כמה await בתוך פונקציית async אחת. מה שצריך לזכור, וגם לתרגל, הוא שבסופו של דבר מדובר בציפוי 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:

בדיוק כמו בפרק הקודם, צרו פונקציה checkIfNumberOK שמקבלת מספר ומחזירה הבטחה. אם המספר גדול מ-10, תחזור הבטחה מקוימת עם מחרוזת הטקסט Number is OK. אם המספר קטן מ-10, תחזור הבטחה מופרת עם מחרוזת הטקסט Bad Number.

התרגיל האמיתי הוא לממש את הקריאות עם async. צרו פונקציית async שמשתמשת בשירות, וקראו לה פעם אחת באופן כזה שהיא תדפיס בקונסולה את מה שהשירות מחזיר אם הוא מצליח ופעם אחת באופן כזה שהיא תדפיס בקונסולה את מה שהשירות מחזיר אם הוא נכשל:

```
function checkIfNumberOK(number) {
    let myPromise = new Promise((resolve, reject) => {
        if (number > 10) {
            resolve('Number is OK');
        } else {
            reject('Bad Number');
        }
    });
    return myPromise;
}
async function checkMyNumber(number) {
    try {
        let result = await checkIfNumberOK(number);
        console.log(`success! ${result}`);
    } catch (error) {
        console.log(`error! ${error}`);
```

```
    }
}
checkMyNumber(11); // success! Number is OK
checkMyNumber(5); // error! Bad Number
```

הסבר:

על השירות שנוצר כאן אין מה להסביר יותר מדי כי למדנו אותו בפרק הקודם. זה שירות שמקבל מספר ומחזיר הבטחה. אם המספר גדול מ-10, הוא מקיים את ההבטחה. אם לא, הוא מפר אותה.

מה שמעניין הוא איך שקוראים לשירות: יוצרים פונקציית async בשם checkMyNumber שמקבלת מספר. בתוכה יש try-catch. בתוך ה-try קוראים לשירות ולא שוכחים להשתמש במילה השמורה await כדי להראות שמדובר בשירות אסינכרוני שיש לחכות לו. אם ההבטחה מקוימת, מה שיש בתוך ה-try רץ. אם לא, מה שיש בתוך ה-catch רץ. עכשיו 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:

```
async function callMyPromises() {
    let results = [];
    results[0] = await myPromiseA();
    results[1] = await myPromiseB();
    results[2] = await myPromiseC();
    console.log(results);
```

```
}
callMyPromises(); //  ["Promise A success!", "Promise B success!", "Promise C success!"]
```

הסבר:

ראשית יוצרים פונקציה בשם callMyPromises. כדי לסמן שהיא אסינכרונית משתמשים במילה השמורה async. ברגע שמשתמשים במילה הזו, אפשר להשתמש ב-await לפני כל פונקציה שמחזירה הבטחה. שימוש כזה יבטיח שהשורה הבאה בפונקציה תחכה. כך למשל:

```
results[1] = await myPromiseB();
```

תרוץ רק אחרי שההבטחה של myPromiseA תתמלא.

```
results[0] = await myPromiseA();
```

נותר רק לקרוא לשירותים השונים לפי הסדר ולהדפיס את התוצאה. שימו לב שזרימת הקוד מחוץ ל-callMyPromises מתנהלת כרגיל, אבל זה לא מאוד מעניין.

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 שרתי אינטרנט באמצעות ג'אווהסקריפט ופרוטוקול HTTP. זה נשמע מעט מצחיק, כי ג'אווהסקריפט היא שפה שחיה ברשת. אבל כשחושבים על כך לעומק, ג'אווהסקריפט לא חיה ברשת. ג'אווהסקריפט היא שפה שחיה בדפדפן ונטענת מייד כשהדף נטען. אם התקנתם את סביבת הבדיקות שהסברתי עליה באחד הפרקים הראשונים, אתם יודעים שבכל פעם שמבצעים טעינת דף, קוד הג'אווהסקריפט רץ וזהו. הקוד רץ עם המידע שמכניסים לו. הוא יכול לקחת את המידע מה-DOM או ממשתנים שמגדירים לו, אבל מקור המידע אחד – האתר שהדפדפן טוען.

עם AJAX אפשר לגשת לכל אתר שהוא ולקחת ממנו מידע. לצורך העניין אפשר אפילו להיכנס לאתר חדשות, לשאוב את ה-HTML שלו ואז, באמצעות ביטוי רגולרי (שעליו למדנו בפרק המוקדש לביטויים כאלה), לקחת את הכותרת ולהציגה למשתמש. מה שדפדפן או, נכון יותר, המשתמש בדפדפן יכול לעשות, גם ג'אווהסקריפט יכולה לעשות. אבל בדרך כלל לא משמשים ב-AJAX כדי לפנות לאתרים המיועדים לבני אדם, אלא כדי לפנות לכתובות אינטרנט שמחזירות אובייקטי JSON, שקל לעבוד איתם בג'אווהסקריפט. אם תפעילו את הדפדפן ותנסו להיכנס לכתובת הזו, גם אתם תראו את אובייקט ה-JSON. כאמור, AJAX הוא בדיוק כמו דפדפן שמשגר נתונים, אבל עושים את זה באמצעות קוד.

הבה נדגים. יש ברשת שירותים רבים המספקים מידע, למשל אתר המספק נתוני מזג אוויר או שערים של מטבעות חוץ. אלו אתרים המספקים API, שזה ראשי התיבות של Application Programming Interface, ובעצם המשמעות היא שמי שימצא בהם תועלת הם בעיקר סקריפטים של AJAX. אפשר למצוא רשימות של API כאלו ברשת בחיפוש public API data sites. אחד האתרים המרכזים רשימה של שירותים כאלו נמצא בכתובת הבאה:
https://github.com/toddmotto/public-apis

אפשר למצוא שם עשרות אתרים המספקים API. כאן נתרגל בשירות שנקרא "בדיחות צ'אק נוריס". השירות אינו דורש הזדהות, רישום או כאב ראש אחר ובעצם כל מה שצריך לעשות הוא לשלוח בקשה אל:

https://api.chucknorris.io/jokes/random

כדי לקבל בדיחה רנדומלית של צ'אק נוריס.
קל לבדוק אותו! פשוט היכנסו לקישור באמצעות דפדפן רגיל. תוכלו לראות שאתם מקבלים ממש JSON עם מידע.

```
{
  "category": null,
  "icon_url": "https://assets.chucknorris.host/img/avatar/chuck-norris.png",
  "id": "uDjkIgHpQhe7kv8xoyJm6g",
  "url": "http://api.chucknorris.io/jokes/uDjkIgHpQhe7kv8xoyJm6g",
  "value": "Chuck Norris is currently suing NBC, claiming Law and Order are
trademarked names for his left and right legs."
}
```

מובן שעבור משתמש רגיל זה קצת בעייתי, אבל לסקריפט שמשגר בקשת AJAX זה מעולה. יוצרים בקשת AJAX אל האתר הזה ומציגים את המידע.

נשאלת השאלה "איך?" בג'אווהסקריפט המודרנית יותר מקובל להשתמש ב-fetch, פונקציה גלובלית שקל להשתמש בה וקיימת בסביבת הדפדפן. הפונקציה הזו מחזירה promise שהוא התוצאה. מהתוצאה בוררים את סוג הנתון שרוצים ומחזירים אותו, ואיתו עושים מה שרוצים, ובמקרה של שירות הבדיחות – מדפיסים את הבדיחה.

השימוש ב-fetch:

```
fetch('https://api.chucknorris.io/jokes/random')
```

פשוט מאוד, נכון? הפונקציה הזו מחזירה promise עם התגובה מהשרת. ב-promise למדנו להשתמש בפרקים הקודמים. עכשיו צריך רק להחליט מה עושים אם ה-promise מתקיים. במקרה זה אנחנו רוצים את ה-JSON, סוג המידע שה-API עובד איתו. אם למשל ה-API היה מחזיר תמונה, היה נדרש BLOB שהוא סוג מידע אחר שלא נדון בו כאן:

```
fetch('https://api.chucknorris.io/jokes/random')
  .then((response) => {
```

```
    return response.json();
  })
```

פשוט, נכון? עכשיו אפשר לשרשר עוד then, שבו מחליטים מה לעשות עם ה-JSON. במקרה הזה, מדפיסים את ערך הבדיחה:

```
fetch('https://api.chucknorris.io/jokes/random')
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    document.write(jsonObject.value);
  });
```

וזה הכול! אם אתם שולטים בתכנות אסינכרוני בכלל ובתכנות מבוסס promises, לא צריכה להיות לכם בעיה בהבנת הסינטקס הזה: fetch מחזירה promise עם סוג המידע, ויש לבחור את הסוג ולהחזיר אותו, ואז לעשות בו מה שרוצים.

מתודות של HTTP וארגומנטים נוספים

השימוש שהראיתי עד כה ל-fetch הוא שימוש בסיסי בפורמט HTTP GET. פרוטוקול הרשת לא נלמד בספר זה, אך אפשר לציין בקצרה שבפרוטוקול אינטרנט יש כמה סוגי בקשות. הבקשה הנפוצה היא בקשת GET, לקבל נתונים. זו הבקשה שמפעילים כאשר נכנסים לאתרי אינטרנט למשל. מתודות אחרות הן למשל POST – שבה משתמשים כדי לשלוח נתונים (למשל כאשר ממלאים טופס באינטרנט, הטופס נשלח בבקשת POST), PUT (לעדכון נתונים) או DELETE (למחיקת נתונים).

אם רוצים לשלוח בקשת POST, אין קל מזה. צריך להעביר ארגומנט נוסף ל-fetch, שבו מגדירים את הנתונים ששולחים כשהמתודה מבצעת את הקריאה לשרת.

```
fetch('https://jsonplaceholder.typicode.com/posts/',
  {
    method: 'POST',
    body: { title: 'MyTitle' }
  })
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    console.log(jsonObject); // { id: 101 }
  });
```

בדוגמה הזו שולחים בקשת POST עם אובייקט אל הכתובת https://jsonplaceholder.typicode.com/posts.

קל לראות שזה דומה מאוד לבקשת GET, אלא שכאן נדרשים להעביר ארגומנט שני שמפרט את סוג הבקשה וכמובן

את המידע. באותו אובייקט אפשר גם להגדיר headers למשל. על מנת להבין את סוגי הבקשות יש צורך בידע על HTTP שאינו נלמד בספר זה, אבל הפרקטיקה היא די פשוטה.

כאמור, fetch הוא פשוט מאוד להבנה אם מכירים היטב תכנות אסינכרוני ו-promise.

תרגיל:

נתון ה-API שכתובתו היא http://jsonplaceholder.typicode.com/posts/1 – הדפיסו את הכותרת המתקבלת מהגישה לכתובת הזו.

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    document.write(jsonObject.title);
  });
```

הסבר:

אם נכנסים אל ה-API הזה, רואים שמתקבל אובייקט JSON מהסוג הזה:

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio
reprehenderit",
  "body": "quia et suscipit suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem
eveniet architecto"
}
```

כעת רוצים את ה-title.

ראשית יוצרים fetch אל הכתובת. זה קל ופשוט. הפונקציה fetch מחזירה promise שאותו תופסים עם then. ה-promise מוחזר עם ארגומנט התגובה וצריך תמיד להחזיר את מה שרוצים מהתגובה הזו, לענייננו – JSON. מחזירים את:

response.json

כיוון שאפשר לשרשר promise, צריך להשתמש בעוד then על מנת להדפיס בו את מה שרוצים, לענייננו – הכותרת title, שיש להדפיסה.

תרגיל:

שלחו בקשה מסוג DELETE אל http://jsonplaceholder.typicode.com/posts/1 והדפיסו " Deletion successful" במקרה של הצלחה.

```
fetch('https://jsonplaceholder.typicode.com/posts/1',
  {
    method: 'DELETE'
  })
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    document.write('Delete successful');
  });
```

הסבר:

על מנת לשגר בקשה מסוג DELETE צריך להעביר ארגומנט שני אל fetch. הארגומנט הראשון הוא הכתובת. הארגומנט השני הוא אובייקט שיכול להכיל כמה תכונות אבל במקרה הזה מכיל רק תכונה אחת: method, שהערך שלה הוא DELETE. מפה ממשיכים בדיוק כמו ב-fetch רגיל. מקבלים את התגובה ומעבירים את ה-JSON המתקבל ממנה הלאה. אם מקבלים תגובה מסוג כלשהו, סימן שהמחיקה הצליחה.

ES6 Classes

מחלקות, קלאסים, או classes באנגלית, הן דרך מצוינת לארגן קוד בג'אווהסקריפט. יש לא מעט שפות תכנות שמשתמשות 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, אסביר עליהן כאן, כולל איך הן נראות כאובייקט רגיל.

מחלקה היא אובייקט שיש לו מפתחות. חלק מהמפתחות הם סוגי מידע פרימיטיביים וחלקם פונקציות (ואז קוראים להן מתודות של המחלקה). הבה 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 בדיוק אותו דבר והפונקציונליות היא אותה פונקציונלית, רק שהכול ארוז יפה יותר.

ב-constructor יש את הפונקציה הבנאית עצמה, זו שרצה כאשר מריצים את new. בתוך המחלקה מגדירים גם את המתודות.

זה הכול. בדוגמה אפשר לראות איך יוצרים מהתבנית, שהיא פונקציה בנאית בעצם (שימו לב שגם שם המחלקה הוא עם אות גדולה), את האובייקט הפרטי.

מחלקה היא פשוט ציפוי של סוכר או "סוכר סינטקטי" על הפונקציה הבנאית הרגילה שכבר למדנו עליה. אין צורך להיבהל מהמילים השמורות class או constructor – זה בדיוק מה שלמדנו קודם. המחלקה המפורטת לעיל בעצם נראית כך מאחורי הקלעים:

```
function Book(title) {
  this.title = title;
  this.isCurrentlyReading = false;
  this.start = () => {
    this.isCurrentlyReading = true;
    return this.isCurrentlyReading;
  }
  this.stop = () => {
    this.isCurrentlyReading = false;
  }
}
const theShining = new Book('The Shining'); //Instance of the object.
console.log(theShining.title); //Return "The Shining".
const result = theShining.start();
console.log(result); //Return true.
```

אבל כשם שיש קלאס יש עוד מתודות מוצלחות, שכאמור עוזרות לסדר את הקוד בצורה טובה ונעימה יותר לעין.

בפרק על האובייקטים דיברתי על get ועל set, ובטח תשמחו לדעת שאפשר להגדיר אותם גם פה. אלא מה? צריך להיזהר מאוד מהפניות מעגליות. הינה דוגמה שמתבססת על הדוגמה הקודמת. במקרה הזה יש get שמביא את התכונה הנדרשת (title) ועוטף אותה בתבנית טקסט. ה-set הוא פשוט יותר:

```
class Book {
  constructor(title) {
```

```
    this._title = title;
    this.isCurrentlyReading = false;
  }
  get title() {
    return `Great Book: ${this._title}`;
  }
  set title(newTitle) {
    this._title = newTitle;
  }
}
const theShining = new Book('The Shining'); //Instance of the object.
theShining.title = 'The bible';
console.log(theShining.title); //Great Book: The bible
```

כאמור, את הקונספט של get ו-set כבר למדנו, וכאן מוצגת פשוט דרך פשוטה יותר ונעימה יותר לעין לממש את מה שאתם כבר יודעים. אין פה משהו חדש. זו הסיבה שמיקמתי את הפרק הזה בסוף הספר. מחלקה היא לא וודו אלא פשוט דרך שונה ונעימה לכתיבת פונקציה בנאית, ואת הפונקציה הבנאית שיוצרת אובייקט עם תכונות ומתודות אתם מכירים.

אם יש כמה פונקציות בנאיות שחוזרות על עצמן, אפשר ליצור פונקציה בנאית שמורישה לפונקציות הבנאיות את כל התכונות שלהן. אם למדתם תכנות מונחה עצמים, אז זה דומה – אבל רק כמעט, כיוון שג'אווהסקריפט היא לא שפה מונחית עצמים.

הבה נמחיש זאת באמצעות דוגמה מהחיים. יש שני סוגי לקוחות - רגילים ופרימיום. לשני סוגי הלקוחות יש תכונות דומות: לשניהם יש שמות וכתובות ומתודת set ו-get, אבל ללקוח פרימיום יש גם תכונת "הטבות". איך מממשים כזה דבר? אפשר ליצור שתי מחלקות באופן הזה:

```
class RegularClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class PremiumClient {
  constructor(name, address) {
    this.name = name;
    this.address = address;
```

```
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe", address: "Tel
Aviv"}
const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address: "Petah
Tiqwa"}
console.log(premium.getBenefits()); // 'A lots of premium stuff';
```

מגדירים שתי מחלקות, RegularClient ו-PremiumClient (בתחתית מודגם שימוש בכל אחד מהן). שתי המחלקות זהות, למעט המתודה getBenefits שנמצאת רק במחלקה PremiumClient.

באופן עקרוני אין בעיה עם הקוד הזה והוא עובד. אבל מה כן הבעיה? שכפול קוד. יש כאן קוד שחוזר על עצמו, וזו בעיה. למה? כי אם יבקשו מכם שינויים במבנה הלקוח (למשל להוסיף תכונה נוספת), תצטרכו לשנות את המבנה פעמיים. ואם יש יותר משני סוגי לקוחות, תצטרכו לעשות שינויים במקומות נוספים. שכפול קוד הוא משהו שכדאי במידת האפשר להימנע ממנו.

בדיוק בשביל זה קיים ה-extend, שמאפשר ליצור מחלקת-על ולרשת ממנה את כל התכונות. במקרה הזה – ליצור מחלקת client כללית, ש-RegularClient ו-PremiumClient יעתיקו ממנה הכול. במחלקת PremiumClient צריך ליצור את getBenefits. כך זה נראה:

```
class Client {
  constructor(name, address) {
    this.name = name;
    this.address = address;
  }
  getClientObject() {
    return {
      clientName: this.name,
      address: this.address
    }
  }
}
class RegularClient extends Client {
}
class PremiumClient extends Client {
```

```
  getBenefits() {
    return 'A lots of premium stuff';
  }
}
```

```
const regular = new RegularClient('Moshe', 'Tel Aviv');
console.log(regular.getClientObject()); // {clientName: "Moshe", address: "Tel Aviv"}

const premium = new PremiumClient('ran', 'Petah Tiqwa');
console.log(premium.getClientObject()); // {clientName: "ran", address: "Petah Tiqwa"}
console.log(premium.getBenefits()); // 'A lots of premium stuff';
```

אפשר לראות שהקוד נראה הרבה יותר אלגנטי ושאין חזרות. בעצם, RegularClient לוקח את Client ומשכפל אותו. ברגע שעושים extends, כל המתודות והתכונות של Client נכנסות ל-RegularClient ול-PremiumClient באופן אוטומטי. אפשר להוסיף להן מתודות חדשות.

זה אולי נראה לכם כמו קסם, אבל זה לא. זה בדיוק מה שלמדנו בפרק על אובייקטים – רק באריזה חדשה. אם אתם מתקשים בתרגילים, כדאי לחזור שוב לפרק על האובייקטים.

תרגיל:

כתבו מחלקה של מכונית המקבלת בפונקציה בנאית 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 למחלקה "מכונית" פונקציה המחזירה את מספר הזיהוי.

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

הסבר:

באמצעות המילה השמורה Class יוצרים מחלקה בשם Car. בפונקציה הבנאית מקבלים את שלושת הארגומנטים הנדרשים ונוצר modelNumber. שימו לב ל-this – ברגע שמשתמשים בו, this.modelNumber זמין לכל המחלקה. יוצרים מתודת get מסודרת כדי שתחזיר אותו.

למטה משתמשים במחלקה עם הנתונים. אפשר ליצור עכשיו איזו מכונית שאתם רוצים.

לשם השוואה – הסתכלו על התרגיל הראשון בפרק על new ו-this. זה אותו תרגיל בדיוק והתשובה דומה מאוד לתשובה הזו, אלא שכאן השתמשנו בקלאס. כאמור, המילה קלאס והטכניקה האלגנטית יותר מבלבלות מתכנתים שמסתכלים על כל העניין של המחלקות כעל וודו, אבל ברגע שמבינים שזה ציפוי מעל משהו שכבר היה קיים בשפה – הכול נראה פשוט יותר.

ומה עכשיו?

למדנו לא מעט על ג'אווהסקריפט בספר הזה. התחלנו בעצם ממשתנים בסיסיים ומסוגי מידע פרימיטיביים; המשכנו הלאה אל פעולות ואל זרימת קוד באמצעות משפטי תנאי ופונקציות; למדנו גם על מערכים ואובייקטים ועל לולאות ושינויי מערכים ואיך לארגן את המידע; העמקנו הלאה בקוד באמצעות לימוד משמעותי של אובייקטים מובנים ואיך ג'אווהסקריפט מתנהלת בסביבת דפדפן; ובפרקים האחרונים למדנו איך עובדים בקוד אסינכרוני. זה לא מעט ללמוד, במיוחד אם לא ראיתם שפת תכנות קודם לכן.

אם עברתם על כל פרקי הספר כסדרם, יש לכם ידע תיאורטי מבוסס על עקרונות ג'אווהסקריפט והתחביר שלה. אתם אמורים לדעת לכתוב קוד פשוט, לנתח קוד מורכב יותר ולפתור בעיות. על אף שהבסיס שלכם יציב וטוב, עדיין נדרש תרגול על מנת לחזק את הידע שלכם בשפה ולהעשיר אותו. אף אחד לא הופך למתכנת בעקבות קריאת ספר, טוב ככל שיהיה.

אם אתם רוצים להמשיך ולהרחיב את הידע התיאורטי שלכם, אתם מוזמנים להמשיך לספרי ההדרכה הקצרים הנוספים שכתבתי: על jQuery, על ריאקט ועל Node.JS (ג'אווהסקריפט בצד השרת). אם אתם לא יכולים לרכוש את הספרים האלו, 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 לעבור למעשה ולכתיבת קוד שעושה משהו, וזה הצעד הכי קשה. איך בדיוק עוברים ממצב שבו אני יודע (תיאורטית) לכתוב קוד ג'אווהסקריפט למצב שבו אני כותב משהו שעובד ושאנשים משתמשים בו?

הדרך הכי טובה היא לבנות ולכתוב. השאלה היא מה. פה שום ספר או מדריך כבר לא יעזרו לכם וזה תלוי בכם ובצרכים שלכם. ג'אווהסקריפט היא השפה הפופולרית ביותר בעולם ומשתמשים בה בהרבה מאוד מקומות: באתרי

אינטרנט, באפליקציות דסקטופ, במובייל, בשרתים ואפילו לצורכי אבטחת מידע. אחרי שלמדתם את אחד מהנושאים האלו או את כולם, כדאי להתנסות בקוד ממשי.

המקום הטוב ביותר להתחיל הוא באמצעות תרומת קוד לפרויקטי קוד פתוח שאפשר למצוא ב-GitHub. עבודה במסגרת פרויקט קוד פתוח כזה יכולה להעשיר ולהעמיק מאוד את הידע שלכם. בחרו פרויקט מבוסס ג'אווהסקריפט ונסו לתרום לקוד שלו. באמצעות פתרון באג שמתועד בפרויקט, באמצעות תוספת לדוקומנטציה, לבדיקות האוטומטיות או אפילו באמצעות הוספת תכונה מסוימת. נסו להתקין קומפוננטת ריאקט על אתר של חבר, לבנות תוסף קטן ל-jQuery עבור האתר של העמותה החביבה עליכם, לתרום לדוקומנטציה של קומפוננטה שאתם אוהבים במיוחד, להגיע להאקתון ולהצטרף לצוות קיים או אפילו רק לשבת ולהסתכל על מתכנתים עובדים. אפשר ואף רצוי להתנדב בעמותה ולהתנסות בעבודה בפרויקט ממשי שאנשים ישתמשו בו. יש הרבה דרכים – איזו דרך תבחרו? זה תלוי בכם.

יש לא מעט מיטאפים ופגישות של מתכנתים שכדאי להגיע אליהם. במפגשים האלו פוגשים מתכנתים שעובדים בחברות מגוונות, והאווירה נעימה מאוד. אפשר ורצוי לשאול את האנשים האלו איך מתקדמים הלאה בלימוד. יש גם קבוצות פייסבוק בנושא, שהאנשים הנחמדים והמקצועיים בהן יכולים לייעץ לגבי המשך הדרך.

באתר הזה יש רשימת של מקורות ושל קבוצות שאני משתדל להיות חבר בהן, אני ממליץ לכם להסתכל בה ולבדוק את הקבוצות ואת המקורות השונים: https://github.com/barzik/web-dev-il-resources

כך או אחרת, חשוב מאוד להמשיך לכתוב בשפה. כאמור, שפת תכנות אינה שונה מהותית משפה מדוברת; אם לא תשתמשו בה, תשכחו אותה. אפשר להשתמש בג'אווהסקריפט במגוון עצום של פרויקטים – רק צריך לבחור פרויקט ולהמשיך לתרגל. להשתלב בתחום ההייטק זה לא בשמיים וזה אפשרי. אם אני הצלחתי – כל אחד יכול.

נספח: Best Practices

מחברים: שחר טל, רוני אורבך, דניז רוזג, נופר ברנס, חברת Really Good

מה זה Best Practices ולמה כדאי ליישם אותם?

הם כללים, המלצות ומוסכמות שנועדו לסייע ליצור תוכנות איכותיות, בין השאר באמצעות Best Practices כתיבת קוד בצורה טובה ונכונה שממזערת טעויות ובלבול.

תכנות הוא פעולה חברתית. כשחושבים על תכנות, לרוב מדמיינים האקר בודד שיושב בחושך בקפוצ'ון ומקליד במרץ בלי לחשוב שנייה, כאילו הקוד עף מתוכו. הסיטואציה הזאת היא נדירה, ובינינו, זה די לא נוח לשבת לתכנת ככה. לרוב תוכנות נכתבות עם קולגות במקום העבודה או עם חברים או אפילו עם אנשים זרים באינטרנט – והיצירה שלהן דורשת קצת יותר סבלנות ושיקול דעת. מתכנתים אחרים יסתכלו על הקוד לפני שהוא יוכנס באופן סופי לפרויקט (אדבר על Code Review בהמשך), ומתכנתים אחרים יצטרכו להבין ולעבוד איתו בעתיד כשירצו להוסיף, לשנות או לתקן את התוכנה.

לאורך הזמן מתכנתים עוזבים צוותים ואחרים צריכים להתמודד עם קוד שהכותב שלו כבר לא בסביבה, וגם אתם בטח תרצו להשאיר אחריכם קוד שמתכנתים אחרים יוכלו ואפילו ישמחו לעבוד איתו. אפילו בפרויקט שכולו שלכם, ככל שהזמן עובר גדל הסיכוי שיום אחד תסתכלו על קטע קוד ותשאלו את עצמכם "מי לעזאזל כתב את זה?" והתשובה הטרגית-קומית תהיה – אתם בעצמכם. לכן חשוב לשמור על קוד מסודר ומאורגן שנכתב בדרך מוסכמת. כך מצמצמים באגים וטעויות, מקילים על כולם את קריאת והבנת הקוד במהירות וחוסכים רעשי רקע וויכוחי סרק על שטויות – מחליטים פעם אחת על סגנון ומדיניות ונמנעים מ"מלחמות עריכה" אינסופיות שבהן כל אחד מסדר את הקוד איך שבא לו והבא אחריו מבטל את השינויים.

אפשר להסכים על כללים כאלו בעל פה, אפשר בכתב, ויש כלים שאדון בהם בהרחבה ממש כאן, שיכולים לעזור לאכוף לפחות חלק מהכללים באופן אוטומטי.

במיוחד בשפה גמישה כמו ג'אווהסקריפט, שמאפשרת לעשות המון דברים יצירתיים ובתוכם המון שטויות, מומלץ לעקוב אחרי Best Practices מקובלים בתעשייה, שאולי לא נכתבו בדם אבל בהחלט עלו ביזע ובדמעות. מעבר למניעת שגיאות ולשימוש מושכל באפשרויות שהשפה מציעה, חלק מה-Best Practices הם העדפות סטייליסטיות

של המתכנתים בפרויקט, שהתקבעו ושהצוות רוצה לשמר – בשני הסוגים יש מקום לשיקול דעת בריא, ולא צריך לקחת הכול כתורה מסיני בלי להבין ולבחון מה מתאים לכם.

דוגמה להמלצה שמונעת טעויות היא תמיד להשתמש בהשוואה קפדנית, כלומר ב-=== או ב-!== במקום ב-== המתירנית יותר, שמבצעת המרה לסוג המשתנים לפני השוואה (type coercion).

לא מומלץ:

```
if (numberOfThings == possiblyNotANumber) { /* code */ }
```

מומלץ:

```
if (numberOfThings === possiblyNotANumber) { /* code */ }
```

דוגמה לכלל סובייקטיבי יותר, שפשוט מבטא העדפה סגנונית ועוזר לשמור על אחידות, היא ריווח בין חלקי קוד שונים, למשל ריווח בין שם פונקציה לבין הסוגריים שמגדירים את הארגומנטים:

אופציה א':

```
function jump(x) {}
```

אופציה ב':

```
function jump (x) {}
```

אופציה ב' היא המועדפת כי היא מאפשרת להבדיל בחיפוש בין הגדרת הפונקציה לקריאות אליה.

סט הכללים וההמלצות יכול להיות שונה מאוד בין פרויקטים שונים, ובעבודת צוות יש חשיבות בשם העקביות והשפיות להתגבר על ההעדפה האישית שלכם ולכבד את מה שכבר נקבע – מומלץ לדבר בפתיחות ולבחון ביחד צורך בשינויים.

Best Practices שכל מפתח מקצועי צריך להכיר

בחירת שמות

משתנה או פונקציה עם שם טוב יאפשרו לכם להבין מייד מה התפקיד שלהם. אם השם לא מוצלח 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 שמוטב לא לפזר הערות ליד כל מקום שבו משתמשים במשתנה או בפונקציה.

איך בוחרים שם טוב? שיטה מקובלת עבור פונקציות היא שיטת ה-verb-noun (פועל–שם עצם).
לדוגמה:

- getStudents
- sortColorsByBrightness
- setLunchBreakTimer

`setLunchBreakTimer` למשל הוא שם יותר טוב מ-`setTimer`, שעלול להיות כללי מדי, וכשנתקלים בו צריך להיזכר באיזה טיימר מדובר.

יש גם יוצאי דופן בולטים – שמות משתנים סטנדרטיים שצריך להכיר ואין צורך להסביר או לפרט אותם. דוגמה לשמות כאלה הם `x` ו-`y` בהקשר של פיקסלים או כיוונים על המסך, או `i` ו-`j` בשימוש הקלאסי שלהם בלולאות. אבל חוץ מהם, כדאי לזכור שכמעט תמיד משתנה של אות או שתיים זה רעיון רע. אין שום יתרון בראשי תיבות מסתוריים על פני מילים מובנות.

שם המשתנה הוא לא הדבר היחיד שעוזר לקריאות הקוד. אפשר לכתוב אותו שם בכמה דרכים בעזרת אותיות גדולות וקטנות. כל דרך כזו נקראת "קייס", ומיועדת לייצג סוג משתנה אחר.
אומנם השם זהה, אבל על כל אחד מהמשתנים האלה נוסף רובד של משמעות שנרמז רק מבחירת הקייס:

camelCase
lunchBreakTimer
כל המילים מלבד הראשונה מתחילות באות גדולה; אין רווחים בין המילים.
קייס זה נמצא בשימוש עבור כמעט כל המשתנים והפונקציות בג'אווהסקריפט.

PascalCase / TitleCase
LunchBreakTimer
כל המילים מתחילות באות גדולה; אין רווחים בין המילים.
קייס זה נמצא בשימוש עבור קלאסים, וככה מקל את ההבחנה בין קלאס ל-instance.

SCREAMING_SNAKE_CASE / UPPER_CASE
LUNCH_BREAK_TIMER
כל האותיות גדולות; רווחים בין המילים הופכים לקו תחתון.
קייס זה נמצא בשימוש בעיקר עבור 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 id, קלאסים ואלמנטים.

קייס זה לא נוח לשימוש בג'אווהסקריפט כי קו מפריד אינו תו חוקי בשמות משתנים, כך שקשה עד בלתי אפשרי להשתמש בו ברוב המקרים.

snake_case

lunch_break_timer

לא בשימוש בג'אווהסקריפט בדרך כלל.

KISS

"Keep It Simple, Stupid", אמירה שקלעה בול כשנטבעה לפני יותר מחמישים שנה על ידי מהנדס המטוסים קלי ג'ונסון בלוקהיד, ועדיין מתאימה כהנחיה למתכנתים של 2020 והלאה. כתבו את הקוד בדרך הפשוטה והקצרה ביותר, כל עוד הקריאות נשמרת. למשל, הימנעו מסינטקס אזוטרי שלא נמצא בשימוש רחב ומהתחכמות יתר בכלל.

DRY

Don't Repeat Yourself. אין סיבה לכתוב את אותו קוד יותר מפעם אחת. לפעמים זקוקים לאותה פונקציונליות בכמה מקומות אבל מהססים להפוך אותה לפונקציה מכיוון שמדובר בקוד פשוט וקצר או בקוד שדורש שינוי קטן בכל שימוש שלו.

אל תהססו. כשעוטפים את הקוד בפונקציה גם נותנים לו שם, שהופך את השימוש בו לקריא יותר. ככל שמשכפלים יותר קוד שלא לצורך, מגדילים את הסיכון שהמתכנת הבא יפספס חלק מהמקומות שבהם יש קוד כפול, והתחזוקה תהפוך לקשה ורצופת תקלות.

זכרו – ההפך מ-DRY הוא WET, ראשי תיבות של Write Everything Twice ;-)

לא להמציא את הגלגל

למרות שתכנות זה כיף ומעניין, לא בכל דבר שבונים צריך להקים הכול מאפס. וכמו שכנראה ברור כיום שתשתמשו בספרייה כלשהי שתעזור לרנדר את הממשק, נניח React, ככה חשוב להשתמש גם בספריות ובכלים קטנים וממוקדים יותר כשזה הגיוני. מהוצאת בקשות לשרת עם axios, דרך ספריות לפעולות נפוצות על מערכים

ואובייקטים כמו lodash ועד לאלפי הרכיבים המוכנים לרוב חלקי הממשק שתרצו לבנות – תהיו בטוחים שמישהו כבר פיתח פתרון טוב כמעט לכל צורך. הבה נודה באמת – אתם לא הראשונים שכותבים Color Picker או Dropdown עם השלמה אוטומטית, וגם אלגוריתמים לערבוב רשימות כבר נכתבו בעשורים שקדמו להצטרפות שלכם לתחום.

יש שלושה מקרים שבהם דווקא הגיוני להיכנס לעובי הקורה ולכתוב בעצמכם משהו לא אלמנטרי:

1. אם אין פתרון מוכן שעונה על כל הדרישות שלכם (עדיין זכרו שלעיתים יש רכיבים קטנים יותר בתוך המכלול שיוכלו לעזור לכם או שתוכלו ללמוד 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 כלי מוכן לא יספיק לכם – למשל במסגרת העבודה ב-Really Good אנחנו עובדים עם פלטפורמת וידיאו מובילה, אז כשהתבקשנו לפתח נגן וידיאו חדש יצרנו אותו מאפס על בסיס תגית ה-video של HTML, בלי אף ספרייה חיצונית לניגון וידיאו.
3. אתם רוצים לתרגל או נבחנים על פיתוח עצמאי של משהו כזה. בהצלחה!

בשולי הדברים, שימו לב לרישיונות בקוד פתוח – רישיונות שונים מאפשרים שימוש בתנאים שונים, וחלקם מגבילים שימוש מסחרי. אם אתם חלק מארגון גדול ולא בטוחים, כדאי לוודא שרישיון השימוש מקובל על הגורם המשפטי בחברה. בכל מקרה תמיד טוב לקרוא ולהכיר את הרישיון של הקוד שמשתמשים בו, גם באופן פרטי.

Make it work, make it right, make it fast

כאשר כותבים תוכנה, גם קטנה אבל בעיקר גדולה – אי אפשר לכתוב את כולה בצורה מושלמת במכה אחת. מתחילים מגרסה בסיסית שפשוט עובדת עם הפונקציונליות הבסיסית. אחר כך חוזרים לעבות ולכסות את כל הדרישות, ומשפרים את הקוד כך שיהיה קריא יותר, תמציתי ומובן ככל האפשר. אל תתפתו להקדים את המאוחר ולחטוא ב"Premature Optimization" – אופטימיזציה שומרים לסוף.

תיעוד

כבר למדתם מוקדם יותר בספר על הערות בקוד. שני השימושים הכי נפוצים בהערות הם:

4. כיבוי זמני של חתיכת קוד בלי לאבד אותה. נוח תוך כדי פיתוח ובדיקות, אבל שימו לב שהערות הן לא שיטה לשמור גרסאות של קוד – מייד אצלול לניהול גרסאות נכון.
5. תיעוד – להסביר מה מצפים שיקרה או למה פיתחתם משהו בצורה מסוימת. תיעוד טוב מוסיף מידע, ולא סתם חוזר על מה שהקוד עושה. אפילו כמה מילות הסבר עדיפות על כלום. הערה שכתובה טוב יכולה לחסוך למתכנת אחר הרבה כאב ראש. כשאתם ניגשים לכתיבת התיעוד, נסו לחשוב מה הדבר הכי מורכב

או לא ברור בקוד שלכם ושימו את הדגש שם. מתכנתים אחרים יודו לכם, ואפילו אתם תודו לעצמכם כשתחזרו לעבוד על קוד ישן ומתועד 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 כזה – 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 מהצוות ירצה לשנות קוד הוא יצטרך לברר אצל מי הקובץ החדש ביותר ולבקש ממנו שישלח לו את הקובץ. באיזו שנה אנחנו?

כדי להתגבר על כל הבעיות 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, לרוב ב-github.

שימוש בניהול גרסאות מאפשר למפתחים מרובים לכתוב קוד בכמה ענפים (Branches) שונים במקביל ולשתף אותו ביניהם. אפשר לראות איזה מפתח כתב מה ומתי, אפשר לשלב כמה ענפים מוכנים אל תוך גרסה אחת שאותה משחררים, והכי חשוב – אפשר לחזור אחורה במידת הצורך.

לבקש עזרה

כשנתקעים במהלך פיתוח – לבקש עזרה זו לא בושה. אם אחרי Debugging, בידוד הבעיה ובדיקת הלוגים (אם יש כאלה) עדיין לא צלחתם את הבעיה, עוברים לגוגל. מנסים לחפש את השגיאה בצורה הכי נקייה שאפשר, בלי שמות קבצים או מספרי שורות מהקוד הספציפי. למרבה המזל, רוב הבעיות 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 מכירים את הבעיות הנפוצות והאתגרים שאתם עשויים להיתקל בהם ויוכלו להכווין אתכם ולחסוך לכם זמן יקר. שיתוף הבעיה עם חברי הצוות גם גורם לעבור על הקוד פעם נוספת ואפשר למצוא את הפתרון תוך כדי תיאור הבעיה – מניסיון, זה עובד.

אם בכל זאת לא מצאתם תשובה בגוגל או אצל המתכנתים האחרים בצוות, זה הזמן לשאול את השאלה שלכם אונליין. האתר הכי פופולרי לשאלות תכנות הוא StackOverflow, שמנגנון דירוג התשובות בו עוזר לזהות מה מבין התשובות עובד ורלוונטי בעיני הקהילה, בניגוד לפורומים שגם הם רלוונטיים אך בחיפוש פתרונות בהם נדרשת קריאה יותר ביקורתית. אין מה להתבייש, השאלה שתשאלו תהיה כנראה רלוונטית למתכנתים אחרים בעתיד שייתקלו באותה בעיה ויחפשו תשובה.

בדרך לפרסום שאלה תצטרכו להכין דוגמה ממוקדת ומבודדת של הבעיה (מכונה בדרך כלל Reduced Test Case), וגם זה תהליך מומלץ שעצם ההשקעה בו עשוי להוביל אתכם לפתרון עוד לפני שתפרסמו משהו.

ביקורת עמיתים – Code Review

סקרתי הרבה כללים, וזה יכול להיראות מאיים. האם באמת מקפידים על כולם בכל צוות פיתוח?
התשובה היא שאמנם לא מקפידים במאת האחוזים, אבל מפתחים וארגונים מקצועיים יתעקשו לעמוד בכמה שיותר מהם, כי מוצרים הולכים ומתפתחים וככל שפרויקט מתעבה ונוספות עוד שכבות של קוד כך נהיה קשה לתחזק אותו ולהבין מי נגד מי. אם מראש היסודות רעועים ואף אחד לא מקפיד על Best Practices, מהר מאוד תמצאו את עצמכם מול "קוד ספגטי" (שמעורבב כמו ספגטי בצלחת) שקשה מאוד לתחזק.

אם זכיתם לעבוד בצוות ולא לבד, Code Review עשוי להיות חלק מהאחריות שלכם, וזה מצוין. העבירו את הקוד שלכם לבדיקה אצל מתכנת אחר והימנעו מהסברים מיותרים. תנו קצת רקע כללי על ההקשר של הקוד ומה ניסיתם להשיג ואפשרו לקוד לדבר בעד עצמו. כך, אם חסר תיעוד – זה יבלוט. למי שעובד לבד, יש אתרי אינטרנט וקבוצות שבהם אפשר להתייעץ, לשתף קטעי קוד ולבקש ביקורת.

את ההערות על הקוד מקבלים ונותנים בשתי דרכים. הדרך הראשונה היא בעל פה (פנים אל פנים או בטלפון). הדרך השנייה היא בכתב, בצמוד לשינויים במערכת ניהול הגרסאות. לדוגמה, אם עובדים עם Git, אפשר להשאיר הערות על Pull Request. בקשה זו היא בעצם בקשה דרך מערכת ניהול הגרסאות לאשר את הקוד ולמזג אותו לתוך הענף הראשי של הפרויקט. מתכנתת שבודקת קוד תוכל להשאיר עליו הערות, ולאחר שהמפתח יבצע את השינויים היא תוכל לאשר את הבקשה כדי למזג את שני הענפים.

כשאתם מקבלים ביקורת, קבלו באהבה הצעות, הערות ושאלות. אל תיקחו את הביקורת כתקיפה אישית נגדכם. זכיתם בעוד זוג עיניים שיעברו על הקוד שלכם, לא משנה למי מכם יש יותר ניסיון. קחו אותה בשתי ידיים, הקשיבו באמת ואל תרוצו למגננה – אפילו אם עבדתם קשה על משהו ואולי קצת התאהבתם בפתרון מסוים או נקשרתם אליו, או אם הערה שקיבלתם תדרוש שינוי יחסית גדול והרבה עבודה מצידכם. זכרו, באתם לעבוד ביחד ולייצר קוד איכותי וברור, לא להראות מי יותר חכם וצודק.

כשאתם עורכים למישהו אחר Code Review – נסו להבין מה קורה בקוד ולהעריך אם תהיו מסוגלים להרחיב או לשנות אותו במידת הצורך. בחנו את הקוד אל מול עקרונות מנחים כמו כל אלו שמוזכרים בפרק הזה. האם השמות הגיוניים וברורים? הפונקציות או הקבצים ארוכים מדי? יש קוד שחוזר על עצמו והיה אפשר להפוך ליותר DRY? הקוד מסודר ומעומד בצורה עקבית וקריאה? וכן הלאה.

בסופו של דבר, בכל יום לומדים משהו חדש. Code Review היא דרך מצוינת ללמוד מהאנשים סביבכם ולהמשיך להתפתח בתור מתכנתים. זה הרגל מעולה שיחדד אתכם כמתכנתים לזהות קוד פחות טוב ולכתוב קוד יותר טוב. כולם מרוויחים.

Tech Design

כשניגשים למאמץ פיתוח רציני, סוף מעשה במחשבה תחילה. כמה שמנסים לקבל באהבה ביקורת, כמעט תמיד מתבאסים לשמוע בסוף התהליך שהיה עדיף להשתמש בספרייה מסוימת שכבר קיימת ולא ידעתם עליה או שאי אפשר ללכת על פתרון שסיימתם לפתח משיקולים שונים שלא הכרתם. לפעמים אפילו תיתקעו עם הפתרון הלא-מושלם שלכם כי מאוחר מדי לחזור אחורה ולתקן.

כדי להימנע ממצבים כאלה, ראוי לעשות תכנון טכני, שהוא מסמך בפורמט גמיש. יש אזורי התמחות שבהם הגיוני להשתמש בתרשימי זרימה (למשל תכנון של ארכיטקטורת שרת או מסדי נתונים); כשעובדים גם בצד לקוח וגם בצד שרת מקובל לתאר את ה-API שמצפים לבנות לקריאות ביניהם – לאן שולחים כל בקשה, אילו פרמטרים מעבירים ואיך בדיוק תיראה תגובה טיפוסית.

בעזרת Tech Design טוב אפשר לוודא שמבינים זה את זה ולהתחיל לפתח את הצדדים במקביל, ולא לגלות בסוף שכל אחד תכנן מבנה נתונים שונה לחלוטין ולהתווכח מי צריך להתאים את עצמו לָאחר.

את התכנון הזה תעבירו לביקורת עמיתים, דברו עליו וצאו לדרך רק אחרי שהסכמתם על הפרטים – ככה תמנעו לפחות חלק מההפתעות הגדולות והיקרות בשלבים מאוחרים יותר.

Best Practices ואוטומציה

אז כתבתם Tech Design וקיבלתם אישור לצאת לדרך, עבדתם קשה ואז הגיעה ערמה ענקית של הערות ב- Code Review. איך בכלל מתחילים לעבור על זה? ועם כל הכבוד וההערכה, איך לא לוקחים את זה קשה כשאולי אתם מרגישים ש"מחפשים אתכם" ושכמעט על כל שורה יש למישהו מה להגיד?

עבדתם יפה! לדברים יש נטייה להתבלגן כי כולם בסך הכול אנושיים. היו לכם כוונות טובות, אבל יכול להיות שפה שכחתם להיות עקביים בסגנון, שם העדפתם להעמיס יותר מדי על פונקציה אחת, ומדי פעם לא הקפדתם על משהו מהכללים המומלצים. זה טבעי, אבל כשמכפילים את הנטייה האנושית לעגל פינות ולפספס פרטים במספר אנשי צוות ובכמה חודשי פיתוח, האיכות מידרדרת מהר מאוד.

אחד הדברים שהכי עוזרים בריכוך המעמד של הביקורת, ובכלל בשמירה על איכות על ידי איתור ומניעת שגיאות מראש, הוא אוטומציה. יש כלים אוטומטיים מעולים, שאציג בהמשך, שעוזרים לשמור על סדר ולהקפיד על הכללים שתקבעו לעצמכם, לבד או כצוות.

כך, כשתגיעו ל-Review תדעו שבחלק נכבד מההערות פשוט לא תיתקלו, כי כבר קיבלתם פידבק מיידי או תיקון אוטומטי בזמן אמת. זה כבר מאחוריכם, וחסכתם גם מהעמיתים וגם מעצמכם דיונים על ריווחים, שורות ארוכות מדי ועוד המון הערות, מהותיות יותר או פחות. יהיה קל, מהיר ונעים יותר להתמקד ולהקשיב לביקורת שכן תגיע כשהיא תהיה מצומצמת יחסית.

בצוות, מומלץ מאוד להחליט פעם אחת (מוקדם ככל האפשר) על הקווים המנחים שלכם, להיעזר בכלים האוטומטיים שתלמדו עליהם בחלק הזה וליצור מערכת אוטומטית שתפקח על הקוד שנכנס לפרויקט. למרבה השמחה, מתכנתים אחרים כבר בנו מערכות כאלה, ואף נתנו לפעולת האוטומציה של פיקוח על הקוד שם – Linting.

מה זה linting?

המילה lint (לינט) באנגלית פירושה מוֹדִ – הלכלוך המצטבר בכיס של בגד או הכדורים הקטנים שנוצרים על גבי הסריג לאחר כמה כביסות. בעת כתיבת ועדכון קוד מתווספים לאט לאט שגיאות או דפוסים לא רצויים אחרים שנוגדים את ה-Best Practices שקבעתם – זהו המוך, ולכן יש לעבור על הקוד עם כלי שמסיר את המוך; בהשאלה, קוראים לו linter. לינטרים יודעים לזהות את השגיאות והדפוסים שהגדרתם, להתריע על הימצאותם 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 ההפעלה Unix.

עורכי טקסט מודרניים המיועדים לכתיבת קוד ג'אווהסקריפט מפעילים בודק שגיאות מובנה שמזהיר את המשתמש משגיאות שימנעו מהקוד לרוץ בצורה תקינה בדפדפן. אם כן, למה עדיין צריך לינטר? קוד יכול לפעול באופן תקין ועדיין לא לעמוד ב-Best Practices ובכללים שקבעתם מסיבות שונות, ולכן רוצים 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)

לינטר יכול להזהיר מפני בעיות כמו שימוש במשתנים שלא הוגדרו, קריאות לפונקציות שאינן קיימות, הפרת מוסכמות בנוגע לריווח וסידור של קוד, שימוש בפיצ'רים של השפה שנוטים להוביל לבעיות אבטחה ועוד. הלינטר יסמן בעיות באמצעות סימון בקו מזוגזג אדום לשגיאות וירוק לאזהרות, בדומה לסימון של שגיאות כתיב במעבדי תמלילים. במעבר עכבר מעל הקוד המסומן אפשר יהיה לראות את הסיבה ולעיתים גם הצעות לשיפור הקוד.

הכלי הראשון בעולם הג'אווהסקריפט שעשה זאת היה JSLint, שאכף סט מוגדר מאוד של כללים בצורה קשוחה – מי שלא אהב את הדעות החותכות של היוצר שלו, אושיית הג'אווהסקריפט דאגלס קרוקפורד, נאלץ להשתמש בכלי אחר בשם JSHint, שכן אפשר יותר גמישות.

עם השנים וההתפתחות המהירה של ג'אווהסקריפט, ESLint של ניקולס זאקאס תפס תאוצה, בין היתר בזכות מבנה מאוד גמיש. ESLint מאפשר להגדיר ברמה פרטנית את הערכים הרצויים לכללים קיימים (כמו למשל אורך שורה מקסימלי או איפה בדיוק מותר או אסור להשאיר שורות ריקות), אבל גם להגדיר כללים משלכם, ולהשתמש בקלות בכללים שאחרים יצרו, כמו שארחיב עוד מעט. נכון לעכשיו, ESLint הוא ה-Linter הפופולרי ביותר עבור ג'אווהסקריפט, ולכן אתמקד בו.

ESLint

עורכי ג'אווהסקריפט מודרניים יודעים להציג הערות של ESLint בצורה אוטומטית או בהתקנה פשוטה של תוסף. האזהרות והשגיאות שיוצגו נקבעות על פי קובץ הגדרות, בדרך כלל בשם eslintrc, בתיקייה הראשית של הפרויקט. אם אצלכם עדיין אין קובץ כזה, קראו על `ESlint --init` וצרו באמצעותו קובץ הגדרות משלכם – מומלץ להתחיל מאחד הסטים הקיימים שיוצעו לכם במהלך האתחול.

אם בחרתם סט חוקים שנוח לכם להשתמש בו אבל יש בו כמה חוקים שלא נראים לכם, אל תהססו להיכנס לקובץ ההגדרות ולשנות אותם.

בחלק הבא אעבור על כמה Best Practices, חלקם חדשים וחלקם כבר הזכרתי קודם, לצד הכלל ב-ESLint שמסייע לציית להם.

רשימה של Best Practices והחוק הרלוונטי של ESLint

בלי מספרי קסם

שם הכלל: no-magic-numbers

"מספרי קסם" הם מספרים שמופיעים בקוד בלי הסבר מפורש, כמו הצבה למשתנה שהשם שלו יוכל להאיר עיניים. הצורך בולט בחישובים, ובייחוד בחישובים שמשלבים כמה מספרים. ברגע הכתיבה ברור איך מגיעים אליהם ויום או חודש אחר כך כבר מגרדים בראש ולא מבינים מה הסיפור של המספר הזה.

דוגמה לקוד בעייתי עם מספר קסם:

```
setTimeout(ringBuzzer, 180000);
```

למה 180,000 בעצם? בלי הערה שתספר למה התכוון המשורר, קשה לקלוט במהירות במה מדובר. אפשר להוסיף הערה כמו "three minutes from now", אבל בלעדיה לוקח זמן להבין מה אומר המספר הזה ואיך לשנות אותו אם ביקשו מכם להאריך את ההמתנה לחמש דקות. מחשבים מהירים מאוד בחישוב, לא צריך לרחם עליהם וללעוס עבורם את המספרים מראש. חשוב יותר שהקוד יהיה מובן לכם ולשאר בני האדם שיצטרכו להבין את הקוד, אז מפרקים את המספר הלא מוסבר לגורמים:

```
const MS_IN_1_SECOND = 1000;
const SECONDS_IN_1_MINUTE = 60;
const DELAY_MINUTES = 3;
const BUZZER_DELAY = DELAY_MINUTES * SECONDS_IN_1_MINUTE *
MS_IN_1_SECOND;
setTimeout(ringBuzzer, BUZZER_DELAY);
```

השוואה קפדנית: ===

שם הכלל: **eqeqeq**

זה **ה**כלל של לינטרים בג'אווהסקריפט, עוד מ-JSLint המקורי. ג'אווהסקריפט מאפשרת להשוות בין ערכים מסוגים שונים. זו תכונה מעניינת שמאפשרת גמישות ויכולה להיות שימושית, אבל היא בקלות עלולה לבלבל ולהטעות.

כדי להימנע מטעויות קשות לאיתור כתוצאה מההשוואה המתירנית, מקובל ומומלץ להקפיד על שימוש בהשוואות הקשוחות יותר. מומלץ להשתמש ב-=== לבדיקת שוויון וב-!== לבדיקת אי שוויון. כך משווים בין ערכים בלי להמיר את סוג המשתנים תוך כדי השוואה.

קוד לא נגיש

שם הכלל: no-unreachable

תוך כדי refactor ו-debugging, קורה שנשארת פקודה שיוצאת 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 ירוץ, ו-ESLint תדגיש את הקטע הלא נגיש:

```
function validateEmail (email) {
    return true;
    if (email.length) {
        let result = true;
        // if (!email...) {}
        // TODO complete validation
    }

    return result;
}
```

חלוקה לחלקים קטנים

שמות הכללים: max-len, max-lines-per-function, max-lines

קשה לתפוס הרבה לוגיקה ברצף אחד, וכמו שבכתיבת ספר או מאמר מומלץ להשתמש בנדיבות בפסקאות, כותרות ועמודים, כך גם בקוד חשוב לא להתפתות להעמיס המון קוד על שורה ארוכה מדי, פונקציה אחת שעושה הכול או קובץ אחד של אלפי שורות.

החוקים הבאים הם חוקים שאין עליהם מוסכמות, אך הם יכולים לעזור לכם בכך שישמשו תמרור אזהרה שהקוד שאתם כותבים הפך לארוך ומסורבל. אם תבחרו להשתמש בהם כדאי שתגיעו להסכמה עם חברי הצוות שלכם בנוגע לגדלים שמקובלים על כולם.

אורך שורה (מספר תווים מקסימלי בשורה)

יש להקפיד על אורך שורה שאינו עובר מספר תווים מסוים. הסיבה לכך היא שרוצים שורה שנכנסת ברוחב המסך ללא צורך לגלול הצידה בעת קריאת הקוד. קוד קריא הוא קוד שאפשר לראות את כולו או את רובו בבת אחת. בעבר, כשגודל המסך היה קטן יותר, היה מקובל להגדיר אורך שורה של 80 תווים, אבל בימינו, כשהמסכים הרבה יותר גדולים, נהוגים אורכי שורה של 120 ואף 140 תווים.

אפשר להימנע משורות ארוכות בכמה דרכים. הסיבות העיקריות לשורות ארוכות הן רשימה ארוכה של תנאים וקוד מקונן. רשימה ארוכה של תנאים קל לשבור לכמה שורות. קוד מקונן יותר קשה לתקן, אך אם דואגים לשמור על פונקציות קצרות ופשוטות מצליחים למנוע זאת ברוב המקרים.

אורך פונקציה (מספר שורות מקסימלי בתוך פונקציה)

אין מוסכמה בנוגע לאורך פונקציה למרות שכולם מסכימים שפונקציות ארוכות זה רע. החוק הזה טוב בעיקר כדי להזכיר לכם שהגזמתם, לכן אם תשתמשו בו כדאי לבחור מספר שנוח לכם. אם אינכם בטוחים איזה מספר לבחור, התחילו 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 עושה יותר מדי אפשר לקבל כשתנסו לתת לפונקציה שם. אם במהלך הניסיון תגלו בשם הפונקציה את המילה "and", תבינו שניסיתם להכניס יותר מדי לתוך פונקציה אחת. ברגע שזיהיתם מקרה כזה, זה הזמן לחלק את הפונקציה לשני חלקים או יותר.

אורך קובץ (מספר שורות מקסימלי בקובץ)

גם כאן אין מוסכמה מקובלת, למרות שרוב המתכנתים יסכימו שקובץ שמתקרב ל-1,000 שורות הוא גדול מדי, בעוד אחרים ידברו על מקסימום של 500 שורות או פחות. כאשר הקובץ גדול מדי קל לאבד בו את הידיים והרגליים ולכן כדאי להגביל את עצמכם.

אורך מינימלי ומקסימלי לשמות משתנים

שם הכלל: id-length

שמות משתנים קצרים מדי או ארוכים מדי הופכים את הקוד ללא קריא. שמות קצרים מדי מקשים להבין מה מטרת המשתנה ושמות ארוכים מדי מסרבלים את הקריאה. כשבוחרים שם למשתנה כדאי להתחשב בקווים המנחים שדנתי בהם תחת הכותרת "בחירת שמות".

בלי eval

שם הכלל: no-eval

כפי שכבר הוסבר מוקדם יותר בספר, פונקציית ה-eval היא פונקציה מסוכנת מבחינת אבטחת מידע. תמיד אפשר וכדאי להימנע משימוש ב-eval באמצעות חלופות שונות לפתרון הבעיה.

בלי משתנים שלא הוצהרו

שם הכלל: no-undef
בג'אווהסקריפט אפשר להציב ערך לתוך משתנה שלא הוצהר קודם. במקרה כזה המשתנה ייווצר על האובייקט הגלובלי (window, במקרה של ריצה בדפדפן)

```
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 לבאגים לא צפויים כתוצאה מחוסר תשומת לב ל-scope שפועלים בו. מכיוון שהשמה לתוך משתנה שלא הוצהר מתבצעת כמעט תמיד בטעות, הכלל עוזר לאתר מקרים כאלו ולהימנע מהם.

לסיכום, כדי ליצור תוכנות איכותיות שהקוד שלהן ברור ונוח לתחזוקה חשוב להקפיד על Best Practices, כמו אלו שסקרתי ברשימה החלקית כאן. הפעילו שיקול דעת בבחירת ובהתאמת הכללים שלפיהם תעבדו, כדי ליצור לעצמכם תהליך עבודה יעיל וקוד ממש טוב לאורך זמן.

נספח: בדיקות, יציבות ואיכות קוד

מחבר: דניאל שטרנליכט, חברת Outbrain

ברכות! אם הגעתם לנספח הזה, כנראה אתם כבר מבינים איך ג'אווהסקריפט עובד ואתם מוכנים להתחיל לבנות אפליקציות ווב. אבל רגע אחד לפני שאתם מתחילים, רציתי לספר לכם קצת איך לוודא שהקוד שאתם כותבים הוא יציב, איכותי, ויוכל לעמוד במבחן הזמן כשמפתחים שאיתם תעבדו יעשו בו שינויים בעתיד.

קצת רקע

באאוטבריין, חברת המלצות התוכן הגדולה בעולם שמגישה המלצות למיליארד משתמשים מדי חודש, עובדים יותר מ-200 מפתחים שכותבים אלפי שורות קוד ומשחררים עשרות גרסאות ביום. האתגר הוא לא קטן, שכן כל שינוי בקוד שמפתח עושה צריך להיבדק ולהיבחן לפני שהוא עובר ללקוחות.

בקנה מידה כל כך גדול, איך מוודאים שקוד חדש שנכנס לא שובר את החוויה שהלקוחות מצפים לקבל? איך מאפשרים למפתחים להמשיך לעשות שינויים בקוד בלי ליצור צוואר בקבוק בתהליך העלאת הגרסאות?

התשובה: בדיקות. המון המון בדיקות. למעשה, לפני כל העלאת גרסה באאוטבריין רצות, באופן אוטומטי, לא פחות מאלפי בדיקות!

מבנה בדיקות

כשכותבים בדיקות, אפשר בקלות להגיע למצב שבו יש כל כך הרבה עד שהולכים לאיבוד. בדיוק בשביל זה קיימים היום כל הספריות והפריימוורקים שעוזרים לסדר את הבדיקות. רוב הספריות מסודרות במתודולוגיה בשם behavior-driven development או **BDD** בקיצור, שמטרתה לסדר את הבדיקות לפי התנהגות כך שייראו פחות או יותר ככה:

```
describe('utils', function() {
describe('string utils', function() {
test('should validate strings are strings, function() {
const foo = ‘hey test’;
expect(foo).toBeString();
})
})
})
```

הפונקציה **describe** מאפשרת לתאר סדרה של בדיקות מסוג מסוים. תחת הפונקציה הזאת אפשר לקרוא לעוד פונקציה מאותו סוג כדי לתאר תת-סדרה. הפונקציה **test** היא הבדיקה עצמה, שתכיל "טענות" (או **Assertions** באנגלית), ופונקציית **expect** היא הטענה עצמה. אם הטענה נופלת, הבדיקה נכשלת.

כשמדברים על בדיקות, יש לא מעט סוגים: Unit Test, Integration Test, End-to-End Test, Sanity Test, Regression Test, System Test וכו', ולכל בדיקה המטרות שלה.

בנספח הזה אתמקד בשלושה סוגי בדיקות שונות:

- בדיקות יחידה (Unit Tests)
- בדיקות קצה לקצה (End-to-End או בקיצור E2E)
- בדיקות ממשקי 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. בדיקות יחידה יעבדו מול פונקציות או מול מחלקות, ולרוב יבדקו שקלט מסוים יחזיר פלט מסוים.

קחו לדוגמה את הפונקציה הבאה, שיודעת לקבל מספר ולהחזיר את המספר מוכפל בעצמו:

```
function multiply(number) {
    return number * number;
}
```

כדי לוודא שהפונקציה תקינה ועושה בדיוק את מה שהיא נועדה לעשות, מריצים כמה בדיקות שונות:

1. בתור התחלה, מעבירים לפונקציה מספר ובודקים שהיא מחזירה את התוצאה הנכונה:

```
expect(multiply(5)).toBe(25);
```

2. בהמשך, מוודאים שהפלט המתקבל הוא מהסוג שמצפים לו:

```
expect(typeof multiply(1)).toBe('number');
```

3. אבל מה יקרה אם מעבירים לפונקציה קלט שלא מצפים לו, כמו מחרוזת טקסט למשל?

```
multiply('text');
```

כשכתבתם את הפונקציה לא חשבתם על האפשרות הזו, וכרגע בעקבות ההרצה הזאת היא תחזיר **NaN**, מה שעלול לגרום לשגיאות בהמשך. התרחיש הזה, אגב, עלול להתקיים בעיקר בשפות דינמיות כמו ג'אווהסקריפט, שבהן משתנים מסוגלים לשנות את סוגם בזמן ריצה.

כדי לתקן את הבעיה, אפשר לשדרג את הפונקציה ולהוסיף בדיקה של סוג המשתנה:

```
function multiply(number) {
        if (typeof number !== 'number') {
                throw new Error('Parameter is not a number');
    }
    return number * number;
}
```

4. עכשיו אפשר להוסיף בדיקה שמוודאת שאם הפרמטר שעובר הוא לא מספר, מצפים לשגיאה:

```
expect(multiply('test')).toThrowError('Parameter is not a number');
```

בדרך כלל רוצים לעשות בדיקות יחידה על יחידות עצמאיות באפליקציה כמו utilities, helpers, services וכדומה, והבדיקות שמריצים יבדקו שימוש פשוט לצד מקרי קצה – זאת על מנת לכסות את כל האפשרויות ולהגן עליכם מפני טעויות והתנהגויות שלא ציפיתם להן בשימוש בפונקציה. בדיקות יחידה הן דינמיות, וצריך לזכור לשנות אותן אחרי שינויים משמעותיים בקוד.

בדיקות קצה לקצה (End-to-End)

אז יש בדיקות יחידה שיודעות לרוץ על חתיכות קטנות באפליקציה, אבל זה שכל אחד מהחלקים עובד בנפרד לא אומר שהם מסונכרנים ועובדים היטב יחד. נוסף על כך, בדיקות יחידה לא בודקות תרחישים שבהם מעורבות מערכות אחרות – לצורך העניין בדיקות יחידה שרצות על קוד ג'אווהסקריפט בצד הלקוח לא יודעות להגיד אם ממשק המשתמש עובד כמו שצריך עם צד השרת.

קחו לדוגמה מפעל לייצור רכב מסוג פורד. יש מחלקה אחת שאחראית להרכבת המנוע, אחת שאחראית לשלדה ואחת למחשב הרכב. המנוע עובד מצוין, השלדה נראית מעולה, המחשב עובד מצוין. ההרכבה של המנוע לשלדה עוברת בשלום, אבל כשבאים לחבר את המחשב, מגלים שהוא מיועד לרכבים של פורמולה 1.

בדיוק בשביל זה קיימות בדיקות E2E. בעזרת בדיקות E2E אפשר לבדוק שהאפליקציה עובדת מקצה אחד לקצה שני. בדיקות E2E יכללו תרחישים שמשלבים מערכות אחרות, ויוודאו שהבדיקה מצליחה להגיע ממצב א' למצב ב' ללא הפרעות.

לשם ההדגמה אקח אפליקציית חיפוש שכולם משתמשים בה מדי יום: גוגל.

הינה התרחיש שרוצים שהבדיקה תבדוק:

1. לך לאתר google.com.
2. לחץ על אלמנט תיבת חיפוש עם ID בשם "search".
3. הקלד את המילים "אאוטבריין" 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 "אאוטבריין".
7. בדוק שתיבת החיפוש בעלת ID מסוג "search" תכיל את הערך "אאוטבריין".

שימו לב: הבדיקה נעשית בסביבת דפדפן ומדמה פעילות של משתמש אמיתי באתר אמיתי. הרעיון הוא לבדוק אם האפליקציה עובדת ללא התחשבות במבנה שלה או בטכנולוגיה שעומדת מאחוריה, סוג של "קופסה שחורה".

בדיקות ממשקי משתמש (UI Tests)

בדיקות קצה לקצה הן מעולות ויודעות לתת מענה לתרחישים שכוללים מערכות אחרות, אבל לרוב הן בודקות "Happy flows" – מקרים שבהם הכול עובד כמו שצריך. אבל מה לגבי מקרי קצה או מקרים שבהם המערכות שהבדיקות תלויות בהן לא עובדות או לא מחזירות את מה שממשק המשתמש מצפה לקבל?

הבה ניקח כדוגמה את התרחיש שבודק חיפוש בגוגל. התרחיש יוצא מתוך נקודת הנחה שהשרתים שאחראים על החזרת תוצאות החיפוש עובדים ומחזירים תשובה מסוימת. אבל מה יקרה אם תהיה בעיה בשרתים 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 לכתוב סדרה של בדיקות קצה לקצה שעוברות על תרחישים מהסוג הזה, אבל כאמור בדיקות קצה לקצה הן יחסית כבדות, והמטרה שלהן היא להריץ בדיקות שיוצאות מתוך נקודת הנחה שהכול עובד.

מפתחי אאוטבריין מספרים: נתקלנו בבעיה הזאת ורצינו שתהיה לנו דרך להריץ בדיקות על ממשק המשתמש ועל איך הוא מגיב למקרי קצה ולתרחישים מורכבים, אז הוספנו עוד שכבה של בדיקות שבאה לכסות מקרים מהסוג הזה. אנחנו קוראים לבדיקות האלה "בדיקות ממשקי משתמש" (UI Tests באנגלית).
כדי להריץ את בדיקות ה-UI צריכים סביבה סגורה שלא תלויה בשרתים (או לפחות לא בשרתים אמיתיים), אז בנינו כלי בשם "לאונרדו" שיודע לזהות בקשות לשרת, להשתלט עליהן ולדמות תשובה שאנחנו מגדירים מראש.

כך למשל אפשר לראות איך האפליקציה מתמודדת עם תשובות ריקות, עם שגיאות שמגיעות מהשרת ועם מצבים שבהם השרת לא מגיב. כמו כן אפשר לבדוק מצבים שבהם התגובה מהשרת איטית.

בדיקות מהסוג הזה הן מהירות, יציבות ויעילות משום שיש להן את היכולת לבדוק כל תרחיש שרוצים ללא תלות במערכות אחרות.

הכלי לאונרדו הוא כלי open source וזמין לכולם בגיטהאב תחת הכתובת:
https://github.com/outbrain/Leonardo

ספריות ופריימוורקים מומלצים

יש לא מעט ספריות ופריימוורקים לכתיבת בדיקות בג'אווהסקריפט, הינה כמה שאנחנו, באאוטבריין, משתמשים בהם וממליצים לכם בחום לנסות:

- Jest – ספריית הבדיקות של פייסבוק. נותנת מענה לרוב סוגי הבדיקות ובעלת API אינטואיטיבי לכתיבת הבדיקות ולהתמודדות עם שגיאות. עובדת מעולה עם ספריות מודרניות כמו React, Angular, ו-Vue.

- Jasmin – ספריית בדיקות ותיקה ומאוד פופולרית. טובה מאוד בבדיקות יחידה ועומדת בפני עצמה ללא תלות במערכת כלשהי.
- Karma – כלי פשוט שיאפשר לכם להריץ קוד ג'אווהסקריפט ובמקביל לבצע בדיקות בדפדפנים.

סיכום

למדתם מה אומר המושג "בדיקות" בכתיבת קוד, ואיך הוא יעזור לכם לשמור על יציבות ועל קוד נקי. ראיתם דוגמאות לבדיקות יחידה ובדיקות קצה לקצה והבנתם למה הן כל כך חשובות. וגם המלצנו לכם על ספריות ופריימוורקים שיעזרו לכם לכתוב בדיקות.

עוד משהו קטן ששווה לציין: בקבוצות פיתוח מסוימות לקחו את עניין הבדיקות רחוק יותר ואימצו מתודולוגיה בשם "פיתוח מונחה-בדיקות" או באנגלית **Test-Driven Development** (בקיצור TDD). לפי המתודולוגיה הזו, בדיקת יחידה תיכתב עוד לפני כתיבת הקוד שאותו היא בודקת. קצת על הקצה, אבל אם תחליטו ללכת עם המתודולוגיה הזאת, אתם יכולים להיות בטוחים שהקוד שתכתבו יעבוד בצורה חלקה.

נספח: Corvid by Wix

מחברת: מור גלעד, חברת Wix

הקדמה

Corvid היא פלטפורמת פיתוח אפליקציות שנבנתה על העורך הוויזואלי של Wix.

העורך הוויזואלי של Wix נותן למפתח את היכולת לייצר ממשק ויזואלי מפותח מאוד מבלי להשתמש ב-HTML או ב-CSS. הוא מכיל מאות רכיבים חזותיים (elements) שאפשר לעצב אותם באינסוף אפשרויות. בעזרתו אפשר ליצור אתרים מרהיבים.

הפלטפורמה של Corvid מאפשרת:

- לכתוב קוד בג'אווהסקריפט עבור הפונקציונליות של האתר
- לשנות פעולות והתנהגות של רכיבים באתר
- לייצר מבני נתונים

מה בונים?

בפרק הנוכחי אלמד אתכם בצעדים קטנים איך לבנות אפליקציית ניהול משימות בעזרת Corvid.

אפשר לראות את האפליקציה שנבנה בכתובת corvidtodomvc.com.

- הקוד בספר נכתב במטרה ללמד ולהמחיש בצורה ברורה ופשוטה ולאו דווקא לגרום למערכת לעבוד בצורה אידיאלית ויעילה.

תיאור כללי של אפליקציית המשימות:

1. תיבת טקסט (text input) וכפתור ההוספה המאפשרים למשתמש להוסיף משימה חדשה.
2. רשימת המשימות – עבור כל משימה רואים את הסטטוס שלה, אם היא הושלמה או לא.
3. מספר הרשימות שנותרו לבצע.
4. רכיב עם כפתורי בחירה (Radio group) המשמשים לסינון המשימות.

אפשר לראות את כל המשימות (All Tasks), את המשימות שהושלמו (Completed) או את המשימות שעדיין לא הושלמו (Active).

5. כפתור למחיקת כל המשימות אשר הושלמו. בלחיצה על הכפתור תופיע למשתמש חלונית שתשאל אותו אם הוא מעוניין לבצע את הפעולה. אם המשתמש יסכים, כל המשימות אשר הושלמו – יימחקו.

- במהלך הפרק תשתמשו בהרבה פונקציות ש-Corvid מספקת עבור מתכנתים ואסביר את כולן. אם ברצונכם להרחיב את הידע שלכם וללמוד פונקציות נוספות שלא מתוארות בפרק הזה, אתם מוזמנים לבקר באתר המידע של Corvid reference:

https://www.wix.com/corvid/reference/APIsOverview.html

עבור כל פונקציה שאלמד פה, אוסיף הפניה לאזור שבו היא מוסברת באתר המידע.

איך מתחילים?

הכנתי עבורכם תבנית מעוצבת מראש שממנה תוכלו להתחיל =>

https://www.wix.com/website-template/view/html/2186

העורך הוויזואלי של Wix עשיר ביכולותיו ויש הרבה מאוד מה ללמוד עליו. אבל מכיוון שאתם לומדים כרגע ג'אווהסקריפט, רציתי לחסוך לכם את זמן העיצוב והבנייה של הרכיבים באפליקציה על ידי שימוש בתבנית ולהשאיר לכם רק את החלק המהנה – כתיבת הקוד. מובן שאתם יכולים לשנות את העיצוב ולהוסיף רכיבים חדשים.

לאחר שנכנסתם לתבנית, לחצו על

, הירשמו ל-Wix באמצעות כתובת האימייל שלכם, וכעת אתם נמצאים בעורך הוויזואלי של Wix – ברוכים הבאים ☺

כדי להפעיל את Corvid, עברו בעזרת העכבר מעל Dev Mode, בתפריט שנמצא בחלק העליון של האתר ואז על

.

אחרי שהפעלתם את Corvid, תראו את המסך הבא:

אז מה בעצם רואים פה?

1. עורך ויזואלי**.** האזור שבו עורכים את הצד הוויזואלי של האתר. אל אזור זה מוסיפים רכיבים ומשנים את העיצוב שלהם. אפרט לגבי הרכיבים בהמשך.
2. הכפתור +, להוספת רכיבים חדשים לאתר**.** תפריט הרכיבים מכיל מאגר של מאות רכיבים שאפשר להוסיף לאתר על ידי גרירה.
3. מבנה האתר | **site structure**. פאנל שמציג את כל העמודים, חלוניות (**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 כותבים את הקוד. אפשר לכתוב קוד עבור כל עמוד באפליקציה. עבור כל עמוד חדש באפליקציה, מתחילים עם תבנית קוד מוכנה המכילה **$w.onReady()** – שאדבר עליה בהמשך.

* יכול להיות שסביבת הפיתוח שלכם תהיה מצומצמת בשורה התחתונה. לחצו על ↗ כדי להרחיב אותה.

6. כפתור הצג | **preview**. מציג את האתר שעבדתם עליו עד עכשיו. לרוב, משתמשים בכפתור הזה תוך כדי עבודה על האפליקציה לפני שרוצים לפרסם אותה לעולם.

במהלך הפרק, אעשה עצירות מדי פעם כדי לראות מה עשיתם עד עכשיו. בעצירות האלו אבקש שתלחצו על preview.

7. כפתור פרסם | **publish**. מפרסם את האתר שבניתם עד עכשיו וחושף אותו לכל העולם.

התבנית שעליה אתם בונים את האפליקציה מגיעה מוכנה עם מסד נתונים, שבו טבלת Tasks שמכילה משימות.
תלמדו איך להוסיף לטבלה הזו משימות חדשות, לשנות את סטטוס המשימה, למחוק משימות ועוד כל מיני פעולות שעוזרות לנהל 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 – תיאור המשימה
- Completed – משתנה בוליאני המתאר אם המשימה הושלמה או לא

אם אתם רוצים ללמוד איך לייצר את הטבלה בעצמכם, קפצו לסוף הפרק, לנספח שבו הוספתי לכם פירוט והרחבה על בניית טבלאות.

לפני שמתחילים לכתוב קוד, לוחצים על preview ומתבוננים בנקודת הפתיחה:

כפי שאפשר לראות, כל הרכיבים מוכנים ומעוצבים מראש אבל אף כפתור לא מגיב ללחיצה ומספר המשימות שנותרו אינו נכון.

עכשיו לוחצים על Back to Editor כדי להמשיך לעבוד על האפליקציה.

הצגת נתונים ממסד הנתונים באתר

בחלק זה אלמד איך אפשר להוסיף נתונים מהטבלה על ידי "חיבור" רכיבים מהאפליקציה לטבלת ה-Tasks שלנו.
כך חוסכים כתיבת קוד טריוויאלית. מובן שניתן לעשות את זה גם באמצעות קוד – אבל אל דאגה, תכתבו הרבה שורות קוד ממש בקרוב ☺

מתחילים בהכרת הרכיבים שייקחו חלק בהצגת המשימות שבטבלת ה-Tasks של האפליקציה.

1. Repeater – רכיב שיוצר בתוכו כמה עותקים של רכיב אחר. כל עותק מכיל עיצוב זהה אך מידע משתנה. מספר העותקים שייווצרו הוא משתנה ונקבע לפי מספר הפריטים שעליו להציג. במקרה הזה, מספר המשימות שהוא יציג.

   https://www.wix.com/corvid/reference/$w.Repeater.html

2. Checkbox | תיבת סימון – רכיב המאפשר הכנסת ערך בוליאני (true | false).

   https://www.wix.com/corvid/reference/$w.Checkbox.html

3. Text | רכיב טקסטואלי – רכיב שמציג טקסט.

   https://www.wix.com/corvid/reference/$w.Text.html

4. Dataset – הרכיב שמקשר בין המידע בטבלת ה-Tasks לבין מרכיבים אחרים בעמוד.

   https://www.wix.com/corvid/reference/wix-dataset.html

Dataset

ה-dataset מחבר רכיבים בעמוד למידע מטבלאות נתונים.

רכיב זה הוא וירטואלי, כלומר, הוא יעשה את תפקידו בניהול מידע בעמוד אבל מבקרי העמוד לא יראו אותו באפליקציה הסופית.

אפשר להסתכל על ההגדרות של ה-dataset בעזרת לחיצה על כפתור ה-settings:

אז נראה שה-dataset מחובר לטבלת ה-Tasks ונמצא במצב של קריאה וכתיבה (read & write).

הוא צריך להיות מוגדר במצב של קריאה וכתיבה, מכיוון שתרצו אפשרות להציג את המשימות אבל גם לשנות את סטטוס ה-completed שלהן.

* יש עוד שני מצבים ל-dataset:

Read Only – מצב שמאפשר רק קריאה של נתונים מהטבלה

Write Only – מצב שמאפשר רק הוספה של נתונים חדשים לטבלה

נוסף על כך, Number of items to display מהווה את כמות הרשומות המקסימלית שה-dataset ישלוף מהטבלה. אם אתם רוצים לתמוך 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 את זה לוחצים על האייקון של מסד הנתונים, שמופיע כאשר לוחצים עם העכבר 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-connect a dataset.

התוצאה של הפעולה הזו היא שמספר הקופסאות ב-repeater ישוכפלו כמספר הרשומות שה-dataset שולף מטבלת המשימות.

הצעד הבא הוא להציג כל משימה בשני הרכיבים שיש בתוך ה-repeater.

מתחילים ברכיב הטקסט. פותחים את פאנל החיבור של רכיב הטקסט (זה דורש לחיצה ראשונית על ה-repeater ולאחר מכן לחיצה על רכיב הטקסט) ומחברים את הטקסט של הרכיב לשדה Title מטבלת ה-Tasks.

- אולי שמתם לב שה-dataset שיצרתם כבר מחובר. זה קורה באופן אוטומטי מכיוון שחיברתם את ה-repeater ל-dataset הזה.
- ברשימת השדות שנפתחת יש שדות נוספים שאפשר להתחבר אליהם. אלו שדות שנוצרים באופן אוטומטי עבור כל רשומה בטבלה:
  - ID
  - createdDate
  - updatedDate

לאחר מכן, פותחים את פאנל החיבור של ה-checkbox (גם זה ידרוש לחיצה על ה-repeater, לאחר מכן לחיצה על ה-group שמכיל את ה-checkbox, ולבסוף על ה-checkbox) ומחברים את הערך checked לשדה completed מהטבלה.

אחרי שחיברתם את הרכיבים שלכם לטבלה, הגיע הזמן לראות את התוצאות!

לחצו על preview והסתכלו על התוצאות. התוצאות הרצויות הן:

רואים את המשימות? יש!

עכשיו הגיע הזמן להתחיל לכתוב קוד – הגיגה!

הוספת משימה חדשה

הרכיבים שיעזרו לכם להוסיף משימה חדשה:

1. תיבת טקסט | text input – מאפשרת להכניס את תיאור המשימה.
   https://www.wix.com/corvid/reference/$w.TextInput.html
2. כפתור | Button – בלחיצה עליו תתווסף המשימה מתיבת הטקסט לטבלת Tasks.
   https://www.wix.com/corvid/reference/$w.Button.html

אם רוצים לאפשר למשתמשים להוסיף משימה חדשה, יש ללמוד איך בלחיצה על הכפתור מוסיפים משימה חדשה לתוך מסד הנתונים ומציגים אותה ב-repeater.

Events

https://www.wix.com/corvid/reference/$w.Event.html

בדומה לאירועים בג'אווהסקריפט, גם הרכיבים של Corvid 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/corvid/reference/$w.Button.html#onClick
- textInput.onKeyPress – event שנקרא כאשר המשתמש לוחץ על כפתור במקלדת בזמן שהוא נמצא בתוך רכיב ה-text input.
  https://www.wix.com/corvid/reference/$w.TextInput.html#onKeyPress

מתחילים בהוספת event לכפתור . עושים זאת על ידי פתיחת פאנל ה-properties של הכפתור, בקליק ימני על הכפתור ולחיצה על View Properties.

פאנל ה-properties מחולק לשני חלקים:
בחלקו העליון אפשר לראות את ה-ID של הרכיב. בעזרת ה-ID אפשר לבצע פעולות עם הרכיב דרך הקוד. בתבנית שהכנתי לכם, נתתי מראש לכל רכיב ID שם משמעותי, אבל אתם יכולים לשנות את זה.
נוסף על כך, מתחת ל-ID מופיעות הגדרות הנראות של הרכיב בזמן העלייה של העמוד.

בחלקו התחתון של הפאנל נמצאת רשימה של כל ה-events הנתמכים עבור הרכיב:

- onViewportEnter מופעל כאשר הכפתור מופיע בחלון של הדפדפן
- onViewportLeave מופעל כאשר הכפתור נעלם מחלון הדפדפן
- onMouseIn מופעל כאשר המשתמש מעביר את העכבר מעל הכפתור
- onMouseOut מופעל כאשר העכבר יוצא מגבולות הכפתור
- onClick מופעל בלחיצה על הכפתור
- onDblClick מופעל בלחיצה כפולה על הכפתור

* לכל רכיב יש events שונים בהתאם לפונקציונליות שהוא מספק. ניתן לראות הסבר מדויק על כל ה-events של הכפתור בלינק:

https://www.wix.com/corvid/reference/$w.Button.html

במקרה הזה רוצים להוסיף משימה כאשר המשתמש לוחץ על הכפתור. לכן, עוברים עם העכבר על הצד הימני של event ה-onClick ולוחצים על כפתור ה- ⊕. לחיצה על הכפתור תכניס 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:
}
```

עכשיו, בכל לחיצה על הכפתור, ייקרא ה-event שלכם. הבה נראה שזה אכן נקרא בכל לחיצה. מוסיפים כתיבה ל-console:

```
export function addTaskButton_click(event) {
    console.log('button clicked');
}
```

שימו לב שהפונקציה שלכם מקבלת ארגומנט שנקרא event. הארגומנט הזה הוא אובייקט המכיל מידע על ה-event שנקרא. האובייקט הזה משתנה בהתאם לסוג ה-event.

* מעתה תכתבו ותעתיקו הרבה שורות קוד. אם אתם רוצים שהקוד יעבור הזחה באופן אוטומטי, לחצו על הכפתור <‹›>, שנמצא בצד הימני של החלק העליון בסביבת הפיתוח.

עוברים ל-preview, לוחצים על הכפתור ובתחתית הדפדפן ייפתח ה-console של Corvid ויציג את מה שכתבתם בכל פעם שתלחצו.

עובד לכם? מעולה! ממשיכים הלאה ושומרים את המשימה.

$w

https://www.wix.com/corvid/reference//$w.html

על מנת לשמור את המשימה שהמשתמש כתב, יש "להוציא" את הטקסט שנכתב מתוך רכיב ה-text input.
את זה עושים בעזרת שימוש בפונקציה $w.

$w היא פונקציה שזמינה בקוד ומאפשרת לבחור רכיבים מתוך העמוד באמצעות הקוד, בדומה לפונקציה document.querySelector. לכל רכיב סט מאפיינים ופונקציות ייחודי המאפשר אינטראקציה עם הרכיב, לקבל ולשנות את המידע שלו, לשנות את מצבו ולהירשם ל-events שהוא חושף.

איך בוחרים רכיב בעזרת $w?
זוכרים את ה-ID שיש לכל רכיב בפאנל ה-properties? אז על מנת לקבל את המשתנה של הרכיב משתמשים ב-ID שלו. ואז, כדי לבחור אותו, קוראים ל-$w עם ארגומנט מסוג 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, ואז תבחרו אותה כך:

```
$w('#taskInput').value
```

אחרי שיש תיבת טקסט, רוצים לדעת מה הטקסט שהמשתמש הכניס לתוכה. לכל רכיב יש מספר רב של נתונים שהוא יכול לספק. במקרה הזה צריכים את ה-value שהמשתמש יכניס לתיבה:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;
  console.log('taskTitle:', taskTitle);
}
```

למידע נוסף על משתנה ה-value של תיבת הטקסט:
https://www.wix.com/corvid/reference/$w.TextInput.html#value

עוברים ל-preview, כותבים את המשימה בשורת המשימה, לוחצים על הכפתור והידד! רואים את הטקסט שכתבתם.

הצעד הבא => לשמור את המשימה כמשימה חדשה בטבלת ה-Tasks.

wix-data

https://www.wix.com/corvid/reference/wix-data.html

wix-data הוא מודול המאפשר עבודה מול המידע שיש לכם בטבלאות. בעזרתו, אפשר להוסיף רשומות לטבלה, למחוק רשומות וגם לערוך רשומות קיימות.
במהלך הפרק אלמד בכל פעם שימוש ופונקציונליות חדשה של wix-data.
על מנת להשתמש במודול, יש לייבא אותו מהאזור העליון של קובץ הקוד:

```
import wixData from 'wix-data';
```

wixData.insert() – הוספת רשומה לטבלה

https://www.wix.com/corvid/reference/wix-data.html#insert

מתחילים עם היכולת להכניס משימה חדשה לטבלה. זה אפשרי באמצעות הפונקציה האסינכרונית insert()
לפונקציה הזו מעבירים שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה – במקרה הזה Tasks.
2. המשימה שאותה רוצים להוסיף. המשימה תהיה במבנה של אובייקט, שבו המשתנים יהיו זהים למאפיינים של המשימה בטבלה והערכים שלהם.

אז על מנת להוסיף משימה חדשה לטבלה, מייצרים את אובייקט המשימה שיכיל:

- title – טקסט המתאר את המשימה, שנלקח מתיבת הטקסט
- completed – משתנה בוליאני המייצג את סטטוס המשימה ויאותחל ב-false

קוראים ל-wixData.insert() עם האובייקט שיצרתם. נוסף על כך, 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);
    }
}
```

* שימו לב שמכיוון שהפונקציה wixData.insert() היא אסינכרונית, מוסיפים async ל-event.

עכשיו קופצים שוב ל-preview, מוסיפים משימה חדשה, לוחצים על Enter, חוזרים ל-editor של Wix ועוברים לעורך התוכן של טבלת ה-Tasks כדי לראות אם המשימה החדשה התווספה.

התווספה? מעולה!

אבל אני מניח ששמתם לב שחסרים כמה דברים כדי שהחוויה תהיה שלמה:

1. שורת המשימה לא התרוקנה אחרי הוספת המשימה.
2. המשימה לא התווספה לרשימת המשימות שלכם.

ציינתי קודם שבעזרת $w אפשר לקבל וגם 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 החדשים, יש לרענן את המידע שה-dataset מעביר אליו. זה לא קורה באופן אוטומטי.
כמו שכל רכיב רגיל באתר ניתן לשינוי, כך גם ה-dataset. לכן מבקשים מה-dataset לקרוא מחדש את המידע מהטבלה בעזרת קריאה לפונקציה האסינכרונית refresh().

```
await $w('#dataset1').refresh();
```

למידע נוסף על פונקציית ה-refresh() של dataset:
https://www.wix.com/corvid/reference/wix-dataset.Dataset.html#refresh

זה הקוד המלא:

```
export async function addTaskButton_click(event) {
  const taskTitle = $w('#taskInput').value;

  if (taskTitle.length !== 0) {
    const newTask = {
      title: taskTitle,
      completed: false
    }
    await wixData.insert('Tasks', newTask);
    $w('#taskInput').value = '';
    await $w('#tasksDataset').refresh();
  }
}
```

ושוב, עוברים ל-preview ומכניסים משימה חדשה. איזה כיף! זה עובד!

תרגיל:

בצעו שמירה של משימה חדשה כאשר המשתמש כותב משימה חדשה בתיבת הטקסט ואז מקליד על Enter.

רמז: הזכרתי קודם את ה-event על תיבת הטקסט – textInput.onKeyPress ואת המשמעות של הארגומנט event שמקבלים בפונקציה.

תשובה:

לאחר פתיחה של פאנל ה-properties ואז יצירה של event ה-onKeyPress יתווסף event נוסף לסביבת הפיתוח. אבל הפעם יהיה צורך לוודא שהכפתור שעליו המשתמש לחץ הוא כפתור ה-Enter, ורק אז לשמור 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, מוציאים את פעולת 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 הפך לאסינכרוני ונוסף לו async.

עוברים שוב ל-preview ומכניסים משימות חדשות באמצעות לחיצה על Enter וגם בעזרת לחיצה על כפתור ההוספה.

שינוי סטטוס המשימה

אז אפשר לראות את המשימות הקיימות ולהוסיף משימות חדשות. אבל המשתמש עדיין לא יכול לעדכן שהמשימה הושלמה.

בכל פעם שהמשתמש משנה את הסימון בכפתור הבחירה (checkbox), שנמצא בתוך ה-repeater, נרצה לעדכן את הבחירה החדשה שלו בטבלה עבור המשימה הרלוונטית. מתחילים בהוספת event של onChange על ה-checkbox דרך פאנל ה-properties, כמו שלמדתם קודם:

```
export function completedCheckbox_change(event) {
   //Add your code for this event here:
}
```

למידע נוסף על event ה-onChange של כפתור הבחירה:

https://www.wix.com/corvid/reference/$w.Checkbox.html#onChange

* מכיוון שה-checkbox נמצא בתוך ה-repeater, ה-event שנממש על checkbox אחד יופעל אוטומטית על כל אחד ממה-checkboxes שיש בכל איבר ב-repeater. כלומר אין צורך לכתוב event לכל checkbox אם יש לנו repeater.

כדי לשנות את הערך הבוליאני של מאפיין ה-completed של המשימה לערך החדש שהמשתמש סימן ב-checkbox, קודם כול צריך להשיג את הערך החדש, ועושים זאת על ידי שימוש במשתנה ה-target, שנמצא על ארגומנט ה-event שמועבר לפונקציה:

```
export function completedCheckbox_change(event) {
   const completed = event.target.checked;
}
```

למידע נוסף על משתנה ה-target של event:

https://www.wix.com/corvid/reference/$w.Event.html#target

לאחר מכן נרצה לעדכן את הערך החדש בטבלה.

wixData.get() – שליפת רשומה מהטבלה

https://www.wix.com/corvid/reference/wix-data.html#get

על מנת לייצר את המשימה שאותה רוצים לערוך, יש לשלוף את הרשומה הזו מהטבלה כדי שכל הנתונים הנוכחיים שלה יהיו זמינים.

למודול wixData יש פונקציה אסינכרונית הנקראת get(), שמאפשרת שליפה של משימה מהטבלה.
לפונקציה הזו מעבירים שני ארגומנטים:

1. שם הטבלה שממנה רוצים לשלוף 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.

אז איך בעצם משיגים את המזהה הייחודי הזה מתוך המשימה ב-repeater שעליו לחץ המשתמש?
בעזרת ארגומנט ה-event. כפי שכתבתי קודם, ארגומנט ה-event משתנה בהתאם ל-event שנקרא. אבל יש חשיבות למיקום הרכיב שבו מתרחש ה-event ובמיוחד אם הוא בתוך ה-repeater. אם ה-event הופעל והרכיב שהוא הופעל ממנו נמצא בתוך ה-repeater אנו נקבל באובייקט ה-event משתנה נוסף, שאין ב-event רגיל, שנקרא context ומכיל את ה-ID של הרשומה שצריכים מהטבלה:

```
event.context.itemId
```

למידע נוסף על משתנה ה-context של event:
https://www.wix.com/corvid/reference/$w.Event.html#context

וכעת, כשיש בידיכם המזהה הייחודי של הרשומה, אפשר לקרוא ל-wixData.get() ולקבל את הרשומה הנוכחית:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;

  const item = await wixData.get('Tasks', itemId);
}
```

* שוב, הפכתם את ה-event שלכם לאסינכרוני כיוון ש-wixData.get() היא אסינכרונית.

wixData.update() – עדכון רשומה בטבלה

https://www.wix.com/corvid/reference/wix-data.html#update

יש לכם רשומה, כפי שהיא שמורה בטבלה. עכשיו, יש לשנות את ערך ה-completed שלה לערך החדש ולעדכן אותו בטבלה.

למודול wixData יש פונקציה update() שמאפשרת עדכון של רשומה מסוימת. הפונקציה הזו מקבלת שני ארגומנטים:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה – Tasks.
2. המשימה המעודכנת שרוצים לשמור.

מייצרים משימה מעודכנת ואז קוראים ל-wixData.update() עם העדכון:

```
export async function completedCheckbox_change(event) {
  const completed = event.target.checked;
  const itemId = event.context.itemId;

  const item = await wixData.get('Tasks', itemId);
  const updatedItem = Object.assign({}, item, { completed: completed });

  await wixData.update('Tasks', updatedItem);
}
```

הבה נבדוק אם זה עובד ב-preview. מסמנים כמה משימות וחוזרים לעורך התוכן לבדוק אם זה באמת עודכן.

נהדר! זה עודכן! הפונקציונליות הבסיסית של האפליקציה עובדת.

מספר המשימות שלא הושלמו

היכולת הבאה שרוצים להוסיף לאפליקציה היא תצוגה של מספר המשימות שלא הושלמו (שהערך הבוליאני שלהן במאפיין ה-completed הוא false) ברכיב הטקסט שמופיע מעל רשימת המשימות.

מתחילים בבניית פונקציה אסינכרונית חדשה שתפקידה יהיה לעדכן את רכיב הטקסט במידע הרצוי.
שמה: updateActiveTaskCount

```
async function updateActiveTaskCount() {

}
```

הצעד הבא הוא לספור את מספר המשימות שמאפיין ה-completed שלהן הוא false.
את זה עושים בעזרת פונקציונליות נוספת ש-wixData מספק, ונקראת wixDataQuery.

wixDataQuery

https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html

wixDataQuery היא שאילתה המאפשרת לבצע בקשה למידע מטבלת המשימות. הבקשה יכולה להכיל שילוב של פקודות שיגדירו את התוצאות שלהן מצפים.

יצירת שאילתה נעשית בשלושה שלבים:

1. אתחול השאילתה בהינתן שם הטבלה שממנה רוצים לקבל את המידע.
2. בניית הפקודות שמגדירות את השאילתה.
3. הפעלת השאילתה.

1. wixData.query – אתחול השאילתה

https://www.wix.com/corvid/reference/wix-data.html#query

את השאילתה מאתחלים בעזרת הפונקציה wixData.query(), שמקבלת ארגומנט אחד:

1. שם הטבלה שאליה רוצים להוסיף את הרשומה החדשה.

מתחילים לבנות את השאילתה בעזרת:

```
wixData.query('Tasks')
```

2. בניית השאילתה

לאחר מכן בונים את הפקודות של השאילתה באמצעות מגוון פונקציות.

במקרה הזה יש לקבל רק את המשימות שלא הושלמו. כלומר, כל המשימות שערך ה-completed שלהן הוא false.

לכן, משתמשים בפונקציה eq (מהמילה שווה – equal), שמקבלת שני ארגומנטים:

1. שם המאפיין שבאמצעותו רוצים לסנן את התוצאות – במקרה הזה completed.
2. הערך שרוצים שיהיה לתוצאות – במקרה הזה false.

למידע נוסף על פונקציית ה-eq:

https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#eq

```
wixData.query('Tasks')
.eq('completed', false)
```

* יש פונקציות רבות ומגוונות שמאפשרות בנייה של שאילתה ואפשר לקרוא על ולן בלינק:

https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html

3. הפעלת השאילתה

לביצוע השאילתה אפשר להשתמש בשתי פונקציות אסינכרוניות שונות:

1. find() – החזרת כל האיברים שעונים על השאילתה.

https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#find

2. count() – החזרת מספר האיברים שעונים על השאילתה.

https://www.wix.com/corvid/reference/wix-data.WixDataQuery.html#count

מכיוון שצריך לדעת רק את מספר המשימות שלא הושלמו, משתמשים בפונקציה count():

```
const activeTaskCount = await wixData.query('Tasks').eq('completed', false).count();
```

עכשיו, כשהושג מספר המשימות שלא הושלמו, כותבים אותן ברכיב הטקסט.

מתחילים בייצור הטקסט שרוצים לראות:

```
const activeTaskText = activeTaskCount === 1 ?
   '1 item left' :
   `${activeTaskCount} items left`;
```

עכשיו, מציבים את הטקסט החדש בערך ה-text של רכיב הטקסט שה-ID שלו הוא activeTaskCount.

למידע נוסף על משתנה ה-text השייך לרכיב הטקסט:

https://www.wix.com/corvid/reference/$w.Text.html#text

זוכרים איך עושים את זה?

```
$w('#activeTasksCount').text = activeTaskText;
```

הינה כל הפונקציה:

```
async function updateActiveTaskCount() {
  const activeTaskCount = await wixData.query('Tasks')
    .eq('completed', false)
    .count();

  const activeTaskText = activeTaskCount === 1 ?
    '1 item left' :
    `${activeTaskCount} items left`;

  $w('#activeTasksCount').text = activeTaskText;
}
```

יצרתם פונקציה שמעדכנת את מספר המשימות שלא הושלמו, אבל אף אחד לא קורא לפונקציה.

מתי בעצם צריכים לקרוא לפונקציה?

1. לאחר הוספה של משימה חדשה – בסוף פונקציית addNewTask

```
async function addNewTask() {
  const taskTitle = $w('#taskInput').value;

  if (taskTitle.length !== 0) {
    // task insert code

    await updateActiveTaskCount();
  }
```

```
}
```

2. לאחר שינוי סטטוס של משימה קיימת – בסוף ה-completedCheckbox_change event

```
export async function completedCheckbox_change(event) {
  // completed update code

  await updateActiveTaskCount();
}
```

3. כאשר העמוד סיים להיטען בעזרת $w.onReady

$w.onReady()

https://www.wix.com/corvid/reference/$w.html#onReady

זוהי פונקציה שרצה כאשר כל הרכיבים של העמוד סיימו להיטען. בפונקציה הזו אפשר לכתוב קוד שירוץ לפני שהמשתמש יתחיל לבצע שינויים באפליקציה.

לכן, קוראים ל-updateActiveTaskCount בתוך הפונקציה onReady כך:

```
$w.onReady(function () {
  updateActiveTaskCount();
});
```

עוברים ל-preview.

- מוודאים שהמספר הראשוני שמופיע נכון
- מוסיפים משימה חדשה ומוודאים שמספר המשימות שלא בוצעו עולה ב-1
- מסמנים משימה שלא בוצעה ומוודאים שמספר המשימות שלא בוצעו יורד ב-1
- מורידים סימון למשימה שבוצעה ומוודאים שמספר המשימות שלא בוצעו עולה ב-1

עובד? שיגעון! הלאה!

סינון המשימות לפי סטטוס המשימה

עכשיו מסננים את המשימות שהמשתמש רואה לפי הסינון שהוא יבחר בכפתורי הרדיו (radio button group) שנמצאים מעל רשימת המשימות:

למידע נוסף על רכיב כפתורי הרדיו:

https://www.wix.com/corvid/reference/$w.RadioButtonGroup.html

להלן ההתנהגות הרצויה עבור כל אחד מהכפתורים:

- All Tasks – כל המשימות יופיעו ללא סינון
- Completed – רק המשימות שהושלמו (שהמשתנה completed שלהם שווה ל-true) יופיעו
- Active – המשימות שעדיין לא הושלמו (שהמשתנה completed שלהם שווה ל-false) יופיעו

מכיוון שהסינון יתבצע לאחר שהמשתמש ילחץ על אחד הכפתורים, ה-event המתאים הוא radioButtonGroup.onChange – מוסיפים אותו כמו שלמדתם קודם:

```
export function filterRadioGroup_change(event) {
  //Add your code for this event here:
}
```

למידע נוסף על ה-onChange event של כפתורי הרדיו:

https://www.wix.com/corvid/reference/$w.RadioButtonGroup.html#onChange

בשלב הבא מבינים מהו סוג הסינון שבחר המשתמש וממשים אותו.

עבור כל אחת מהאפשרויות ב-radio group מזינים ערך הזהה 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 כפתור רדיו, וכשמו כן הוא – הנתון (או הערך) שאותו אנו רוצים לקבל מהמשתמש.

```
$w('#filterRadioGroup').value
```

למידע נוסף על משתנה ה-value של כפתורי הרדיו:

https://www.wix.com/corvid/reference/$w.RadioButtonGroup.html#value

עכשיו מבצעים סינון של המידע שה-dataset מביא אל העמוד בהתאם לבחירה של המשתמש.

סינון מידע מה-dataset נעשה בשלושה שלבים:

1. אתחול המסנן wixData.filter().
2. בניית הפקודות שמגדירות את הסינון.
3. הפעלת הסינון על ידי העברה של אובייקט הסינון שיצרתם ל-dataset.

1. wixData.filter()

https://www.wix.com/corvid/reference/wix-data.html#filter

מייצר אובייקט סינון, שמועבר ל-dataset, ובכך מאפשר לסנן את המידע שה-dataset מביא לעמוד:

```
let filter = wixData.filter()
```

2. בניית הסינון

לאחר מכן בונים את הפקודות של הפילטר על ידי שימוש באותן פונקציות שמגדירות שאילתה.

במקרה הזה צריכים ליצור שני סוגים שונים של פילטר:

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-completed שלהן שווה ל-true. לצורך זה משתמשים ב-eq (בדומה לשאילתה 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 ה-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();
```

3. dataset.setFIlter() – הפעלת הסינון על ה-dataset

https://www.wix.com/corvid/reference/wix-dataset.Dataset.html#setFilter

dataset.setFilter() היא פונקציה אסינכרונית שמפעילה את הסינון על רכיב ה-dataset. היא מקבלת ארגומנט אחד – אובייקט הפילטר שיצרתם עד עכשיו.
לאחר שקוראים לפונקציה הזו, ה-dataset יבצע קריאה מחודשת למשימות מהטבלה, אבל ללא משימות שאינן עונות על קריטריון הסינון.

```
await $w('#tasksDataset').setFilter(filter);
```

עכשיו מחברים הכול יחד ל-event של ה-onChange שיצרתם קודם:

```
export async function filterRadioGroup_change(event) {
  const filterValue = $w('#filterRadioGroup').value;

  let filter;

  switch (filterValue) {
  case 'Completed':
```

```
        filter = wixData.filter()
.eq('completed', true);
        break;
    case 'Active':
        filter = wixData.filter()
.eq('completed', false);
        break;
    case 'All Tasks':
        filter = wixData.filter();
        break;
    }

    await $w('#tasksDataset').setFilter(filter);
}
```

לא מספיק לקרוא לפונקציית הסינון רק כאשר המשתמש לוחץ על אחד מכפתורי הסינון. יש לסנן את המשימות בעוד כמה מצבים בקוד. לכן מתחילים להעביר את כל קוד הסינון לפונקציה ייעודית ששמה filterTasks:

```
async function filterTasks() {
    const filterValue = $w('#filterRadioGroup').value;
    // create filter code

    await $w('#tasksDataset').setFilter(filter);

}
```

עכשיו, קוראים לה מכל המקומות השונים שדורשים אף הם סינון:

1. כאשר המשתמש לוחץ על אחד הכפתורים מה-radio group buttons (כמו שעשיתם):

```
export async function filterRadioGroup_change(event) {
    await filterTasks();
}
```

2. כאשר המשתמש משנה את סטטוס ה-completed של אחת המשימות:

```
export async function completedCheckbox_change(event) {
   // change completed status
   // update uncompleted counter
   await filterTasks();
}
```

3. כאשר המשתמש מוסיף משימה חדשה:

```
async function addNewTask() {
   const taskTitle = $w('#taskInput').value;

   if (taskTitle.length !== 0) {
      // add new task

      await updateActiveTaskCount();
 await filterTasks();
   }
}
```

עוברים ל-preview, "משחקים" עם הסינון שזה עתה נכתב, משנים סטטוס משימות כאשר נמצאים על סינון מסוים ומוודאים שהן נשארות או נעלמות ולבסוף, מוסיפים משימה חדשה כאשר נמצאים במצב סינון של משימות שלא הושלמו ומוודאים שהמשימה לא מופיעה ברשימה.

שמחה וששון! יש סינון! ☺

ניקוי המשימות שהושלמו

עכשיו מממשים את כפתור ה-Clear completed, שבלחיצה יפתח את חלון האישור:

- אם המשתמש ילחץ Yes => סוגרים את חלון האישור ומוחקים את כל המשימות שהושלמו
- אם המשתמש ילחץ No => סוגרים את חלון האישור ולא עושים דבר

את חלון האישור כבר הכנתי עבורכם וקראתי לו Clear Confirmation. תמצאו אותו ב-Site Structure מתחת ל-Lightboxes.

מוסיפים event של לחיצה על כפתור ה- Clear completed כמו שלמדתם (דרך פאנל ה-properties):

```
export function clearCompletedButton_click(event) {
  //Add your code for this event here:
}
```

ועכשיו אלמד איך פותחים את חלון האישור.

wix-window

https://www.wix.com/corvid/reference/wix-window.html

wix-window הוא מודול שמאפשר עבודה מול חלון הדפדפן שעליו עובדים.

על מנת להשתמש במודול, יש לייבא אותו מהאזור העליון של קובץ הקוד (בדומה ליבוא של wix-data):

```
import wixWindow from 'wix-window';
```

wixWindow.openLightBox()

https://www.wix.com/corvid/reference/wix-window.html#openLightbox

המודול wixWindow תומך בפתיחה של חלון חדש על ידי קריאה לפונקציה האסינכרונית openLightBox, שמקבלת שני ארגומנטים:

1. שם החלון שאותו רוצים לפתוח. במקרה הזה – "Clear Confirmation".
2. האובייקט שרוצים להעביר לחלון. במקרה הזה אין צורך בכך.

ערך ההחזרה של ה-promise יהיה האובייקט שיוחזר על ידי החלון (במידה שקיים).

לכן, על מנת לפתוח 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 המשתמש לחץ על Yes או על No, כדי לדעת איך לפעול. כלומר, רוצים שהתשובה שתחזור מה-promise כשהחלון ייסגר תהיה ערך בוליאני, שאם הוא חיובי – יימחקו המשימות שהושלמו ואם הוא שלילי – לא יקרה דבר.

עוברים לחלון ה-Clear Confirmation כדי להחזיר תשובה ל-event.

גם פה, כמו בעמוד ה-HOME שלכם, אפשר לכתוב קוד בעורך הקוד שבתחתית המסך.

ועכשיו, מוסיפים event לחיצה על כפתור ה-Yes:

```
export function approveBtn_click(event) {
   //Add your code for this event here:
}
```

ומוסיפים event לחיצה על כפתור ה-No:

```
export function rejectBtn_click(event) {
  //Add your code for this event here:
}
```

כעת רוצים לסגור את החלון בלחיצה על כל אחד מהכפתורים. מתחילים ביבוא המודול wix-window לחלק העליון של הקוד:

```
import wixWindow from 'wix-window';
```

wixWindow.lightbox.close()

https://www.wix.com/corvid/reference/wix-window.lightbox.html#close

wixWindow.lightbox.close() היא פונקציה שסוגרת את החלון שבו נמצאים ומחזירה את המשתמש לעמוד שפתח את החלון. הפונקציה הזו מקבלת ארגומנט אחד: ערך שיינתן כתשובה של ה-promise שחזר מ-wixWindow.openLightbox().

לכן, בלחיצה על Yes מחזירים true ובלחיצה על No מחזירים false:

```
export function approveBtn_click(event) {
  wixWindow.lightbox.close(true);
}

export function rejectBtn_click(event) {
  wixWindow.lightbox.close(false);
}
```

אש! חוזרים לעמוד ה-HOME כדי לבצע את המחיקה במידה שהמשתמש ילחץ על Yes.

במידה שהמשתמש בחר למחוק את המשימות שהושלמו, מבצעים את המחיקה בשני שלבים:

1. שאילתה שתחזיר את כל המשימות שערך ה-completed שלהן הוא true.
2. מחיקה של כל המשימות שהתקבלו מהשאילתה.

למדתם מוקדם יותר איך לבצע שאילתה דומה, אבל בשאילתה הקודמת רציתם לקבל את מספר המשימות ולכן השתמשתם 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/corvid/reference/wix-data.WixDataQuery.html#find

כשקוראים ל-find(), חוזר אובייקט שמכיל כל מיני דברים. אחד מהם הוא ה-items שמחזיק מערך של כל הערכים שחזרו – במקרה הזה, מערך של כל המשימות שחזרו מהשאילתה.

כעת עוברים על כל אחת מהמשימות הללו ומוחקים אותן באמצעות:

wixData.remove() – מחיקת רשומה מהטבלה

https://www.wix.com/corvid/reference/wix-data.html#remove

מחיקת רשומה מהטבלה נעשית על ידי שימוש בפונקציה האסינכרונית remove(), שמקבלת שני ארגומנטים:

1. שם הטבלה שממנה רוצים למחוק את הרשומה – במקרה הזה Tasks.
2. ה-id של הרשומה שאותה רוצים למחוק.

זוכרים את מזהה ה-_id שנכנס באופן אוטומטי לכל משימה בטבלה? מעולה!
אז עכשיו תשתמשו בו.

מכיוון ש-remove() היא אסינכרונית, וצריך לעבור על מערך של משימות שחזרו מהשאילתה, משתמשים ב-Promise.all() כדי לבצע את המחיקה:

```
await Promise.all(completedTasks.map(task => wixData.remove('Tasks', task._id)));
```

ולקינוח, קוראים שוב לפונקציה filterTasks(), כדי להסיר מהתצוגה את המשימות שנמחקו.

ולקוד המלא:

```
export async function clearCompletedButton_click(event) {
   const shouldClearCompleted = await wixWindow.openLightbox('Clear
Confirmation');
```

```
    if (shouldClearCompleted) {
        const queryResult = await wixData.query('Tasks').eq('completed', true).find();
        const completedTasks = queryResult.items;

        await Promise.all(completedTasks.map(task => wixData.remove('Tasks',
task._id)));
        await filterTasks();
    }
}
```

ועכשיו 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 שכל המשימות שהושלמו יימחקו.

ממש כמעט סיימתם!

זוכרים את כפתור ה-publish מתחילת הפרק?

עכשיו הזמן ללחוץ עליו!

אחרי לחיצה על כפתור ה-publish תהיה האפליקציה שלכם זמינה כאתר לכל דבר ולכל העולם!

הכתובת של האפליקציה שלכם תוצג בחלון שייפתח לאחר שתלחצו על publish:

* אחרי שפרסמתם את האפליקציה, המשימות שהוספתם בזמן שעבדתם על האפליקציה לא יופיעו ב-preview, מכיוון שאתם עובדים על מסדי נתונים שונים עבור פיתוח ועבור האפליקציה האמיתית.

נספח – יצירת מסד נתונים

מסד נתונים הוא אמצעי המשמש לאחסון מסודר של נתונים. במקרה של Corvid, הנתונים הללו מאוחסנים במודל של טבלאות (collections) ובכל טבלה העמודות מייצגות שדות (fields) והשורות מייצגות רשומות שונות.

נהוג לשייך כל טבלה לישות מסוימת באפליקציה, ואז כל שורה מייצגת מקרה ספציפי של הישות.
נוסף על כך, כל עמודה בטבלה היא בעצם שדה שמייצג מאפיין מסוים של הישות. לכל שדה יש סוג כגון טקסט, מספר, בוליאני, תמונה וכדומה.

הבה נתכנן את מסד הנתונים של האפליקציה.

באפליקציה שלכם יש ישות שנקראת משימה (task), ולכן יש ליצור טבלה שנקראת Tasks. כל שורה בטבלה תייצג משימה בודדת.
על מנת ליצור טבלה חדשה עוברים עם העכבר על כפתור +, שיופיע במעבר על צידו הימני של ה-database שנמצא בפאנל ה-site structure.

לאחר מכן לוחצים על ה-New Collection:

Create Collection

הלחיצה תקפיץ מסך שבו נותנים לטבלה את השם tasks ולאחר לחיצה על תיווצר טבלה חדשה.

יצירת הטבלה הוסיפה טבלה חדשה ב-site structure מתחת ל-database, ובנוסף העבירה אתכם ישר לאזור שנקרא "עורך התוכן" – content manager, שבו אפשר להוסיף שדות ולערוך את התוכן של הטבלה.

הגיע הזמן לייצר את השדות. כל משימה מכילה את תיאור המשימה בטקסט וערך בוליאני שמגדיר את סטטוס המשימה – אם היא הושלמה או לא. לכן, יש צורך בשני שדות:

- שדה Title מסוג טקסט (text) – שמיוצר באופן אוטומטי ברגע שמייצרים טבלה חדשה ולכן לא צריכים להוסיף אותו ידנית.
- שדה Completed מסוג בוליאן (boolean).

על מנת להוסיף שדה יש ללחוץ על כפתור הפלוס ואז ייפתח פאנל ההגדרות של השדה.

בפאנל ההגדרות יש שלושה קלטים שצריך למלא כאשר מייצרים שדה חדש:

Field Name – שם השדה שרוצים. במקרה הזה – Task.

- Field Key – זהו מפתח השדה שנוצר באופן אוטומטי על ידי השם שניתן לשדה. הערך של השדה ישמש אתכם כאשר תרצו לפנות לשדה הזה דרך קוד. לרוב אין צורך לגעת בערך שהמערכת נותנת באופן אוטומטי.
- Field Type – סוג השדה שרוצים לייצר. אפשר לבחור ממבחר גדול של סוגים.

כעת קוראים לשדה הנוסף Completed ומגדירים אותו מסוג Boolean.

לבסוף לוחצים על

והשדה יתווסף.

כעת יש טבלה בעלת שתי עמודות. אפשר להוסיף רשומות באופן ידני, לשנות אותן ואפילו למחוק אותן מתוך עורך התוכן.

למדתם את רוב המידע שיהיה רלוונטי לכם עבור מסדי נתונים ב-Corvid. עכשיו תלמדו על עבודה עם מסד הנתונים לאחר שמפרסמים את האתר.

Sandbox | Live

אם תשימו לב, בחלק העליון של עורך התוכן יש שני כפתורים:

- בלחיצה על כפתור ה-Sandbox יוצגו הרשומות שיופיעו בזמן עריכת האפליקציה. כלומר, הרשומות שרואים כאשר לוחצים על preview.
- בלחיצה על כפתור ה-Live יוצגו הרשומות שרואים באפליקציה האמיתית אחרי שלוחצים על publish.

Permissions

לכל טבלה יש הרשאות שמגדירות מי יכול לראות את הנתונים שבטבלה, מי יכול ליצור אותם, לערוך אותם ולמחוק אותם.

ההרשאות הראשוניות שמקבלים כשיוצרים טבלה חדשה הן מאוד שמרניות. כלומר, הן לא ייתנו 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 ההרשאות. עושים זאת באמצעות לחיצה על Edit Permissions:

ואז, ייפתח חלון שבו תוכלו לערוך את ההגדרות של ההרשאות. עבור אפליקציית המשימות שלכם, תרצו שההרשאות ייראו כך:

* בתבנית שיצרתי עבורכם כבר הגדרתי את ההרשאות של הטבלה בצורה הזו.

סיכום

סיימתם לכתוב את האפליקציה שלכם. מה למדתם?

- חיבור מידע מטבלאות הנתונים לאפליקציה
- יצירת אינטראקציות עם רכיבים של העורך הוויזואלי דרך הקוד
- צפייה, הוספה, עריכה ומחיקה של רשומות בטבלאות בעזרת wix-data
- יצירת שאילתות וסינונים על הרשומות
- פתיחת חלונית וסגירתה באמצעות קוד
- מי שקרא את הנספח – איך ליצור טבלאות חדשות

עכשיו הגיע הזמן שלכם ליצור אפליקציות חדשות, ללמוד על עוד יכולות ש-Corvid מספקת לכם ולהתנסות בהן.

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר הזה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוצרי העריכה. יותר מ-1800 אנשים תמכו בספר הזה ואיפשרו לו לצאת לאור.

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

העותק הזה נמכר ל:

arik500@gmail.com

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

תודה וקריאה נעימה!