

# HTML5 & CSS3 & JavaScript

## שיעור 1

מתחילים את העבודה קודם עם html ונעבוד דרך Visual Studio Code שהוא הפלטפורמה הנוחה של Microsoft.

השפה הזו (html), באופן כללי, היא שפה פשוטה שבעצם במקום לכתוב פיזית את הדברים אנו עושים זאת באמצעות פקודות.

כל העניין של דף אינטרנט עובד על הקונספט של שרת-לקוח, כאשר לקוחות מקבלים את השירות מהשרת ומתקשרים איתו. השרת הוא בעצם מחשב שיושב ומחכה לבקשות שיגיעו מהלקוחות.

כל האתרים עובדים רק עם השפות שהדפדפן מכיר שזה בכללי html ו-css שהיא אחראית לחלק של העיצוב.

ברגע שאני מתחיל לכתוב בקובץ ב'כתבן' שלנו, אני אוכל לפתוח אותו כמו אתר באינטרנט (לא שאפשר להיכנס אליו, הוא לא מפורסם..).

עמוד אינטרנט בתיאור של html בנוי בצורה שמורכבת משני חלקים:

1. מורכב מהגדרות כלליות של העמוד והוא נקרא Header. זה יכול להיות אפילו זכויות יוצרים, שזה לא חייב להיות כתוב אלא רק חלק מההגדרות של העמוד.
2. הגוף של שאר הדברים שנכתבים והוא נקרא – Body.

```
3. <html lang="en" dir="rtl">
4. <head>
5.     <meta charset="UTF-8">
6.     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7.     <meta name="viewport" content="width=device-width, initial-
    scale=1.0">
8.     <title>Document</title>
9. </head>
10. <body>
11.
12. </body>
13. </html>
```

וכך זה נראה ממעוף הציפור.

נדגיש כאן, שאני יכול לכתוב מה שאני רוצה כל עוד הוא בתוך תחום של תג סוגר ותג פותח.

```
<person>Gil</person>
```

נניח לדוגמא כאן, הגדרתי בן אנוש ששמו גיל.

אין לזה שום משמעות מעבר למה שאני הגדרתי.

כאן נזכיר את הפורמט xml שהוא פורמט שמסייע להעביר מידע בין מחשבים (כמו שיש לנו היום בעיקר את JSON).

הבעיה היחידה בפורמט הזה היא שזה קוד נפוח וגדול.

```
<person>
  <Age>30</Age>
  <Temp mesure_in="Celsius" precision="2">37.0</Temp>
  <Name>
    <FirstName>Gil</FirstName>
    <LastName>Alkobi</LastName>
  </Name>
  <Hair_color>Black</Hair_color>
</person>
```

זוהי תבנית שנראית בדיוק כפורמט xml.

חשוב שתמיד יהיה תג פותח ותג סוגר. בנוסף חשוב להצהיר בתחילת הקובץ שזהו קובץ xml (לא קיים בקוד כאן למעלה)

השורה נראית כך:

```
<?xml version="1.0" encoding="UTF-8"?>
```

**נחזור ל-html:**

כמה הבדלים קטנים בין html ל-xml:

1. כל התגיות ב-html הן קבועות שלא כמו xml שם אני יכול להגדיר מה שאני רוצה.
2. ב-html יש תגיות שלא חייבות תג סוגר.
3. שורת הפתיחה של העמוד שונה בשני הפורמטים.

דוגמא ל-3, זו שורת הפתיחה של html (היא אוטומטית ולא תמיד נכתוב אותה):

```
<!DOCTYPE html>
```

**אז נתחיל לעבור על הגדרות:**

יש תגית שאפשר לעבוד איתה שהיא תגיד כותרת שמשנה את השם שנמצא וכתוב על הכרטיסייה.

```
<title>Document</title>
```

היא נכתבת ב-header.

ישנן כותרות נוספות (יש 6 כאלו) שככל שאני עולה במספר המוצמד הכותרת הולכת וקטנה והופכת להיות כותרת משנה.

```
<h1>טפסים</h1>
<h2>טפסים - קטן יותר</h2>
<h3>טפסים - עוד יותר קטן</h3>
```

זה נכתב ב-body.

תגיד שפותחת פסקה היא האות p.

```
<p>Lorem ipsum dolor sit amet consectetur, adipisicing elit.
Numquam voluptate hic incidunt aperiam eos natus nulla
ipsum facere dolore delectus!
Corporis, alias explicabo dolorem facilis vitae ab quis maiores veniam.
</p>
```

כאן הוספנו טקסט שנקרא "טקסט זבל" ("lorem ipsum" שזה ממש לא התרגום, אבל...).

נשים לב שלרווחים (או סתם enter) אין משמעות מבחינת השפה, כך שהיא מכירה רק ברווח אחד. נוכל לרדת שורה באמצעות `<br>`.

בדרך כלל כדי לפתוח פסקה חדשה, לא נשתמש ב-`<br>` אלא נפתח פסקה חדשה.

אם נרצה להוסיף קטע כתוב באמצע פסקה נשתמש בתגית `span`.

```
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit.  
  Qui deleniti enim <span>Hello</span> quod officii distinctio  
  quisquam excepturi delectus nihil voluptate esse,  
  ullam illum iste,  
  doloribus eveniet laborum eligendi nobis?  
</p>
```

זה מסייע לנו בניח לעצב אחרת רק את החלק שבתוך התגיות הללו.

ב-html יש לנו שני סוגים של תגיות (בגדול): תגית בלוק ותגית inline שהיא תגית בתוך שורה.

נניח כותרת היא תגית בלוק, וגם תגית פסקה, זה אומר שבעמוד הן ממש תופסות את כל הבלוק על כל השורה שלו.

אבל תגיות inline כמו תגית `span` לא תפתחנה בלוק חדש אלא ישתלבו בתוך השורה.

ישנה תגית שנקראת `<b>` שהיא פשוט מדגישה את הטקסט שכתוב בין הפותחת לסוגרת.

תגית נוספת, `<strong>` שהיא גם כן מדגישה.

פעם היה הבדל בין שתיהן שהיה מעניין אותנו אבל היום לא באמת מעניין.

תגית `<u>` מניחה קו מתחת לטקסט שבין התגיות.

תגית `<i>` יוצרת כתב נטוי.

## תגיות מטא:

```
1 <meta charset="UTF-8">  
2 <meta http-equiv="X-UA-Compatible" content="IE=edge">  
3 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

אלו תגיות שנמצאות ב-`header`. ונעבור עליהן לפי סדר ההופעה כאן בקוד.

1. מגדירה את הקידוד של המסמך, שזה בעצם הצורה שבה המסמך נשמר מבחינה מיחשובית, כלומר, איך כל תו ותו נשמר אחר כך בקובץ טקסט בצורה הכי גולמית שלו.
2. זה אומר שזה יתאים לכל מיני דפדפנים.
3. אומר שהמסמך הנוכחי יתרחב ויקטן לפי גודל המסך.

רספונסיבי – התאמה לגודל המסך (החלקים של העמוד יסדרו עצמם בצורה שונה בהתאם לגודל המסך).

■ הגישה הנפוצה היום היא `mobile first` שזה אומר שמעצבים את האתר קודם שיהיה יפה במובייל ואחר כך דואגים לנראות על המחשב – זה כמובן נובע מכך שרוב המשתמשים פותחים את האתר מהפלאפון.

לפני שהתחלנו עבודה, התקנו תוסף שנקרא `live server`, והוא משמש אותנו לעבודה נוחה שהדפדפן נפתח ומקבל כתובת אחרת (שבנויה מה-IP של עצמי ובעצם מאזין לעמוד של האתר דרך שער אחד

והוא שלנו). את האפשרות הזו אנו צריכים להפעיל באמצעות מקש ימני => 'Open With Live Server'.

זה מדמה לנו מצב של שרת אמיתי (שאי אפשר לגשת לקבצים מאחורי הקלעים) ומאפשר לנו לראות עדכון אוטומטי של כל שינוי שנבצע לאחר שנשמור את השינויים – כך אפשר לעבוד עם שני המסכים במקביל (או לחלק את המסך לשניים או לשני מסכים פיזית).

אם נרצה להוסיף תמונה נצטרך שהתמונה תהיה בספרייה שלי שבתוכה אני כותב (ממש בתיקייה שלי).

נניח לצורך הדוגמא שקראנו לתמונה שלנו p1, אז אפשר לכתוב בצורה הבאה ואפילו לשנות את גודל ההצגה:

```

```

כאשר width מגדיר רוחב ו-height מגדיר אורך.

כדי שנוכל לראות את התמונה בצורה טובה נצטרך להיות במצב של live server.

אם נרצה לייבא תמונה מהאינטרנט נוכל לכתוב את הלינק של התמונה ב-src במקום שם התמונה מתוך המחשב.

### כמה דברים על תמונות באינטרנט:

ישנם כמה סוגים של תמונות (לפחות): PNG, JPG, JPEG (אותו דבר כמו JPG), BMP, GIF, SVG.

נתמקד ב-PNG וב-SVG, כל הפורמטים האחרים לא מסוגלים להכיל רקע שקוף.

המיוחד ב-SVG הוא שזו לא בדיוק תמונה אלא רצף של קוד שבונה את התמונה.

בדרך כלל עובדים מול סטודיו לעיצוב שמספק את התמונות ואנחנו צריכים להגיד להם מה הגדלים שאני זקוק להם ואם אני צריך את התמונה עם רקע שקוף וכו'..

חשוב שנגיד לאותו מעצב שישמור את התמונה ל-WEB כך שהאיכות יחסית נמוכה אבל עדיין לא ברמה שהעיניים רואות וכך היא תופסת פחות מקום וזה פחות עומס על האתר.

דבר נוסף, אם אנו לא עובדים מול מעצב אז יש מספיק תוכנות שמסייעות לנו לכווץ את הגודל כדי לאפשר הקלה על העומס.

אתר נחמד שאפשר ללמוד בו יותר על כל הקשור לרישיונות (לגבי תמונות שלקחתי ממקור אחר):

<https://tldrlegal.com>

נמשיך הלאה..

אם נרצה שהכיתוב יהיה מימין לשמאל, נוסיף בשורה הראשונה שלפני ה-header את האפשרות dir="rtl" שזה אומר, כיוון (direction) right to left (כאילו rtl).

```
<html lang="en" dir="rtl">
```

### תגית Table:

אם נרצה להגדיר טבלה בעמוד שלנו, נשתמש בתגית <table>.

בתוך התגיד נוכל לשחק עם התגיות <tr> (שהיא אומרת שאני כותב כרגע שורה בטבלה), <th> (שהיא אומרת שאני כותב תא שמוגדר להיות כותרת או ראש עמודה) ו-<td> (שזו כתיבה לתא רגיל) ותכף נראה עוד כלים.

```
<table>
  <tr>
    <th>AAA</th>
    <th>BBB</th>
    <th>CCC</th>
  </tr>
  <tr>
    <td>aaa</td>
    <td>bbb</td>
    <td>ccc</td>
  </tr>
</table>
```

יש לנו אפשרות להשתמש ב-`colspan` או `rowspan` כדי לאחד תאים.

```
<table border="1">
  <tr>
    <th>AAA</th>
    <th>BBB</th>
    <th>CCC</th>
  </tr>
  <tr>
    <td colspan="2">aaa</td>
    <td>bbb</td>
  </tr>
  <tr>
    <td rowspan="3">aaa</td>
    <td>bbb</td>
    <td>ccc</td>
  </tr>
  <tr>
    <td>aaa</td>
    <td>bbb</td>
  </tr>
</table>
```

וכך הטבלה תיראה:

AAA	BBB	CCC
aaa		bbb
aaa	bbb	ccc
	aaa	bbb

התגית <a> מקשרת בין עמודים שונים.

```
<p>שלום וברוך הבא לאתר</p>
  <a href="/other.html">לחץ כאן לעבור לעמוד השני</a>
</p>
```

לצורך הדוגמא, other הוא הקובץ השני אליו אנו מקושרים.

אם נגדיר בפסקה מסוימת בהגדרה שלה id כלשהו, נוכל ליצור קישור עם <a> לעוגנים בעמוד.

נניח שזו הכותרת שנמצאת איפשהו במהלך העמוד:

```
<h1 id="o1">1 כותרת של עוגן</h1>
```

והגדרתי בראש העמוד את ההגדרה הבאה:

```
<a href="#o1">1 לחץ כאן למעבר לעוגן</a>
```

נשים לב שה-id הוא ספציפי לעוגן הזה.

תגיות נוספות לרשימות:

יש תגית לרשימות מסודרות (ממוספרות) ויש רשימות ללא סדר ספציפי (נניח שמסומנות בנקודה).

<ul> - רשימה לא מסודרת.

<ol> - רשימה מסודרת.

וכדי להוסיף פריטים לרשימה נשתמש בתגית <li>.

```
<ul>
  <li>First</li>
  <li>Sec</li>
  <li>Third</li>
  <li>Fourth</li>
</ul>
```

כמובן שאם היינו רוצים שהרשימה תהיה מסודרת אז במקום ul היינו רושמים ol.

תגית <hr> יוצרת קו מפריד בין בלוקים.

יש לנו גם את התגית <div> שבעקרון דומה מאוד לתגית <p> אבל השימוש בה הוא יותר כדי לתפוס רוחב מלא.

חלק מהשימושים בתגית הזו היא ליצור עיצוב של בלוק מסוים.

כאן המקום להתחיל קצת css.

```
<div style="background-color: aquamarine; border: 3px solid burlywood;">
  Lorem ipsum
  <span style="background-color: crimson; border: 2px dotted black;">CONTENT</span>
  dolor sit amet, consectetur adipiscing elit. Expedita similique, deleniti
  blanditiis iure voluptas
  animi nihil sit ad cupiditate rem quod dolor officiis reprehenderit assumenda distinctio nam eveniet modi.
  Soluta.
</div>
```

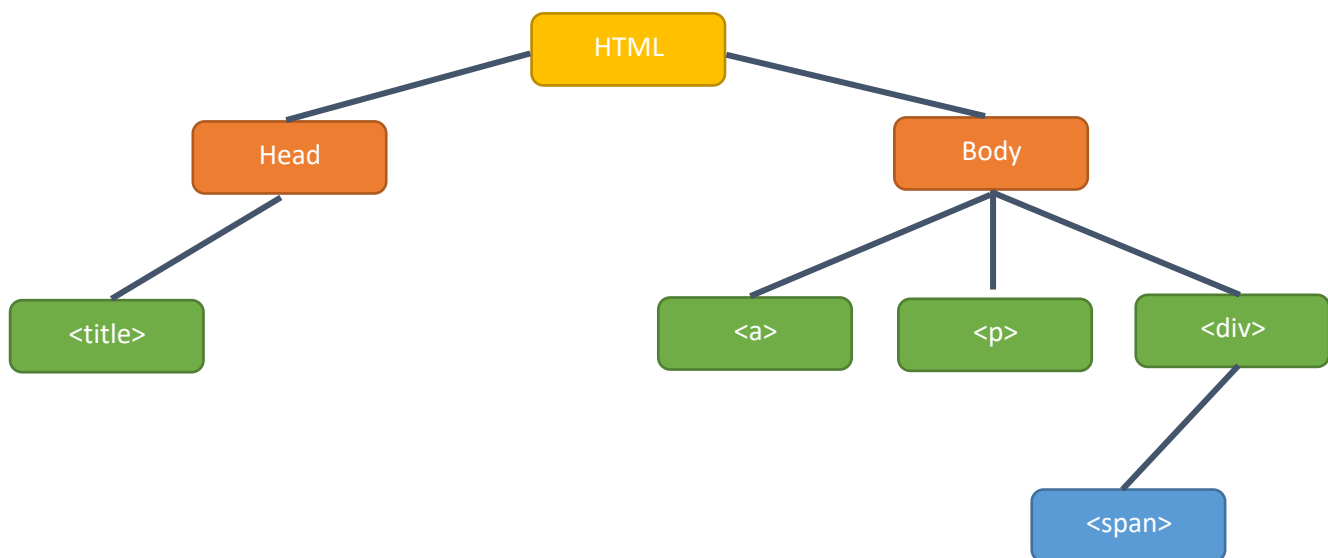
אם היינו מניחים את הסמן של העכבר על הריבוע של הצבע תיפתח לנו חלונית בחירה של צבעים.



נשים לב שהעיצוב נכתב בצורה מאוגדת (נניח על border) ובסדר מסוים שצריך להישמר.

### Document Object Model – DOM

הרעיון הוא שהדפדפן בונה לעצמו ייצוג כלשהו של העמוד. מבחינת האובייקטים העיצוב תמיד מתחיל מעץ.



וכך העץ יכול להמשיך בהתאם לתגיות שנפתחות.

זה כמובן מסייע לאלמנט התכנותי לגעת בנקודות שבהן הוא צריך להשתמש.

#### כעת נעבור לטפסים:

התגית שאנו משתמשים לשימוש בטפסים היא תגית <form>.

נראה את הקוד ונבין משם.

```

<h1>טפסים</h1>

<form action="./formResults.html" method="get">
  <label>
    הקלד בבקשה שם משתמש
    <input type="text" name="user" id="user">
  </label>
  <label>
    הקלד בבקשה סיסמא
    <input type="password" name="password" id="password">
  </label>
</form>
<div>
  <label>
    תאריך לידה
    <input type="date" name="birthDate" id="birthDate">
  </label>
</div>
<div>
  <label for="agree">
    אני מסכים לתנאים
    <input type="checkbox" name="agree" id="agree" checked="checked">
  </label>
</div>
<div>
  <button type="Submit">שלח</button>
</div>

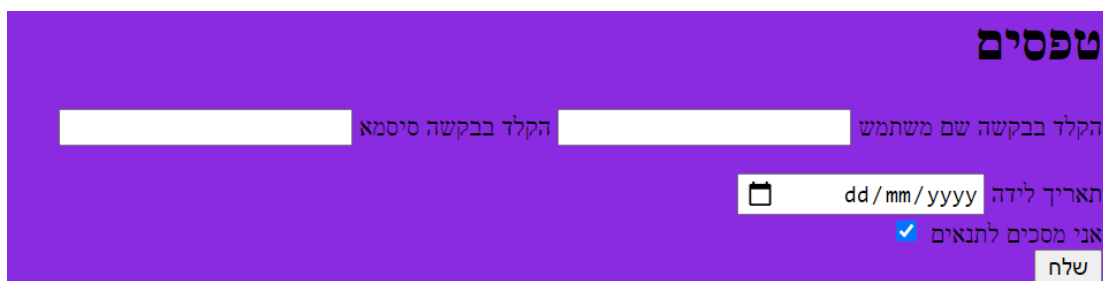
```

במתודה של הקישור שכרגע מוזן בה "get" יש גם אפשרות ל-"post" אבל לזה נגיע ב-javascript.

כל פעם שנלחץ על הכפתור של submit המידע יישלח אל העמוד שמקבל והשרת יוכל לעבוד עם הנתונים האלו.

נשים לב שהשתמשנו בתגית <label> שלרוב אין לזה כל כך משמעות אבל זה עוזר בכך שזה יתפוס את אותה השורה ולא פסקה חדשה כמו <p>.

ניתן לראות שלפי ההגדרה של ה-"type", כך יוגדר החלק הספציפי בטופס.





## שיעור 2

### בהמשך לנושא הטפסים מהשיעור הקודם:

יש עוד אפשרויות להוספה של שדות מסוימים לתוך טופס.

אני יכול להשתמש בפקודה place holder שהיא מדגישה מה צריך למלא בשדה הספציפי הזה גם ללא הצורך בכתיבת כותרת ב-label.

בנוסף, ראינו את האפשרות ליצירת תאריך ושנה גם אפשרות ליצור הגבלת של תאריך מינימום ותאריך מקסימום.

■ אתר שיכול לעזור להיזכר בכל מיני פקודות כאלו ואחרות:

<https://developer.mozilla.org/en-US>

```
<div>
  <label for="agree">
    אני מסכים לתנאים
    <input type="checkbox" name="agree" id="agree" checked="checked">
  </label>
</div>
```

כאן אנו רואים כתיבה של label לפי התקן שכתוב בתגית למי החלק הזה מיועד. החלק הזה נקבע על ידי ה-id שמוגדר ב-input.

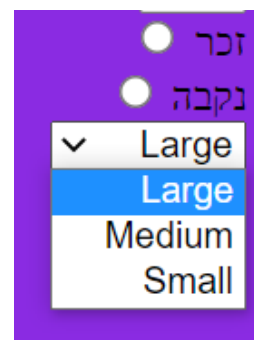
והחלק האחרון (checked="checked") שהוא בעצם מאפשר את הסימון של הבחירה.

ראינו גם כפתורי radio, שמאפשרים לי בחירת ברירה.

```
<div>
  <label for="male">
    זכר
    <input type="radio" name="sex" id="male" value="male">
  </label>
</div>
<div>
  <label for="female">
    נקבה
    <input type="radio" name="sex" id="female" value="female">
  </label>
</div>
```

עטפנו כל אחד מהם ב-<div> וב-<label> כדי שיופיעו בנפרד והגדרנו להם את אותו השם כדי שהעמוד ידע שזו אותה הקבוצה וכך לא נוכל לסמן את שניהם יחד.

נשים לב שכך זה נראה, ובעמוד הבא נראה הסבר וקוד לחלק של פתיחה של רשימה לבחירה.



אז כדי לפתוח רשימת בחירה, לצורך הדוגמא, של מידת חולצה, נכתוב כך:

```
<div>
  <label for="Shirt-Size">
    <select name="Shirt-Size" id="Shirt-Size">
      <option value="L">Large</option>
      <option value="M">Medium</option>
      <option value="S">Small</option>
    </select>
  </label>
</div>
```

כך שבכל אופציה רשום ערך שיישלח לפי הבחירה, כלומר, אם בוחרים מידה large יישלח הערך "L".

אם נמלא את הטופס (וגם נשלח) כך:

זו השורה שנקבל ככתובת של העמוד שהמידע נשלח אליו:

http://127.0.0.1:5500/Lesson\_Excercises/ExLesson1-2/formResults.html?user=%D7%92%D7%99%D7%9C%D7%90%D7%9C%D7%A7%D7%95%D7%91%D7%99&password=12345&birthDate=1991-02-26&agree=on&sex=male&Shirt-Size=L

אנו רואים שכל המידע ששלחתי נמצא בכתובת. החלק של שם המשתמש יצא בצורה של encoded שזה בעצם כתיבה לפי סימני קידוד ascii.

**עוד כמה שדות שיכול להיות מאוד נוח להשתמש בהן:**

```
<div>
  <label for="email">
    אימייל
    <input type="email" name="email" id="email">
  </label>
</div>
<div>
  <label for="age">
    גיל
    <input type="number" name="age" id="age">
  </label>
</div>
<div>
  <label for="phone">
    טלפון
    <input type="tel" name="tel" id="phone">
  </label>
</div>
```

כאשר אנו רואים אפשרויות מוכרות שהן הזנת מייל, טלפון או גיל וההגדרה שלהם מאפשרת לטופס להישאר צמוד לפורמט הכתיבה החוקי של השדות הללו.

וכמובן את שני הכפתורים שלנו:

```
<div>
  <button type="reset">אפס טופס</button>
  <button type="Submit">שלח</button>
</div>
```

כאשר הראשון הוא לאיפוס טופס והשני הוא לשליחת הטופס.

יש כמובן עוד הרבה שדות שאפשר למצוא את כולם באינטרנט כמו זמן, טווח, חיפוש וכדומה..

יש תגית שנקראת `<nav>` שהיא תגית navigation שלא משפיעה על העמוד או על התצוגה אלא בא להגיד למי שקורא את הקוד (בעיקר מנועי חיפוש) שהחלק הזה יש בו קישור למקומות אחרים.

תגית `<header>` שגם כן רק מדגישה שמדובר בחלק הזה בכותרות ולא משפיעה על התצוגה.

`<footer>` מגדירה את החלק התחתון של האתר וגם כן אותה מטרה כמו הקודמים.

`<aside>` בא להגדיר מה שכתוב בצד האתר, התגית הזו היא לא זו שמגדירה את העמידה בצד אלא רק מגדירה שזה אמור להיות כתוב בצד – את העיצוב מגדירים במקום אחר.

`<section>` - מיועדת לכל דבר שנרצה, נניח לחלק קטע גדול שלנו למקטעים.

`<article>` - אם רוצים לציין למנוע החיפוש שעיקר הכתבה נמצאת בחלק הזה וכל השאר היה דברים נלווים וגם כאן זה לא חלק מחייב אלא רק הכוונה לקורא הקוד שלנו.

## עכשיו נעבור ל-CSS:

זהו בעצם השלב השני, שהוא שלב העיצוב.

התרגום הוא גיליונות עיצוב מדורגים – Cascading Style Sheets.

שלוש דרכים לכתיבת העיצוב:

1. INLINE – בתוך השורה – שילוב הכתיבה בתוך התגית הרצויה של HTML.
2. INTERNAL – יצירת תגית `<style>` בדף ה-HTML.
3. EXTERNAL – יצירת קובץ חיצוני. (השיטה המומלצת)

נעבור על הדרכים:

**:INLINE**

```
<span style="background-color: crimson; border: 2px dotted black;">CONTENT</span>
```

כאן אנו רואים ממש כתיבה של עיצוב בתוך תגית HTML.

כאן המקום להגיד שזו הדרך הכי "חזקה", כלומר, היא תדרוס כל עיצוב שייכתב על המקום הנוכחי באחת מהדרכים הקודמות.

לרוב נעדיף שלא להשתמש כך אלא רק במקרים מיוחדים.

**:INTERNAL**

```
<style>
  body {
    background-color: blueviolet;
  }
</style>
```

את התגית הזו בדרך כלל (לפי התקון) נשים בחלק של ה-head (על אף שזה יעבוד גם ממקום אחר).

בעצם בקוד כאן ראינו שבחרנו צבע רקע לחלק של ה-body (לעמוד בעצם).

כמובן שגם בדרך הזו נעדיף שלא להשתמש.

בנוסף נשים לב שעד עכשיו התעסקנו בעיצובים פשוטים כמו רקע או גבול, אבל בהמשך נראה דברים מעניינים יותר.

- כאן המקום להעיר הערה לגבי הערות 😊  
כתיבת הערות ב-HTML מתבצעת באמצעות `<!--hghg-->`

```
<!-- הערה -->
```

- וכמובן אם נרצה לעטוף קטע בהערה אפשר להשתמש ב- `/*annotation*/`

```
/*
  body {
    background-color: blueviolet;
  }
*/
```

**:EXTERNAL**

במונחים המקצועיים אנו קוראים לקבצים שמגיעים עם האתר "נכסים" ("assets"). הנושא הזה הוא עניין מקובל ואינו חובה – אבל תמיד נוח לעבוד לפי המקובל, כך השפה המשותפת נוחה יותר.

את הקובץ נקשר לקובץ שלנו בתוך החלק של ה-head באמצעות הקוד הבא:

```
<link rel="stylesheet" href="./Assets/CSS/main.css">
```

את התוכן עצמו פשוט נכתוב בקובץ חדש מסוג CSS (זה שקראנו לו main כאן בקוד), כאשר הכתיבה עצמה היא כמו בתוך תגית style.

ההיררכיה של ה"חוזקה" היא כך:

ה-INLINE דורס את כולם אבל בין השניים האחרים זה תלוי מי מופיע לפני מי.

**סלקטורים:**

אם הייתי מחליט לכתוב div במקום body (רק לצורך הדוגמה), אז העיצוב יחול על כל תגיות ה-div בעמוד.

כמובן שיכולתי להחליט עיצוב אפילו רק על כותרת מסוג ספציפי כמו לשנות לה גופן.

- נשים לב שאנו תמיד מזינים שלושה סוגי פונטים כדי שתהיה למערכת אפשרות לקפוץ ולבחור בפונט הבא למקרה שהקודם לא קיים לי במחשב (חייב להיות לי במחשב).

```
<style>
  h1{
    font-family: Arial, Helvetica, sans-serif;
  }
  div{
    background-color: blueviolet;
  }
</style>
```

עד עכשיו ראינו דוגמא ראשונה לסלקטור – שזה **בחירה של סוג התגית** עליה יחול העיצוב.

**דוגמא שנייה זה class:**

הגדרה של סלקטור class מתבצעת בצורה הבאה:

```
.kishurim{
  font-size: 18px;
  border: 1px dotted blue;
  font-family: Arial, Helvetica, sans-serif;
  background-color: beige;
}
```

והמימוש של הסלקטור יהיה כך:

```
<p>
  לחצו על הקישור למעבר לעמוד
  <a class="kishurim" href="./index.html">זה הקישור</a>
</p>
```

■ התגית <a> משמשת ליצירת קישור לעמוד אחר.

נניח שכתבתי עוד class:

```
.specialLinks{
  cursor: pointer;
}
```

נוכל להוסיף פשוט עם רווח בצורה הבאה:

```
<p>
  לחצו על הקישור למעבר לעמוד
  <a class="kishurim specialLinks" href="./index.html">זה הקישור</a>
</p>
```

**סלקטור שלישי וחשוב לא פחות הוא הסלקטור ID:**

אנו מקצים id לתגית ואז בעיצוב נוכל להגדיר עיצוב ל-id הספציפי באמצעות הוספת # לתחילת ה-id.

```
<div id="userDiv">
  <label>
    הקלד בבקשה שם משתמש
    <input type="text" name="user" id="user">
  </label>
</div>
```

כאן הגדרנו id ל-div של שם המשתמש בטופס.

```
#userDiv{
  font-size: 20px;
  color: white;
}
```

וכך אנו מגדירים עיצוב לבעלים של ה-id.

נוכל "לשרשר" שני classes וכך רק אם התגית תממש את כל ה-classes אז השינויים יחולו.

במצב כזה נוכל לממש את ה-classes הללו רק אם יוזנו יחד. לרוב זה מצב שאני מקבל פרויקט של מישהו אחר ואני לא יכול לגעת בחלק מהאלמנטים, ואז כדי להתעסק עם האלמנטים מבלי לגעת אני יכול להוסיף מחלקה בצורה שכזו.

אם נגדיר אותם עם רווח ביניהן, אז השנייה תוכל להתממש רק אם התגית האבא מממשת את הראשונה.

## שיעור 3

### רגע השלמה:

אם נרצה להוסיף ברקע תמונה, נשתמש ב-Background-image ונוסיף כתובת ב-url.

### Padding, Margin ומודל התיבה:

Wrapper – מעטפה, נקרא כך בדרך כלל כשנרצה לעטוף בעיצוב מסוים.

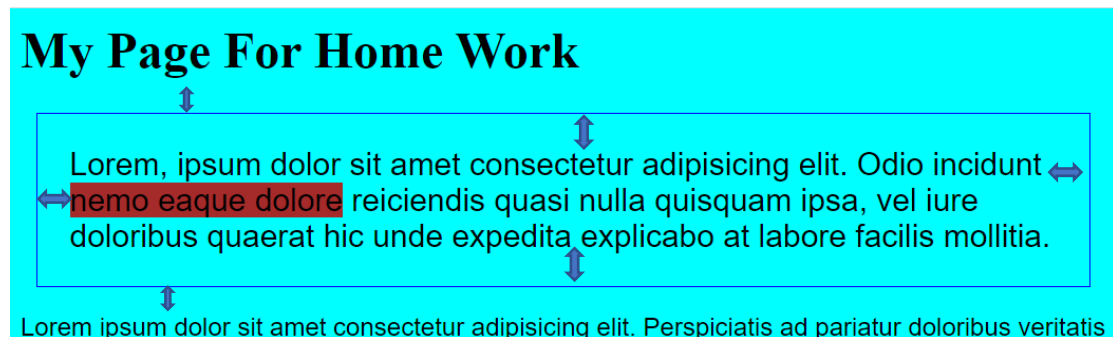
Margin – גבול חיצוני.

Padding – ריפוד פנימי.

כאשר ניצור גבול סביב כותרת מסוימת, ה-padding ייצג את המרווח בין הכותרת לבין הגבול. כמובן שיש לנו אפשרות להגדיר padding לכל כיוון בנפרד.

```
.sub-header{
  border: 1px solid blue;
  /*padding: 20px;*/
  padding-top: 20px;
  padding-bottom: 20px;
  padding-left: 20px;
  padding-right: 20px;
  margin: 10px;
}
```

כך זה ייראה אם נצמיד את זה לטקסט כלשהו:

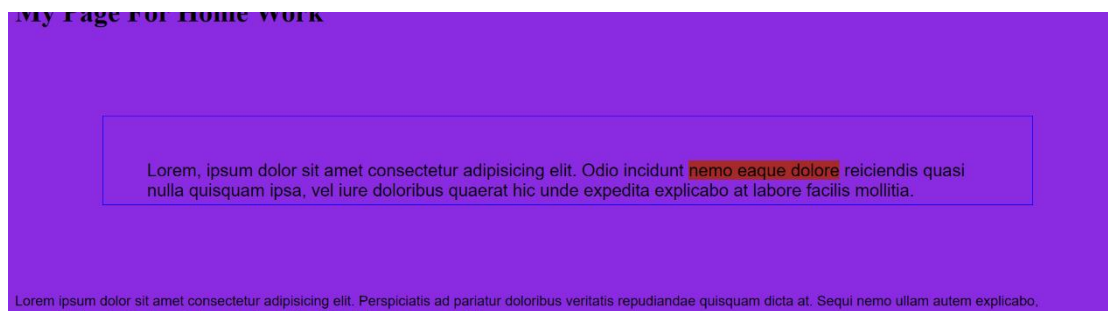


### הכללים הם כך:

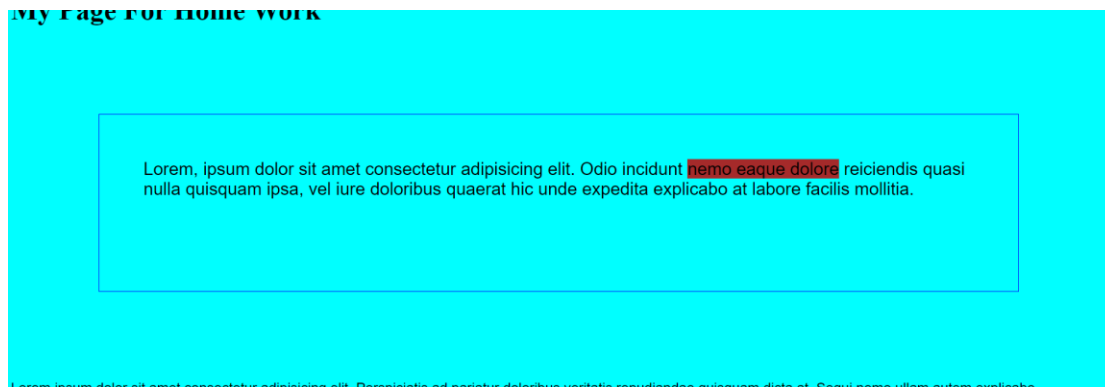
1. Margin מתאחד בין שורות, כלומר, ה-top של השורה התחתונה וה-bottom של השורה העליונה מתאחדים (אם top=20 וגם bottom=20 אז המרווח לא יהיה 40 אלא רק 20).
2. בבולקים מסוג inline, ההתנהגות שונה, margin מתווסף לצדדים וה-padding מכל כיוון.

יש אלמנט שנקרא border-box. שבו כאשר אני מגדיר אותו אז הכל יוגבל ביחס לאותה הקופסא, כלומר אם נגדיר margin אז התיבה תהיה חלק מה-margin.

```
.sub-header{
  border: 1px solid blue;
  padding: 50px;
  margin: 100px;
  height: 100px;
  box-sizing: border-box;
}
```



עם border-box.



בלי border-box.

נמשיך..

אם נוסיף לעיצוב של פסקה שהיא inline את האפשרות display, נוכל להגדיר לו כל מיני אפשרויות, כמו נניח שיתפוס בלוק משלו. כמובן שהבלוק יהיה 100% מבלוק האב ולא מהמסך כולו.

יש אפשרות שנקראת inline-block שאם נגדיר אותה ונגדיר גם margin אז הפעם ה-margin יתפוס לכל כיוון ולא רק לצדדים.

נדגיש כאן שאנו יכולים להגדיר גדלים גם בצורה של אחוזים ולא פיקסלים, רק שכאן בחלק מהמקרים, תלוי בסוג התגית, זה יתפוס את האחוז ביחס לחלק האב ולא כל ה-body.

מידה נוספת שקיימת היא view שהיא בעצם לוקחת את האחוז ביחס למסך הקיים במחשב.

## נחזור לסלקטורים:

הרבה פעמים עובדים עם ספרייה צד שלישית, כלומר, ספרייה שמביאה לנו רכיבים ויזואליים למסך וצריך לעצב אותם ולכן צריך סלקטור כדי לבחור את האלמנט בתוך הספרייה.

עם סלקטורים אפשר גם לבחור דברים בצורה תכנותית (נלמד יותר ב-JS), כל אלמנט שיש לו class או id אפשר לשחק איתו.

ראינו בהמשך לשיעור קודם שכאשר אני מגדיר עיצוב לפי תגית אני יכול גם להגדיר לפי היררכיה, נניח בדוגמה הבאה:

```
p a{
  background-color: brown;
}
```

אני קובע צבע רקע לכל תגית a שנמצאת בתוך תחום תגית p.



דוגמא נוספת שיש לנו היא אפשרות ליצור שינוי צבע ב-textbox כאשר אני מרחף מעליה או כאשר אני בתוכה בזמן הזנת מידע

```
input:focus{
    background-color: aquamarine;
}

input:hover{
    background-color: chartreuse;
}
```

דבר נוסף שיש לי הוא על רשימה, כאשר אני רוצה לעצב את השורה הראשונה אני יוצר את הדבר הבא:

```
li:first-child{
    background-color: purple;
}
```

ואז בעצם השורה הראשונה ברשימה תעוצב לפי מה שהוגדר כאן.

אם הייתי רוצה שורה אחרת, הייתי מגדיר כך:

```
li:nth-child(4){
    background-color: purple;
}
```

וכך בעצם בכל רשימה שתופיע באותו עמוד העיצוב הזה יחול.

כאשר אנו כותבים את כל העיצובים בדף CSS עצמאי, אנו מסדרים בסדר הבא:

1. Id
2. Class
3. אלמנטים רגילים.

ככל שהסקלטור יותר ספציפי הוא ידרוס את הקודמים לו.

בכל class ניתן להגדיר אותה inherit באחת ההגדרות או ביותר וזה יירש את אותה ההגדרה שיש במחלקה ששולטת בחלק שמעליו בהיררכיה.

אנו יכולים גם לקבוע משתנה שנשתמש בו במקומות אחרים, נניח:

```
body{
    --main-bg-color: brown;
}

p{
    color: red;
    background-color: var(--main-bg-color);
}
```

אם אגדיר לאחר מכן אלמנט נוסף אוכל לדרוס בו את המשתנה ולהזין לו משהו אחר. ואז אם דרסתי אותו נניח בתוך אלמנט של div אז רק כאשר זה יגיע בתוך div שם הערך הדורס יתקיים.

ישנו פסודו-אלמנט שהוא בעצם מעל לכל (גם מעל body) והוא נקרא root: – ואם אגדיר אותו אז הוא מעל לכל.

```
.general{
  color: blue;
  background-color: wheat;
  font-size: larger;
  text-decoration: line-through;
  font-family: 'Times New Roman', Times, serif;
  line-height: 50px;
  letter-spacing: 0.1em;
}
```

ראינו בקוד שיש אפשרות להגדיר קו חוצה על הכתב (שורה שלישית בתוך ה-class), פונטים שכבר ראינו קודם, רווח בין שורות (שורה שישית) ואפשרות של רווח בין האותיות (שורה אחרונה).

### השלמה קטנה לגבי Margin ו-Padding:

יש אפשרות להגדיר margin:auto אבל חשוב לזכור שזה עובד רק בתוך container. ואז זה גם יישר את הטקסט לאמצע.

יש מאפיין נוסף שנקרא float שהוא בעצם משפיע על האלמנט איתו הוא עובד בצורה שבה האלמנט "צף" לפי מה שהגדרנו.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
  <style>
    #leftBox{
      background-color: red;
      height: 280px;
      width: 25%;
      float: left;
    }
    #middleBox{
      background-color: green;
      height: 280px;
      width: 25%;
      float: left;
    }
    #rightBox{
      background-color: blue;
      height: 280px;
      width: 25%;
      float: left;
    }
  </style>
</head>
```

```
<body>
  <h1>Float Example</h1>
  <div id="leftBox">bla bla bla</div>
  <div id="middleBox">bla bla bla</div>
  <div id="rightBox">bla bla bla</div>
</body>
</html>
```

נשים לב שהגדרנו את כולם float:left וזה בעצם לקח כל אחד לשמאל שנותר לו, כלומר הראשון הוגדר בשמאל ואז מימינו זה השמאל החדש וכן הלאה.

וכך זה נראה:

## Float Example



כעת, אם היינו כותבים באחרון, נניח, float:right אז הוא היה נדחף הצידה יותר והיה נראה כך:

## Float Example



כאן המקום להדגיש שהשתמשנו כאן בתגית div מה שבדרך כלל אמור להופיע אחד מתחת לשני, אבל כאן הציפה של float חזקה יותר ומעמידה אותם אחד ליד השני.

## :Flex-Box

בעברית – קופסא גמישה.

כאשר העולם התחיל לדרוש יותר עיצובים, הועדה שאחראית על כך הבינה שלא יעיל לעצב בשיטות הישנות ולכן פותחה השיטה של flex-box ומה שהיא מאפשרת זה לקחת קופסא, קונטיינר שנוכל לעצב בצורה מסוימת את כל התכולה שלו.

אז הכנו דף HTML וני דפי CSS לעיצוב העמוד.

באופן כללי, יש לנו מיכל, קופסא (קונטיינר) שאנחנו מעבדים את האלמנטים לתוך המיכל.

ברגע שהוספנו את הפקודה: display: flex הפכנו את הקונטיינר לקופסא גמישה.

נציג כעת את העיצוב הכללי של הדף ובשיעור הבא נעמיק יותר בכל הנושא של קופסא גמישה.

נתחיל מהגדרה ב-HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
  <link rel="stylesheet" href="basic.css">
  <link rel="stylesheet" href="index.css">
</head>
<body>
  <h1>Start with flex</h1>
  <nav class="container">
    <div class="home">Home</div>
    <div class="search">Search</div>
    <div class="logout">Logout</div>
  </nav>
</body>
</html>
```

זוה העיצוב בתיקיית basic.css:

```
.container > div {
padding: 10px;
text-align: center;
font-size: 2em;
color: #fffeed;
}

html, body {
background-color: #fffeed;
margin: 10px;
}

.container > div:nth-child(1) {
background-color: #96ceb4;
}

.container > div:nth-child(2) {
background-color: #ff6f69;
}

.container > div:nth-child(3) {
background-color: #88d8b0;
}
```

## Start with flex



וכך העמוד נראה.

כעת באמצעות תיקיית index.css נוכל להתחיל לשחק בכל הקשור לקופסא הגמישה.

בקובץ השני נוכל להגדיר כיוון עמודה או שורה (כרגע במצב שורה).

נוכל גם להגדיר איך התאים home, search ו-logout יחולקו מבחינת מקום בתוך הקונטיינר.

## שיעור 4

אז התחלנו תצורה של display בשם flex-box שנמשך היום ובשיעור הבא נלמד גם Grid.  
עם התצורות האלו אפשר ליצור אתר בצורה די טובה כבר עם שני הכלים הבסיסיים HTML ו-CSS.  
נראה בשיעור הזה גם את הכלי שנוסף ל-CSS בשם SASS.

נזכיר שיש אלמנטים מסוג בלוק שתופסים את כל הרוחב המוקצה (מסך או חלקו) ויתחילו שורה חדשה, ויש אלמנטים מסוג inline שלוקחים קטע בשורה ולא מתפרסים.

ראינו שאפשר גם ליצור בלוק בתוך שורה עם margin ו-padding – מה שבדרך כלל אין לגמרי לאלמנט inline.

■ הערה: אני יכול להפוך תגית span להיות בתצורה של בלוק ואז היא תתפוס ממש שורה משלה אבל צריך לזכור שלא נהוג לעשות כך ולזכור גם שהקוד הזה כנראה יצטרך לעבור אצל עוד אנשים שלא בהכרח יבינו שזה מה שקורה שם.

אז לשימוש ב-FlexBox אנו בעצם יוצרים מעין מיכל שבתוכו יש אלמנטים שהם בנים ישירים שלו (בדגש על ישירים!).

■ הערה: ישנה מוסכמה שקבצים בשם index עם סיומת מסוימת, הם קבצי ברירת מחדל לבחירת שרת WEB, כלומר, שכאשר השרת פונה אל תיקייה מסוימת הוא יפנה לקובץ שייקרא index כברירת מחדל אלא אם כן אמרנו לו משהו אחר.

אז יצרנו לעצמנו עמוד אינטרנט שבתוכו רשימה של קישורים, כאשר כל אחד מן הקישורים מוביל אותנו לתרגול של חלק אחר של FlexBox. כמובן שאני מניח את זה כאן לטובת תזכורת של פקודות קודמות כמו רשימה מסודרת וקישורים לעמודים אחרים.

```
<body>
  <h1>Flex - Menu</h1>
  <ol>
    <li>
      <a href="/1/">Start with flex</a>
    </li>
    <li>
      <a href="/2/">Flex Direction</a>
    </li>
    <li>
      <a href="/3/">Justify content inside flex</a>
    </li>
    <li>
      <a href="/4/">Customized justify</a>
    </li>
    <li>
      <a href="/5/">Making it Responsive</a>
    </li>
    <li>
      <a href="/6/">Align on vertical axis</a>
    </li>
    <li>
      <a href="/7/">Little trick to center</a>
    </li>
  </ol>
```

```
<li>
  <a href="./8/">יישור אלמנטים ספציפיים</a>
</li>
<li>
  <a href="./9/">Wrap</a>
</li>
<li>
  <a href="./10/">Basis, Shrink & Grow</a>
</li>
</ol>
</body>
```

כל קישור שולח אותנו לתיקייה אחרת שבתוכה הוא מחפש את הקובץ שנקרא index.html כמו שכתבנו בהערה בעמוד הקודם.

כל אחד מן העמודים אליהם נפנה יכיל את הקוד הבא וכל מה שנשנה זה את קובץ ה-CSS שלה.

```
<body>
  <h1>Start with flex</h1>
  <nav class="container">
    <div class="home">Home</div>
    <div class="search">Search</div>
    <div class="logout">Logout</div>
  </nav>
</body>
```

Start with flex

Home Search Logout

ואלו הגדרות העיצוב הבסיסיות של העמוד:

```
.container > div {
padding: 10px;
text-align: center;
font-size: 2em;
color: #fffeed;
}

html, body {
background-color: #fffeed;
margin: 10px;
}

.container > div:nth-child(1) {
background-color: #96ceb4;
}
```

```
.container > div:nth-child(2) {  
    background-color: #ff6f69;  
}  
  
.container > div:nth-child(3) {  
background-color: #88d8b0;  
}
```

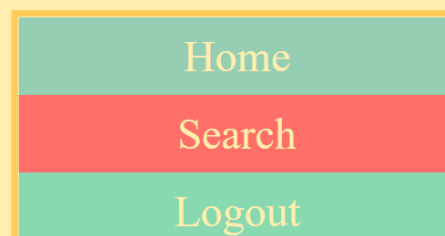
הראשונה מגדירה את המיכל, השנייה את ה-body כולו והשלוש האחרונות מגדירות שלושה בנים של ה-div של ה-container.

בכל אחד מהמקרים שהגדרנו ברשימה שיצרנו שיחקנו בעצם בצורה אחרת עם ה-container שלנו.

**במספר 2** ברשימה ראינו כיוויות של סידור האלמנטים בקופסא, ניתן להגדיר כשורה או כעמודה.

```
.container {  
    border: 5px solid #ffcc5c;  
    display: flex;  
    flex-direction: column;  
    flex-wrap: wrap;  
}
```

### Flex Direction



**במספר 3** ברשימה, ראינו יישור של אלמנטים בתוך המיכל באמצעות ה-justify-content.

יש לנו flex-end שמיישר את כולם לסוף השורה/עמודה ו-start לתחילת השורה/עמודה (מתהפכים במקרה שאנו מגדירים את העמוד מימין לשמאל).

יש center (כולם במרכז), space-around (נמצאים במרווחים שווים ביניהם אבל לא ביחס לצדדים), space-evenly (ממש שווה ברווחים כולל בצדדים) ויש space-between שהקיצוניים נצמדים לצדדים והרווח ביניהם שווה.

### Justify content inside flex



בתמונה אנו רואים דוגמא של:

```
.container {  
    border: 5px solid #ffcc5c;  
    display: flex;  
    justify-content: space-between ;  
    flex-wrap: wrap;  
}
```



■ הערה: ראינו שאם אנו כותבים אתר שיהיה לא אישי כמו בלוג, אז אני צריך לעמוד בתקן של נגישות ולכן נצטרך להוסיף תיאור של קישור.

**במספר 4** ברשימה ראינו אפשרות לסידור מותאם אישית של האלמנטים באמצעות שימוש ב-margin: auto, אם נגדיר את Logout באמצעות margin-left: auto, אז הוא פשוט יהיה בצד הכי שמאלי של ה-container. ואם נגדיר את ה-Search באותה הגדרה במקום (רואים בתמונה), הוא ידחף איתו גם את Logout.

### עימוד מותאם אישית



**במספר 5** ברשימה יצרנו אפשרות שבה כל הקוביות שב-flex יהיו רספונסיביות, שזה אומר שהם יקטנו ויגדלו ביחס לגדילת והקטנת המסך.

הפעם בשונה מהקודמים יצרנו משהו שתופס על כל האלמנטים. כך בעצם אנו מגדירים flex: (number).

זה בעצם יוצר מצב שבו כאשר נרחיב או נצמצם את המסך התאים יחד עם הקופסא יתרחבו או יצטמצמו בהתאם.

**במספר 6** ברשימה ראינו אפשרות ליישור של כל האלמנטים על פי ערך מסוים.

אם נניח היינו מגדירים את הקופסא גדולה יותר כך שנוכל לראות את השינויים, אז שימוש בפקודה align-items בתוספת של חיבור ל: start, end, center יקבע את מיקום האלמנטים.

נדגיש כאן ב-align עובדת מלמעלה למטה ולא לצדדים.

**במספר 7** ברשימה עסקנו במרכז של האלמנטים בתוך הקופסא.

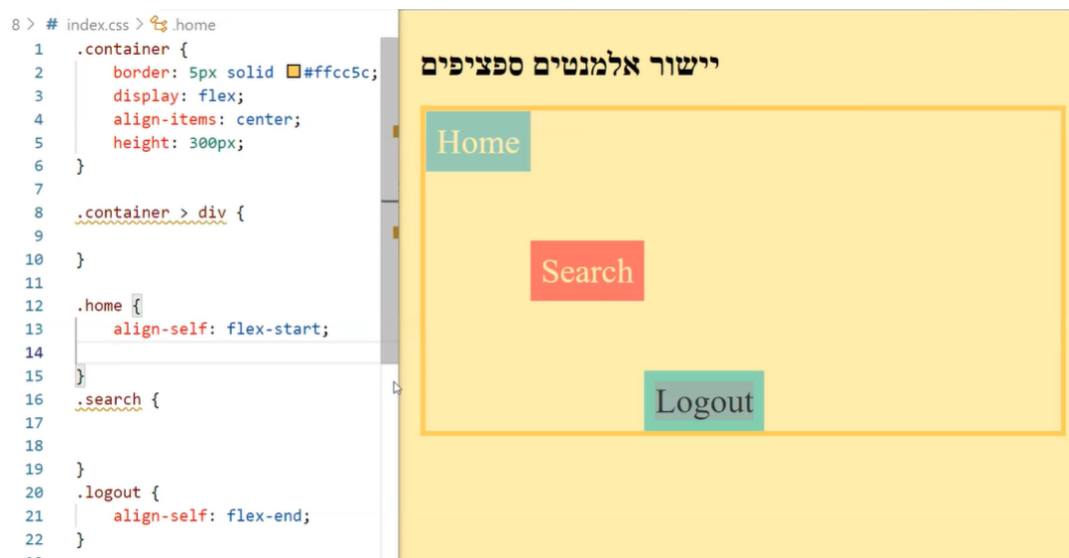
```
.container {
  border: 5px solid #ffcc5c;
  display: flex;
  height: 250px;
  width: 250px;
  align-items: center;
  justify-content: center;
}
```



**במספר 8** ברשימה ראינו אפשרות ליישור אלמנט ספציפי. כלומר, יישור ומשחק עם כל אחד מהאלמנטים ספציפיים.

שימוש בפקודת align בתצורות השונות על כל אחד מהאלמנטים תניח כל אחד במקום שבחרנו לו.

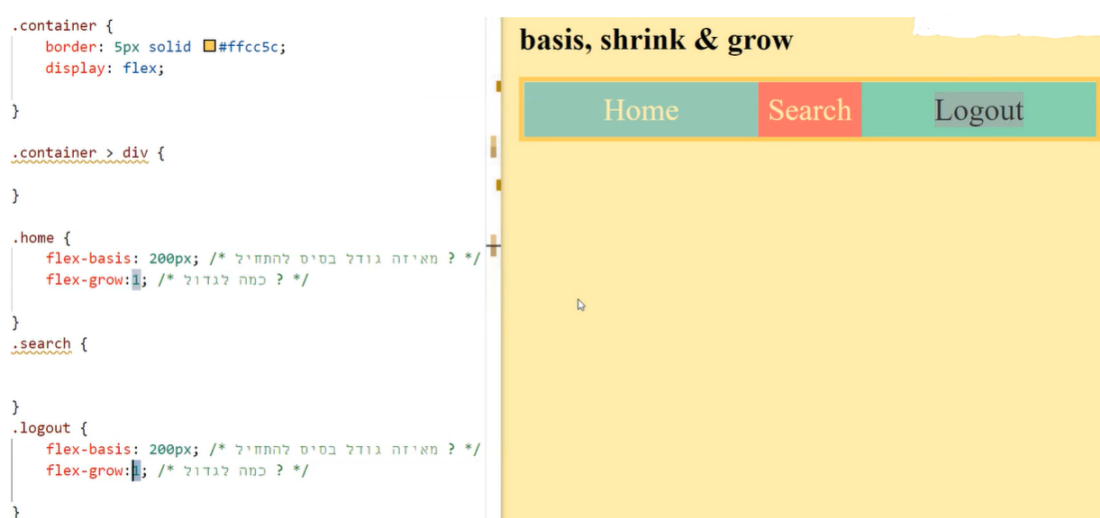
דוגמא בתמונה שבעמוד הבא:



**במספר 9** ברשימה דברנו על wrap, כאשר זה מגדיר את אפשרות ירידת השורה.

כאשר דבר זה לא מוגדר, כברירת מחדל אין ירידת שורה אלא צמצום אלמנטים.

**במספר 10** ברשימה ראינו אפשר של משחק עם הגדלה של אלמנטים מסוימים ואפשרות של גדילה במקרה הגדלת מסך. כלומר, הפקודה grow בעצם מגדירה את הגדילה במצב של הרחבת מסך והמספר מגדיר בכמה לחלק – אם נניח במקרה שבתמונה יש 1 ב-home ו-1 ב-logout אז הם יחלקו את הגדילה ביניהם חצי-חצי.



עוד דבר שראינו, הוא מאפיין בשם order שמסדר את האיברים בסדר שאותו נרצה לפי המספור שנוסיף. נניח אם home הוא הראשון ונגדיר אותו {order: 3} הוא יהפוך להיות השלישי.

## :SASS

מדובר כאן על ספרייה קטנה של javascript שמאפשרת עבודה נוח יותר עם CSS. כאשר אנחנו כותבים ב-SASS את העיצוב, אנו מפעילים את הקומפיילר הקטן שנמצא שם באמצעות לחיצה על watch שבתחתית העמוד ואז בעצם זה מקמפל את כל מה שאנחנו כותבים לתוך ה-CSS.

כמובן שאת קובץ ה-CSS קישרנו בצורה הרגילה לקובץ ה-HTML.

## שיעור 5

### נמשך עם SASS:

אז ראינו שאפשר ליצור משתנים שיחזיקו ערך (גם צבע הוא ערך), נכתוב אותם כאשר הסימן הפותח הוא '\$'.

בעקרון אחד מההבדלים המהותיים הוא שמשתני SASS מתקמפלים והופכים להיות חלק מהדף ונשארים עם אותו הערך בכל הדף. משתני CSS ניתנים לשינוי בתוך סקופ מסוים.

הערות ב-SASS ניתן להוסיף באמצעות '//' כמו ב-C#.

התכונה המרכזית שמהווה יתרון לשימוש ב-SASS היא קינון (nesting).

ב-CSS ראינו שיש אפשרות לסלקטור שמתנה שימוש ב-class מסוים על ידי שימוש באחד אחר.

תכונת הקינון ב-SASS מאפשרת לעשות זאת בצורה יותר נוח של קינון התכונות.

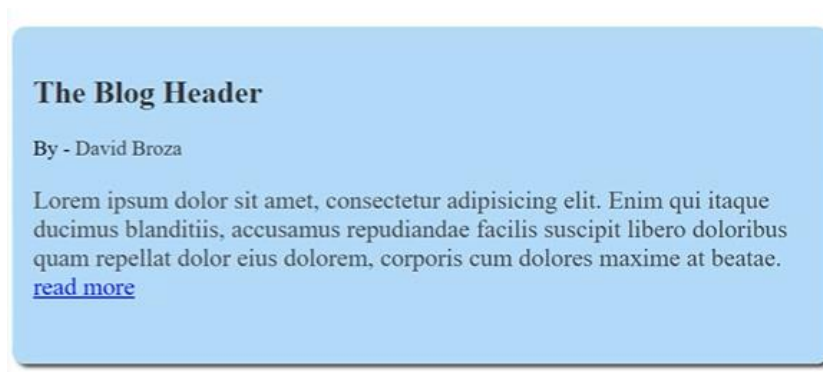
מה שייראה כך ב-SASS:

```
.posts {  
  
  background: rgba(172, 216, 245, 0.9);  
  padding: 1em;  
  border-radius: 10px;  
  box-shadow: 3px 3px 3px rgba(0, 0, 0, 0.7);  
  
  .blogs {  
    margin-bottom: 2em;  
  }  
  
  .header {  
    color: #333;  
  }  
}
```

ייראה בצורה הבאה ב-CSS:

```
.posts {  
  background: rgba(172, 216, 245, 0.9);  
  padding: 1em;  
  border-radius: 10px;  
  -webkit-box-shadow: 3px 3px 3px rgba(0, 0, 0, 0.7);  
  box-shadow: 3px 3px 3px rgba(0, 0, 0, 0.7);  
}  
  
.posts .blogs {  
  margin-bottom: 2em;  
}  
  
.posts .header {  
  color: #333;  
}
```

על הדרך, נראה איך ייראה העיצוב בזכות הפקודות של עיגול הפינות של הקופסא ויצירת הצל:



אפשרות נוספת, היא @extend:

האפשרות הזו בעצם מאפשרת החלה של עיצוב מסוים מאלמנט אחד על אלמנט אחר.

```
button {
  font-size: 1.3rem;
  font-family: serif;
  background-color: maroon;
  color: #fff;
  margin: 20px;
}
.cta {
  @extend button;
  border: 3px solid yellow;
}
```

במילים זה פשוט אומר – "קח את העיצוב של button ותעשה אותו גם על cta.

דבר מעט דומה ניתן לעשות באמצעות Mixin עם שינוי קל – ניתן להתייחס לעיצוב שניתן כפונקציה, כלומר, חלק מהדברים יעברו כפרמטרים ואוכל לשנות אותם בהתאם.

אז נניח שיצרנו שלושה ריבועים:

```
<div class="boxm1">Mixin box1</div>
<div class="boxm2">Mixin box2</div>
<div class="boxm3">Mixin box3</div>
```

וכך בעצם השתמשנו ב-Mixin:

```
@mixin box($width: 100px, $height: 100px, $bgc: aqua) {
  width: $width;
  height: $height;
  background-color: $bgc;
  border: 2px solid #000;
  margin: 20px;
  padding: 20px;
  font-family: Satisfy;
}
.boxm1 {
  @include box(200px, 100px, magenta);
}
```

```
.boxm2 {
  @include box();
}
.boxm3 {
  @include box(200px, 200px, silver);
}
```



אנו רואים שב-box נתנו שלושה משתנים עם ברירת מחדל שאותם בחרתי להיות כפרמטרים שאותם נוכל לשנות, ואם לא נשנה, הן יוכנסו לפי ברירת המחדל. (בתמונה הקטנה רואים את הפלט הציורי)

בחלק מהריבועים החלפנו את המשתנים בערכים אחרים באמצעות @include.

תכונה נוספת שמשתמשים המון היא import שהיא בעצם מאפשרת מעין שרשור של קבצים, או יותר נכון חלוקה של העיצוב לקבצים שונים.

נניח שכתבנו תגית לתחתית העמוד עם כתובת:

```
<footer>
  <p>Address: Eretz Utz</p>
  <p>Phone: 0505666999</p>
</footer>
```

והכתבנו לה עיצוב בקובץ SCSS משלה:

```
footer{
  border: 1px solid orange;
  background-color: yellowgreen;
  border-radius: 20px;
  box-shadow: 2px 3px 4px;
}
```

לקובץ קראנו "\_footer" (כן, כן, עם קו תחתון לפני)

בקובץ ה-SCSS שלנו נוסיף (בדרך כלל בראש העמוד) את השורה:

```
@import "footer";
```

נשים לב שזו פשוט מוסכמה שמוסיפים בשם של הקובץ הנוסף את הקו התחתון לפני השם וב-import כותבים בלי וכמובן שזה מתקבל.

יש עוד תכונות שאפשר להגיע עם SASS אבל עם עבודה מול המידע הקיים ברשת אפשרות להגיע אליהן בקלות.

עד כאן SASS.

## :Grid

הרעיון הכללי הוא שאני מחלק את הבלוק המסוים למעין רשת משבצות ובעצם עובדים איתה בכל מיני צורות, כך שכל האלמנטים הבנים הישירים (רק הישירים) מסתדרים בתוך המשבצות לפי איך שנגדיר אותם.

אז בדומה ל-FlexBox אנו יוצרים "container" שיכיל את הרשת.

כרגע נעבוד עם CSS רק לשם התרגול.

ניצור תגית style שבו נגדיר את ה-container עם העיצוב של הרשת

```

<head>
  <style>
    .grid-container {
      display: grid;
      grid-template-columns: auto auto auto;
      background-color: azure;
      padding: 10px;
    }

    .grid-item {
      background-color: whitesmoke;
      border: 1px solid blue;
      padding: 20px;
      font-size: 30px;
      text-align: center;
    }
  </style>
</head>

```

וכעת הקוד של ה-HTML:

```

<!-- זהו אלמנט האב -->
<!-- המיכל שמוגדר בתור גריד -->
<div class="grid-container">
  <!-- ואלו הילדים שלו -->
  <!-- המשבצות בגריד -->
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
  <div class="grid-item">7</div>
  <div class="grid-item">8</div>
  <div class="grid-item">9</div>
</div>

```

אם לא היינו מגדירים את ה-grid היינו מקבלים כאן פשוט 9 שורות, אבל על פניו כך זה ייראה בעקבות ה-grid:

1	2	3
4	5	6
7	8	9

כמובן שהחלק בעיצוב של grid-item של דואג לנראות היפה יותר, התצורה של הרשת הייתה נשמרת גם בלעדיו.

את המשך הדוגמאות ניתן לראות בסיכומים המעולים על Grid של אייל ברדוגו.

## שיעור 6

:Bootstrap

מהי bootstrap באופן כללי?

ספרייה קיימת של עיצובים שניתן פשוט לקחת ולהשתמש.

האתר שנבנה באמצעות הספרייה הזו יהיה באופן אוטומטי רספונסיבי.

ליד כל תבנית שנרצה להשתמש בה יש לנו אפשרות להעתיק את הקוד הרלוונטי ולהדביק אצלו בדף ה-HTML.

### Starter template

Be sure to have your pages set up with the latest design and development standards. That means using an HTML5 doctype and including a viewport meta tag for proper responsive behaviors. Put it all together and your pages should look like this:

```
<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Hello, world!</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
```

Copy

### Grid של Bootstrap

הגריד של בוטסטרפא מחלק את הרשת אוטומטית ל-12 עמודות.

כל אלמנט שנגדיר נוכל להגדיר שיתפוס מספר מסוים של עמודות, כלומר, באמצעות ה-class המובנה של בוטסטרפא שנקרא col-(num) כלומר יהיה שם מספר בין 1 ל-12 ואז בעצם זה מספר העמודות שהוא יקבל.

אם נגדיר חלק משורה מסוימת גם כן class="row" אז גם החלק הזה יאפשר חלוקה ל-12.

נדגיש שאין הרבה מה ללמוד על האפשרות של בוטסטרפא אלא ממש לשחק עם זה ולהתנסות וכך להכיר עם הזמן יותר טוב את הספרייה.

עוד כמה ספריות נפוצות:

<https://material.io/design>

[/https://bulma.io](https://bulma.io)

[/https://tailwindcss.com](https://tailwindcss.com)

[/https://get.foundation](https://get.foundation)

באחרונה יש התעסקות עם אימיילים ואם נידרש לכך שווה לזכור זאת.

## :JavaScript

זוהי שפת תסריט מפורשת שבפועל כותבים איתה תוכנות ענקיות וטובות (ללא שום קשר לשפה (Java).

השפה הזו רצה בדפדפן ולא בשרת וזה אומר שהלקוח הוא זה שיספוג את המאמץ של האתר (או יותר נכון רכיבי המחשב של הלקוח).

על מנת להוסיף קטע קוד של JS (כך אתייחס כאן ל-JavaScript), נוסיף תגית script בחלק של head- בעמוד ה-HTML. כמובן שזו צורה ספציפית, בהמשך נראה עוד אפשרויות.

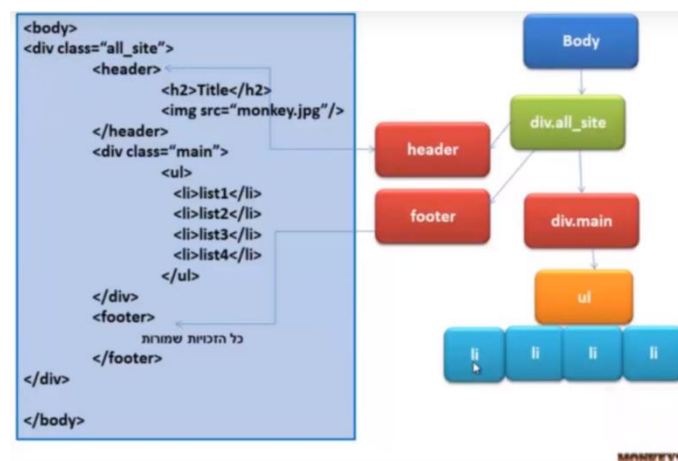
```
<script>
  console.log("hello JS");
  alert("Hello World");
  document.write("<h2>Hi Class!</h2>");
</script>
```

הפקודה console.log היא בעצם מעין הערה למקודד שנוכל לראות את מה שהיא שולחת בקוד של העמוד ולא בעמוד עצמו.

הפקודה alert מקפיצה חלוןית עם הודעה בראש העמוד.

הפקודה השלישית מוסיפה בקוד ה-HTML את הטקסט שהכנסנו, אפילו ברמה של להוסיף תגית של כותרת כמו שבדוגמא.

- הערה: אמנם בתקן החדש אין חובה להוסיף נקודה-פסיק בסוף שורה אבל נהוג בכל זאת כן לכתוב כך.
- Document Object Model – DOM הוא התבניתיות של העמוד מבחינת ענפים, נשים לב שבאמצעות JS אנחנו ממש שולטים בכל ה-DOM.



הדפדפן נטען לפי סדר התגיות, כדאי לזכור שהקוד של JS רץ אחרי שהדפדפן סיים להריץ את התגיות, וזה קורה משום שאם שלחתי פקודה לשנות משהו בעמוד אבל העמוד עוד לא רץ אז הפקודה תיפול.

ניתן לפתוח קובץ עם סיומת js על מנת לכתוב את הקודים בעמוד נפרד. וכדי לקשר את העמוד לעמוד שלנו נשתמש בפקודה:

```
<script src="./main.js"></script>
```



```
<script>
    document.getElementById("e1").innerText = "<strong>Gil</strong>";
    document.getElementById("e2").innerHTML = "<strong>Alkobi</strong>";
    document.getElementById("e2").style.color = "red";
    document.getElementById("e2").style.backgroundColor = "yellow";
</script>
```

בקוד הנ"ל אנו רואים אפשרויות לשינוי דברים באמצעות id שהגדרנו לאלמנט מסוים.

בשניים הראשון ראינו כתיבה לתוך הקובץ כאשר הראשון מוסיף טקסט פשוט ללא משמעות והשני מוסיף טקסט עם משמעות לתגיות שכתבנו בתוכו.

שתי השורות האחרות משנות עיצוב לאלמנט המוגדר.

```
<p id="e1">My First Name</p>
<p id="e2">My Last Name</p>
<p class="cls1">Lorem ipsum</p>
<p class="cls1">Lorem ipsum </p>
```

אלו האלמנטים שיושפעו.

## שיעור 7

ממשיכים עם JS.

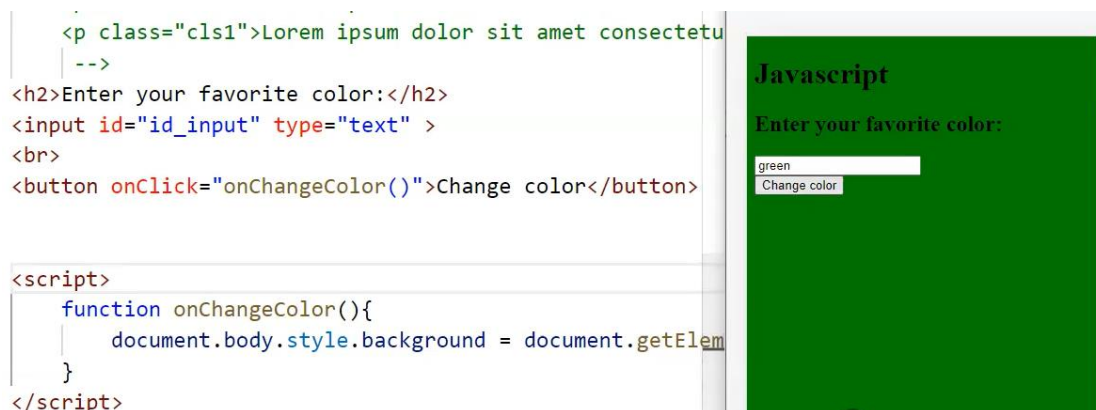
ראינו את הפקודות innerText ו-innerHTML שמאפשרות כתיבה מהחלק התכנותי לתוך העמוד. כאן המקום להדגיש שכך ניתן להכניס תוכן או להחליף תוכן בעמוד גם מבלי לטעון את כל העמוד מחדש (דומה להחלפה של userControl על גבי פאנל ב-winForm).

עוד דגש שנכון להזכיר:

■ אם אנו נותנים id ואלמנט מסוים, אנו חייבים לזכור (כי עורך התוכן לא יתריע) שהוא יכול להופיע רק פעם אחת. לעומת זאת class נוכל לשים כמה שנרצה. גם המתודות של JS פועלות בהתאם, נניח המתודה getElementById שבאופן אוטומטי יכולה להחזיר רק אלמנט אחד (כי יכול להיות רק אחד). פקודה נוספת שקיימת היא querySelectorAll שמקבלת שם של class ומחזירה מערך של האלמנטים שמממשים את ה-class הזה.

ראינו גם כן אפשרות להוסיף עיצוב לפקודת JS. מוסיפים נקודה ואז style ואז שוב נקודה ואז את הפקודה המתאימה לעיצוב אותו נרצה.

נניח שרצינו ליצור כפתור שמקבל צבע מהמשתמש ובהתאם לכך משנה את צבע הרקע:



את אותו הדבר אפשר לעשות עם selectBox שראינו כבר בעבר.

אם היינו מוסיפים אירוע מסוג oninput על תיבת הטקסט אז הוא יבדוק כל הקשה והקשה שלי, כלומר כל אות ואות שאוסיף בכתיבת הצבע וכמובן ברגע שאגיע בכתיבה לצבע קיים אז הוא ישתנה גם ללא לחיצה על הכפתור.

דבר נוסף, האירוע onchange בעצם יגיב לשינוי ב-selectBox גם מבלי שנלחץ על כפתור.

בשאר השיעור בעיקר בנינו עמוד המבוסס על החומר הקודם שלמדנו ב-HTML ו-CSS כאשר כל ההסברים נמצאים בקובץ ה-SCSS של העמוד.

טיפים:

כדאי להגדיר את העמוד כ-borderBox כי זה יעזור לנו להגדיר את ה-padding וה-border כחלק מהקופסא.

ראינו שאפשר לייבא פונטים של טקסט מ-google fonts.

## שיעור 8

בנינו עמוד שמייצג (כאילו מייצג) בנק, כאשר בתבנית ההתחלתית של bootstrap והוספנו לה שתי תיבות (גם כן מ-bootstrap).

כעת אנו רואים שימוש במתודות של JS שראינו בשיעור קודם.

```
<h5 class="card-title">Special title treatment</h5>
  <p class="card-text">Balance:
    <span id="balance">50</span>
    NIS
  </p>
```

נשים לב שהנחנו את ה-50 בכוונה בתוך תגית span כדי שנוכל לשנות רק אותו מבלי לשנות את שאר הכיתוב.

איך בעצם נשנה?

כאן המקום רגע להזכיר שה-HTML בונה את העמוד בהדרגה לפי סדר השורות הנמצאות בחלק ה-body ולכן את חלק ה-script אפשר לשים מתחת לכל הקוד שנמצא ב-body או שאם נרצה להשאיר אותו מחוץ ל-body נוכל להשתמש ב-onload.

השימוש ב-onload דורש מאיתנו לכתוב את הקוד שאנו רוצים לממש בתוך פונקציה בחלק של ה-script ולקרוא לו בתגית באמצעות onload.

```
<script>
  function init(){
    document.getElementById("balance").innerText = 1000;
  }
</script>
```

זו הדוגמה לכתיבת הפונקציה ברמה הבסיסית.

וכך תיראה הקריאה ב-body:

```
<body onload="init()">
```

כמובן שלפונקציה נוכל לקרוא בכל שם אחר ולא דווקא init.

## Zillion Money Bank

Special title treatment

Balance: 1000 NIS

כרגע כך נראה העמוד לאחר שהמספר 1000 החליף את המספר 50.

כאשר אנו יוצרים משתנה אנו פשוט מגדירים אותו באמצעות var ומאחורי הקלעים נוצר איבר שהסוג שלו נקבע לפי התוכן שהזנו לתוכו, שזה יתרון אבל כמובן גם חיסרון מאחר ו-HTML לא מקפלת ולא מודיעה על שגיאות עד שננסה להריץ את העמוד.

בנוסף, עוד חיסרון הוא שאם נבצע פעולות כמו חיבור זה יכול לצאת לנו לא טוב אם בטעות ננסה לחבר משתנה מסוג int ומשתנה מסוג string.

### ישנם סוגים שונים של משתנים (שנדבר עליהם בפרק ובפרקים הבאים) אך הבסיסים שבניהם (שמוגדרים כפרימיטיבים) הינם משתנים מסוג:

**STRING**: מחרוזת - טקסט המזוהה על ידי גרשיים "" ויכולה להכיל אות, מילה, משפט ואף טקסט של מאות ואלפי עמודים מבחינת מידע. בנוסף **STRING** יכול להיות גם מחרוזת ריקה.

**NUMBER**: מספר - יכול להיות גם מספר עשרוני, מספר שלם ואף מספר שלילי.

**BOOLEAN**: בדומה למתג שיכול להיות **ON** או **OFF** משתנה בוליאני יכול להיות **TRUE** או **FALSE** או 0 או 1 וישמש אותנו בהמשך כאשר נתעסק עם תנאים בקוד (ויחסוך לנו בכתיבת קוד לעומת שימוש ב- **STRING**)

ישנם עוד מספר משתנים מורכבים יותר (נדבר עליהם בהמשך בפירוט ואף חלקם מופיעים בספר ה-JS מתקדם) והם:

**ARRAY**: מערך (קבוצה של משתנים/קופסאות שנמצאים במשתנה/קופסא אחת)

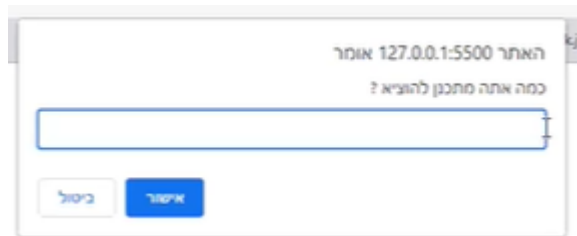
**Object**: אובייקט עם מאפיינים

פונקציה: פונקציה גם נחשבת סוג של משתנה בשפת JS

**REGEXP**: רגולר אקספרשן.

יש מתודה שנקראת `parseFloat` שממירה את המשתנה שהיא מקבלת מסוג `float` ל-`float`.

מתודה נוספת בשם `prompt` מאפשרת לנו להקפיץ תיבה שתבקש ערך מהמשתנה.



כמובן שלא באמת משתמשים בזה.

```
<script>
    var customerBalance = 75;
    var futureExpanse = parseFloat(prompt("20.5"));
    var amountThatWillRemain = customerBalance - futureExpanse;
    var msg = `Amount that will Remain: ${amountThatWillRemain}`;

    function init(){
        document.getElementById("balance").innerText = customerBalance;
        document.getElementById("futureExpanse").innerText = futureExpanse;
        document.getElementById("msg").innerText = msg;
    }
</script>
```

כאן אנו רואים שדרוג של הסקריפט הקודם באמצעות שימוש במשתנים והמתודות שראינו למעלה.

### ■ ניתן דגש כאן לגבי **Closure**:

המשתנים שכתובים בתוך סקופ מסוים, ניתנים לשימוש באותו הסקופ בלבד (ובסיום השימוש נאספים על ידי ה-Garbage Collector).

אבל, אם נגדיר משתנה בסקופ הגלובלי (בתוך תגית `script`) הם יוגדרו עבור כולם כולם כולל תגית `script` אחרת או אפילו ב-`script` מקושר חיצונית, כמובן כל עוד כולם קשורים לאותו עמוד `HTML`.

לסיכום: כל משתנה מוכר בתוך הסקופ שלו (לשימוש בפונקציות וכדומה).

**:Hoisting**

כאשר שפת התכנות מפענחת את הסקריפט (שאינו מתקמפל כזכור) ומתחילה עבודה, היא קודם כל אוספת אליה את כלל המשתנים הקיימים בסקופ של ה-script.

כאן המקום להדגיש שהיא אוספת את ההצהרות ולא את ההשמות. כך שלרוב מדובר בצורך שהצהרתי על משתנה ללא השמה והזנתי ערך בתוך אחת הפונקציות שבסקריפט ואז בעצם ה-hoisting יהיה יעיל מבחינתי.

**סוגי משתנים:**

בצד הגלובלי נשתמש ב-var.

בתוך פונקציות נשתמש ב-let.

נגדיר קבוע באמצעות const.

**יש מונחים שנקראים truthy ו-falsy:**

כל מה שנתפס כתקין כלומר משתנה שמוזן בו ערך כלשהו יוגדר כ-truthy, וכל מה שנתפס שלא רגיל ללא תקין כמו משתנה ריק הוא falsy.

תנאים ב-JS עובדים כמו שאנחנו מכירים מ-C#. if, else if, else.

נשים לב להבדלים לוגיים:

הסימן '==' משווה רק תוכן ללא התחשבות בסוג המשתנה כך שמבחינתו "1" ו-1 זה אותו הדבר (נשים לב לגרשיים).

כדי להשוות גם את הסוג, נשתמש בסימן '==='. בסימן הזה כמעט תמיד נשתמש.

אם נרצה לבדוק מהו סוג של משתנה מסוים, נשתמש בפקודה typeof().

אז נמשיך עם הקוד שהתחלנו קודם, יצרנו כפתור בקופסא הקיימת שיוביל אותנו לקופסא השנייה:

```
<button class="btn btn-primary" onclick="showBox2()">Show Other Box</button>
```

המחלקות של העיצוב שאני משתמש כאן הם מתוך bootstrap (זה צבע כחול).

```
<div class="col-sm-6" id="box2" style="display: none;">
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">How much do you plan to expanse?</h5>
      <p class="card-text">
        <label for="exampleFormControlInput1" class="form-label">Type a number...</label>
        <input type="number" class="form-control" id="fe" placeholder="123...">
      <br>
      You plan to expanse :
      <span id="futureExpanse"></span>
      <br>
      <span id="msg"></span>
    </div>
  </div>
</div>
```

```

    </p>
  </p>
  <button onclick="calc()" class="btn btn-primary">Calc</button>
  <button onclick="getMoney()" class="btn btn-primary">OK</button>
</div>
</div>
</div>

```

כאן אנו רואים יצירה של הקופסא השנייה.

```

function showBox2() {
  var el = document.getElementById("box2");
  el.style.display = 'block';
}

```

וכאן זו ההגדרה של הפונקציה שתחשוף את הקופסא השנייה.

## ZillionMoney Bank

כך זה יראה בהרצה לאחר הלחיצה על הכפתור "Show Other Box".

ואלו הפונקציות calc ו-getMoney שאנו רואים בקוד:

```

function calc() {
  var el = document.getElementById("fe")
  var futureExpanse = parseFloat(el.value);
  amountThatWillRemain = (customerBalance - futureExpanse);
  document.getElementById("futureExpanse").innerText = futureExpanse;
  document.getElementById("msg").innerText = getMsg(amountThatWillRemain);
}

function getMoney() {
  customerBalance = amountThatWillRemain;
  init();
}

```

וכך זה נראה בפעולה עם החישוב:

## ZillionMoney Bank

**Your Status**

Balance: 75 NIS

Show Other Box

**How much do you plan to expanse?**

Type a number...

5

You plan to expanse : 5

Amount that will remain: 70

Calc OK

ראינו שבשורה הראשונה של הקוד של הקופסא השנייה כתבנו 'display: none' אבל נשים לב שהפונקציה שיצרה אותו הגדירה אותו כבלוק כך שה-CSS הוגדר חיצונית בעת היצירה.

בנוסף נשים לב שבתוך המתודה calc אנו מגדירים את ה"השתלטות" באמצעות innerText ומוסיפים את הכיתוב שמפרט את החישוב.

## לולאות ומערכים:

נניח שאנו רוצים לקחת רשימה מתוך מערך.

```
var ex = [
    "ביגוד 10",
    "מזון 6",
    "חשמל 8"];

var html = "";
for (var i = 0; i < ex.length; i++) {
    let element = ex[i];
    console.log(element);
    html += `<li>${element}</li>`;
}
```

אז אנו רואים שיש לנו מערך שמכיל שלושה איברים, כאשר במערך ב-JS כל איבר הוא אובייקט.

לאחר מכן הגדרנו לולאה שרצה על המערך ובכל איטרציה מוסיפה איבר לרשימה.

■ משתנה const ב-JS בתוך הסקופ יכול להגדיר את עצמו מחדש.

כמובן שבמצב האמיתי מדובר על איסוף מידע מתוך מסד נתונים ולא מתוך מערך פשוט.

עכשיו, מכיוון ש-HTML עובד בצורה מדורגת, אנחנו רוצים להצליח להכניס את הפלט שלנו לעמוד, לכן אפשר להשתמש כמובן באפשרות שראינו עם הפונקציה init שפועלת באמצעות onload.

דרך נוספת, יצירתית אך קצת מיותרת ולא עושים זאת בדרך כלל, היא שימוש במתודה setTimeout

```
setTimeout(() => {
    document.getElementById("myList").innerHTML = html;
}, 1000);
```

מה היא עושה? היא אומרת למערכת לחכות זמן מסוים (1000ms שזה שנייה אחת במקרה שלנו) ורק אז להריץ את הפונקציה (או אפילו אנונימית עם למבדא כמו שהגדרנו) שהוכנסה.

מתודות נוספות למערכים:

`push()` – מוסיפה את האיבר שהוכנס לסוף במערך.

`unshift()` – מוסיפה איבר בתחילת המערך ודוחף את השאר אחד קדימה.

`pop()` – מוחק את התא האחרון במערך ומחזיר את הערך שלו.

`shift()` – מוחק את התא הראשון במערך ומושך אחורה את שאר התאים.

`indexOf()` – מקבל ערך של איבר ומחזיר את המיקום שלו במערך.

`splice(4,1)` – ימחק תא מסוים, נניח אצלנו את תא 4 ורק אותו כי כתוב אחד לאחר מכן. ומחזיר מערך של האיברים שנמחקו

`length()` – מחזיר את כמות התאים במערך, נזכור שהספירה מתחילה מאפס כך שהאורך הוא תמיד אחד יותר ממספר המיקום האחרון.

כל המתודות צמודות עם נקודה לשם של מערך כלשהו.

כאן גם המקום להגיד שהלולאה בנויה ממש בדיוק כמו בסי שארפ למעט עוד סוגים של לולאות שנקראות `for...in` ו-`for...of`.

ההבדל בין לולאות אלו הוא:

`for... in` זה אומר שאנחנו רצים על ה-`keys` של האובייקט – זה לא כמו `for each` של סי שארפ.

לעומת זאת `for...of` כן עובד כמו `for each` ומאפשר ליצור לולאה שרצה עם האובייקט עצמו (נניח על איברי המערך).



## שיעור 9

נקודה קטנה לפני שמתחילים:

### CDN – Content Delivery Network

שירות שהוא בד"כ מסחרי אבל יש גם מודלים בחינם, שבעצם הוא מאפשר הגשה מהירה של קבצי JS, HTML, CSS ואפילו תמונות, כל מיני קבצים סטטיים שיאוחסנו בשרת של ה-CDN ואז בעצם כך השרת יודע להביא אותם מהר יותר מאשר אצלנו.

משתמשים בזה משתי סיבות, אחת היא המהירות והשנייה היא שבדרך כלל מוכרים אותם עם שכבת אבטחה כלשהי, שנניח שכל מי שפונה לאתר שלי עובר דרך ה-CDN ורק אם ה-CDN לא יודע להגיש משהו זה יגיע אליי, אז אם ישנה מתקפה כלשהי על האתר – יש לי בעצם למי לפנות ונותני השירות יכולים לחסום לי את התוקף.

אחת הדוגמאות היא מתבניות של BOOTSTRAP:

```
<!-- Bootstrap CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-1BmE4kWBq78iYhF1dvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```

לרוב לא נוהגים לקשור ישירות ל-CDN כי בדרך כלל אומרים שאם נקשור ישירות והוא ייפול אז אני מעדיף שהקובץ יישאר אצלי אבל הדעה השנייה היא שכנראה יש סיכוי שאני אפול לפני חברה גדולה עם הרבה יותר משאבים.

נתחיל בשיעור:

## Classroom

Course : English

Started At : 10 / 11

Teacher : JohnDow

Average Score : 63

## Homeworks

Search

userId	id	title	completed	
1	1	delectus aut autem	false	next week!
1	2	quis ut nam facilis et officia qui	false	tomorrow !!!
1	3	fugiat veniam minus	false	next week!
1	4	et porro tempora	true	tomorrow !!!

אנו נרצה ליצור את העמוד הבא, כך שבתור התחלה ניצור מצב שבו תיאורטית החלק עם הקווים הירוקים והאדומים יגיע משרת.

מה שמסומן בירוק יהיה החלק של ה-HTML שיופיע בעמוד והפסים האדומים זה המידע שכביכול יימשך מהשרת.

אז כמו שכבר ראינו בשיעורים קודמים אנו מתחילים את העמוד עם תבנית של BOOTSTRAP ואני פותח את ה-div הראשון עם container, לאחר מכן row ואז col-12, וכאן נכיל את הכותרת.

ניצור שוב row בתוך ה-container הקיים כמו בכותרת ונוסיף בתגיות <p> לחלקים עם הקו הירוק ולכל פסקה יהיה id או class על מנת שנוכל לפנות אליו אחר להזנת מידע באמצעות JS.

```
<div class="container">
  <div class="row">
    <div class="col-12">
      <h1>Classroom</h1>
      <hr>

    </div>
  </div>
  <div class="row">
    <div class="col-12">
      <p>Course : <span id="course"></span></p>
      <p>Started At : <span id="startedAt"></span></p>
      <p>Teacher : <span id="teacher"></span></p>
      <p class="score">Average Score : </p>
    </div>
  </div>
</div>
```

כאשר אני יוצר משתנה ומגדיר אותו עם סוגריים מסולסלים, אני בעצם יוצר אובייקט וכאן בשונה מסי שארפ הוא יכול להכיל מה שנרצה, אפילו יכול להיות שאחד האיברים הוא אובייקט בעצמו.

ניצור את המודל של הנתונים:

```
<script>
  var classDetails = {
    course: 'English',
    teacher: 'JohnDow',
    startedAt: new Date(2021, 10, 11),
    finishedAt: new Date("2022-02-22"),
    avgScore: Math.round(Math.random() * 100),
    tags: ['Hebrew Speakers', 'evening school', 'Tel-Aviv Branch']
  };
</script>
```

כעת ניצור מתודה בשם init():

```
function init() {
  document.getElementById("course").innerText = classDetails.course;
  document.getElementById("teacher").innerText = classDetails["teacher"];
  document.getElementById("startedAt").innerText =
    `${classDetails.startedAt.getMonth()} / ${classDetails.startedAt.getDate()}`;
  //===== average score =====
  let spanEl = document.createElement('span');
  spanEl.innerText = classDetails.avgScore;
  spanEl.style.border = '1px solid red';
  document.getElementsByClassName("score")[0].appendChild(spanEl);
}
```

נבין כעת את הכתוב בפונקציה.

למתודה `init()` נקרא באמצעות `onload` בתגית של ה-`body`.

בשורה הראשונה אנו רואים כתיבה של שם הקורס באמצעות קריאה לשם מתוך האובייקט.

בשורה השנייה אנו רואים כתיבה של שם המורה באמצעות קריאה בשם של אחד החלקים מהאובייקט, שיטה זו פחות נכונה לשימוש עקב כך שכאשר מישהו אחר יקרא את הקוד שלי, סיכוי נמוך עד כדי אין סיכוי שהוא יבין במי או במה מדובר ולכן זה יכול להקשות או להפוך את המשך העבודה עם הקוד לבעייתי.

בשורה השלישית אנו פוגשים משתנה מסוג `date` (כבר פגשנו אותו בהגדרת האובייקט אבל בסדר..)

באובייקט ראינו משתנה כזה בשני פורמטים שונים. ב-JS כל המשתנים של `date` נשמרים מאחורי הקלעים כ-`EpochTime` שזה בעצם מספר השניות מאז 1.1.1970 שזה ההקמה של Unix.

ספרייה נוספת ל-`date` שהשתמשנו בה היא `Math`, באמצעותה הגדרנו ממוצע.

המתודה `random` מגרילה מספר שבין אפס לאחת ולכן אם נרצה שיהיה מספר בין אחת למאה פשוט נכפיל ב-100 כמו שעשינו.

בסוף כדי לקבל ממוצע עגול השתמשנו במתודה `round` שמעגלת את המספר בהתאם.

נמשיך עם הקוד, ראינו שבמשתנה `Average score` הזנו `class` ולא `id`, ולכן פנינו אל האלמנט הזה באמצעות `getElementsByClassName`.

בנוסף לא השארנו `<span>` שנוכל להזין לתוכה טקסט כלשהו.

וכאן מגיע בעצם התפקיד של השורות הבאות בתוך הפונקציה שבאמצעותן אנו יוצרים אלמנט חדש (`createElement`), שכאן כמובן יצרנו תגית `span` שעוד לא קיימת במסך עדיין.

בשורה שלאחר מכן (שביעית בתוך הפונקציה) הזנו טקסט לתוך התגית שיצרנו ובשורה לאחר מכן יצרנו עיצוב שהיה לטובת השיעור בלבד.

כעת כדי להיכנס את התגית למסך, השתמשנו במתודה `appendChild`.

בחלק הבא יצרנו מערך של אובייקטים, כאשר כל אובייקט מכיל שיעורי בית, כלומר, מטלה ספציפית להגשה.

```
var homeworks = [
  {
    userId: 1,
    id: 1,
    title: "delectus aut autem",
    completed: false,
  },
  {
    userId: 1,
    id: 2,
    title: "quis ut nam facilis et officia qui",
    completed: false,
  }
];
```

בתרגיל עצמו הוספנו עוד אובייקטים עד `id: 7`.

ובחלק של ה-`body` הוספנו כעת גם טבלה שלתוכה נזין את המידע שנרצה עם שיעורי הבית.

נזכיר שנוכל להחליט מתי לעשות `th` לכותרת בטבלה או `td` למילוי תא רגיל וכמובן גם כאן נכניס את הטבלה לתוך `row`.

```

<div class="row">
  <div class="col-12">
    <h1>Homeworks</h1>
    <div class="col-4">
      <label for="search" class="form-label">Search</label>
      <input type="text" class="form-control" id="search">
    </div>
    <table class="table table-striped">
      <thead>
        <tr id="tableHeader"></tr>
      </thead>
      <tbody id="tableBody">

      </tbody>
    </table>
  </div>
</div>

```

בנוסף הגדרנו פונקציה חדשה:

```

function setHomeworkTitles(){
  let someObject = homeworks[0];

  // extract the element that contain the title
  let headerRow = document.getElementById('tableHeader');

  // extract the keys of the properties
  let keys = Object.keys(someObject);

  // run the key with loop and create elements of the cells in table
  // add the cells to title row
  for(let i = 0; i < keys.length; i++){
    let th = document.createElement('th');
    th.innerText = keys[i];
    th.scope = 'col';
    headerRow.appendChild(th);
  }
}

```

אני לוקח את אחד האובייקטים ממערך שיעורי הבית, מחלץ את ה-`keys` (החלק שלפני הנקודתיים – האובייקט מסודר בצורה של `key: value`), שומר אותם במערך באמצעות המתודה `object.keys` ורץ עליהם בלולאה כך שבכל איטרציה אני יוצר אלמנט חדש מסוג תגית `th` בטבלה, מכניס לתוכה טקסט שזה האיבר מהמערך וכמובן מוסיף אותו לעמוד.

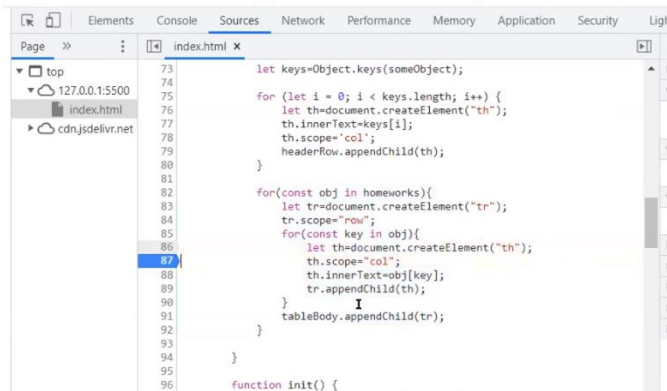
אם הייתי רוצה להשתמש בלולאה של `for...in`:

```

for (const key in someObject) {
  console.log('key', key);
  console.log('value', someObject[key]);
}

```

כדי שנוכל לדבג (Debug) את הקוד שלנו נוכל לפתוח F12 בעמוד שלנו ולהגיע לאפשרות של Sources, שם נוכל לסמן שורות ב-breakpoints:



נלחץ F5 והקוד יעצור במקום שסימנו.

ואם נתקדם קדימה עם F10 נוכל לעבור שורה שורה עד שנגיע לתקלה.

נמשיך, כעת אנו רוצים להמשיך ולמלא את הטבלה:

```
function fillHomeworks(rows){
    let tableBody = document.getElementById('tableBody');
    tableBody.innerHTML = ""; // איפוס של תוכן הטבלה
    var manipulatedData = rows.map(function(homework){
        if (homework.id % 2 === 0){
            homework.dueDate = 'tomorrow !!!'
        }else{
            homework.dueDate = 'next week!'
        }
        return homework;
    });
    //first loop - run on the manipulated rows
    for (let i = 0; i < manipulatedData.length; i++) {

        //Line in table
        let tr = document.createElement('tr');

        // homeworks
        let homework = manipulatedData[i];

        // second loop - run on keys in object
        for (const key in homework) {
            let td = document.createElement('td');
            td.innerText = homework[key];
            tr.appendChild(td);
        }
        // add line
        tableBody.appendChild(tr);
    }
}
```

בשורה הראשונה של הפונקציה יצרנו את האלמנט של הטבלה.

בשורה השנייה אנו מאפסים את תוכן הטבלה (במציאות התוכן אמור להשתנות כל הזמן בהתאם למידע שיוזרם אליה).

לאחר מכן אנחנו יוצרים מעין מניפולציה על הארגומנט הראשי rows ומשתמשים בפונקציה map שמקבלת פונקציה אחרת כארגומנט.

הפונקציה הפנימית שבתוך ה-map, מקבלת ארגומנט בשם homework שהוא בעצם כל פעם איבר אחר ב-rows (בתרגיל שלנו הערך שיוזן ל-rows יהיה המערך homeworks, ובתוך הפונקציה הגדרנו תאריך יעד להגשה (זה בעצם יוצר key: value חדש) שתלוי בחישוב סתמי שעשינו רק לצורך התרגיל – הפונקציה מחזירה את הערך homework לאחר שהוזן).

נדגיש, ש-map מאחורי הקלעים רצה בלולאה על המערך שהיא עובדת עליו וכל איטראציה מבצעת את הפעולה על האיבר הנוכחי, כך שהערך שהוחזר בסוף היה המערך לאחר המניפולציה.

כמובן שיכולתי לכתוב פונקציה חיצונית ולקרוא לה בתוך ה-map (כלומר, לא הייתי חייב לכתוב את כולה בתוך הסוגריים).

■ דגש: אם אחד האיברים שבאיבר במערך (כל איבר הוא אובייקט) היה אובייקט בעצמו אז הוא היה עובר את ה-map byReference.

גם כאן המשכנו את הפונקציה עם לולאה לצורך הזנת הנתונים בטבלה, אלא שהפעם הלולאה מקוננת משום שאנו רצים על שורות ועל עמודות, כאשר לולאה ראשית רצה על שורות ולולאה משנית רצה על עמודות.

לאחר שהגדרנו את מילוי הטבלה, בנינו פונקציה שבעצם מפעילה אירוע לחיפוש:

```
function addEventToSearch(){
    let searchBox = document.getElementById('search');
    searchBox.addEventListener('keyup', function(e){

        // the value we search
        let searchWord = e.target.value;

        //the value we want
        let matchedRows = homeworks.filter(function(row){
            var isMatched = row.title.includes(searchWord);
            return isMatched;
        });
        fillHomeworks(matchedRows);
    });
};
```

מה שבעצם קורה כאן זה שאנו שומרים במשתנה את מה שכתוב בתיבת הטקסט בכל פעם שהפסקתי ללחוץ על כפתור במקלדת (אל תיבת הטקסט הגענו באמצעות ה-id שלה כמובן).

לאחר מכן אנו בודקים אם יש שורות שמתאימות לסינון שרצינו באמצעות ריצה עם פונקציה filter (שבודקת לפי המערך homeworks) על כל השורות בטבלה ומחזירה לי מערך של כל התואמים, מכאן רק נותר להפעיל שוב את הפונקציה שתפקידה למלא את הטבלה.

כאן המקום כמובן להדגיש שלשתי הפונקציות שלנו של מילוי הכותרות ומילוי הערכים קראנו בסיום הפונקציה .init()

ראינו בסוף השיעור עוד פונקציה אחת שנקראת findIndex, שמוצאת את האינדקס של הערך המוזן.

## שיעור 10

ראינו את האפשרות ללולאת do-while וגם את התנאים באמצעות switch-case, עובד בעיקרון כמו בסי שארפ ללא משהו מיוחד ושונה.

```
<script>
    var i = 0;
    do {
        console.log("i = " + i);
        i++;

        switch (i) {
            case 2:
                console.log("2 🐼");
                break;
            case 3:
                console.log("3 🐼");
                break;
            default:
                break;
        }
    } while (i < 5);
</script>
```

נזכיר שבלולאה כזו קודם כל הקוד מתבצע פעם אחת ואז התנאי נבדק לראשונה.

ראינו כאן גם דבר מגניב והוא שילוב של אימוג'ים בקוד, זה כמובן נתמך משום שהקוד מוגדר תחת UTF-8.

בשיעור הקודם ראינו את הפונקציה map, וזו פונקציה חשובה מאוד ומשתמשים בה המון.

בשימוש שלה היא דומה ל-select בשאילתות.

נזכיר שהפונקציה map מקבלת פונקציה כארגומנט ובעצם רצה בלולאה על איברי המערך המוצמד אליה ומבצעת על כל אחד מהם את הפעולה שפונקציית הארגומנט מבצעת. בסופו של דבר מחזירה לנו מערך לאחר הפעולות.

נדגיש שהמערך שהוחזר הוא מערך חדש לגמרי ואין reference בין המערכים – כלומר, לא נוצר רק מצביע.

מה שזה אומר זה שהמערך המקורי לא השתנה! וזה קורה כאשר המשתנים פרימיטיביים.

```
let ary = [1,2,3,4,5];
let newArray = ary.map((i) => i*2);
// c# way: ary.Select(i => i*2).ToList()
// newArray = [2,4,6,8,10]
```

כאשר המשתנים שבמערך הם לא פרימיטיביים אלא אובייקטים כלשהם, המצב יהיה קצת שונה משום שאז כ-reference type גם המערך המקורי ישתנה.

```
var objectAry = [
    {i:1, },
    {i:2, },
    {i:3, },
    {i:4, },
    {i:5, },
];

let newObjectAry = objectAry
    .map((obj) => {
        obj.i = obj.i * 2;
        return obj;
    });
```

נשים לב שבמצב הזה המעניין הוא שהפונקציה map יוצרת מערך חדש אבל האיברים שבו הם by reference (במקרה שלנו זה אומר בעצם שנוצר מערך חדש אבל בתוכו חמישה מצביעים לחמשת האיברים הקיימים במערך המקורי).

דגש נוסף וחשוב הוא שאם בתוך אחד האובייקטים (איבר במערך) היה עוד אובייקט אז הייתי צריך לעשות גם לו העתקה מסודרת אחרת אמנם האיבר היה חדש אבל האובייקט שבתוכו היה נשאר reference.

```
{i:1, some: {z:4} }
```

זו דוגמא לאחד האיברים במערך objectAry.

אם היינו רוצים להעתיק את המערך ממש שגם האיברים (אובייקטים, לא פרימיטיביים) יועתקו לא בצורה של reference, נעשה זאת בצורה הבאה:

```
let newObjectAry = objectAry
    .map((obj) =>{
        let original = obj.i;
        return{i: original};
    });
```

פונקציה נוספת היא reduce, שבבסיסה סוכמת את איברי המערך, העניין הוא שאפשר בתוכה להוסיף לפעולה דברים נוספים כמו חישוב ממוצע.

גם כאן דרך דומה בסי שארפ היא aggregate.

```
var numbers = [1,2,3,4];
//I-1: total = 1 | currentValue = 2
//I-1: total = 3 | currentValue = 3
//I-1: total = 6 | currentValue = 4
// sum = 10
var sum = numbers.reduce(function(total, currentValue){
    console.log("total is: ", total, "value is: ", currentValue);
    return total + currentValue;
});

// c# way: ary.Aggregate(Func<T>)
console.log("sum = ", sum);
```



**דבר נוסף לגבי אובייקטים:** פונקציות בJS הם אזרחים מדרגה ראשונה.

כל פונקציה היא אובייקט וכל אובייקט הוא פונקציה ואפשר להכניס את הפונקציה כאובייקט במערך כמו כל איבר אחר.

```
let obj = {  
  a: 1,  
  b: 2,  
  c: function () {  
    console.log("Hi from obj");  
  }  
}  
obj.c();
```

כאן אנו רואים שיצרנו אובייקט עם איברים a, b, c.

את האיברים אנו יכולים לשנות מחוץ למערך וכמובן גם להוסיף חדשים בצורה יחסית פשוטה:

```
obj.a = 3;  
obj.b = 5;  
console.log(obj);  
obj.otherFunction = () => console.log("Putin is the king!");
```

## המונח 'this':

זהו מושג שכרגע בשיעור כל מה שנלמד עליו מאוד דומה למה שאנו מכירים מסי שארפ אבל בעתיד נלמד על מצבים בהם גם יש שוני.

```
var user = {  
  name: 'Gil',  
  email: 'gilgil@gmail.com',  
  getAge: () => '31',  
  
  fn: function () {  
    return 'Your name is: ' + this.name + ' and your age is: ' + this.getAge()  
  }  
};  
  
// output Your name is: Gil  
console.log(user.fn());
```

אנו רואים שה-`this` הוא האובייקט הנוכחי.

מה ששונה ונלמד עליו בהמשך הוא שב-JS נוכל לקבוע מחדש מי יהיה ה-`this` הזה.

כמובן שאם נשפיע מחוץ למערך ונשנה את האיברים `by reference` זה ישנה את התגובה של הפונקציה מבפנים (את `fn`).

## Setter – Getter:

אנו יכולים ליצור את האובייקט עם פונקציות שמזינות את הערכים בהתאם למה שנזין.

דבר זה דומה מאוד להיכרות עם מאפיין במחלקה בסי שארפ.

כמובן שזו צורה קצת פחות רגילה למימוש אבל עדיין.

```
var product = {
  id: 15,
  price: 25.5,
  title: 'My title',
  article: 'My article',
  set: function (key, value) {
    this[key] = value;
  },
  get: function (key) {
    return this[key];
  }
};

product.set('title', 'Other title');
product.set('price', 45);
product.set('newKey', 'Osama');
```

### שרשור:

נוכל לשרשר מתודות כמו שאנו מכירים מ-LINQ בסי שארפ, כל עוד המתודה הקודמת בקודמת בשרשור מחזירה ערך.

```
var mailer = {
  email: null,
  setMail: function(mail){
    this.email = mail;
    return this;
  },
  getMail: function(){
    console.log( this.email );
    return this;
  },
  hello: () => {
    console.log('hello')
  }
};

// output foo@gmail.com
mailer
  .setMail('foo@gmail.com')
  .getMail()
  .hello();
```

### מחלקות:

Syntactic Sugar – צורה מסוימת בשפה שכאילו מקלה עלינו בתכנות אבל מאחורי הקלעים מה שנכתב יותר מסובך להבנה.

הביטוי הוא שבעצם השפה היא כמו סוכר סינטטי שזה לא אמיתי אלא רק מעטפת.

אנחנו נראה כרגע את מה שקורה מאחורי הקלעים אבל בגרסאות המתקדמות יותר כמו ES6 והלאה זה לא כך.

אמרנו קודם שפונקציה היא אובייקט, ועכשיו נראה זאת.

בצורה הישנה של ES5 הגדירו שניתן לבנות פונקציה ובתוכה איברים (משום שהיא אובייקט) שיכולים להיות גם פונקציות בפני עצמם.

כך שבעצם אנו בונים פונקציה ובתוכה מגדירים איברים ופעולות ומחוץ לפונקציה אנו יוצרים מופע של הפונקציה, כך שעל פניו חוץ מהמילה function זה נראה כמעט אותו דבר כמו מחלקות שאנו מכירים.

```
function Gallery(city) {
    this.id = 6;
    this.name = 'ZeMapel';
    this.city = city;

    this.fn = function () {
        console.log(this.name + ' is presented in ' + this.city);
    }
}
var ob = new Gallery("Paris");
var ob1 = new Gallery("London");
```

אנו רואים שהפונקציה כאן מקבלת ארגומנט ושלוש השורות הראשונות שלנו משמות כבנאי של הפונקציה.

■ מכיוון שפונקציות הן אובייקט ניתן להגדיר אותן בצורה של `var Gallery = function(city)`

נשים לב שאמרנו שנוכל להוסיף keys חדשים באמצעות קריאה לאחד שלא קיים כרגע, לכן, אם נוסיף לאחד המופעים מפתח חדש (לדוגמא `ob.startedDay = "Sunday"`) והוא יתווסף רק לאותו המופע ולא למחלקה בכללי.

ראינו בשיעורים קודמים את הלולאה `for...in`, כעת נוכל לראות שימוש שלה.

נניח שאנו בונים פונקציה דומה ל-Gallery, ורוצים לרוץ על המפתחות שלה זה יראה כך:

```
var Person = function () {
    this.name = "Israel Cohen";
    this.age = 35;

    this.m = function () {
        return true;
    };
};

var obj = new Person();

for (const x in obj) {
    if (typeof obj[x] !== "function") {
        console.log(x + ": " + obj[x]);
    }
}
```

כאן בדוגמא רצים על המפתחות ובודקים אם המפתח הוא מסוג שאינו פונקציה ומדפיס את סוג המפתח.

**:Prototypes**

לכל אובייקט יש גם אב-טיפוס שגם הוא אובייקט ואובייקטים הם דינמיים.

אז נניח שלקחנו את string שהוא אובייקט מובנה, אנו יכולים לבנות מתודה חדשה שנוכל להפעיל אותה על כל string שנכתוב:

```
String.prototype.log = function () {
    console.log("Value: " + this.toString());
};

// output Value: Hello World
"Hello World".log();
// output Value: Hello World
"Hi Whats up".log();
```

JS מחפש את המתודה שקראנו לה ברמה הבסיסית של האובייקט ואם הוא לא מוצא הוא ממשיך לחפש באב טיפוס הבא, וכך ממשיך עד שמגיע לאב טיפוס הכי קדמון (null).

נשים לב שניתן לעשות את כל זה גם על אובייקט שאינו מובנה, כלומר, אובייקט שאנו בנינו אותו בעצמנו.

```
function Validator() {

};

Validator.prototype.submit = true;

Validator.prototype.require = function () {
    console.log('require work!');
};

var check = new Validator();

check.require();
```

רואים כאן submit שמכניס ערך חדש, ו-require שמכילה פונקציה אנונימית כלשהי.

אם היינו משנים מחדש את submit זה ישנה אחורנית גם את השינוי הקודם.

**עוד דגש חשוב:**

אנו יכולים להגדיר prototype גם לפי מופע, כלומר בצורה יותר מדויקת, לשנות ערך שהוגדר על ידי prototype:

```
function User () {

};

User.prototype.name = 'Gili';

var user = new User();
user.name = 'Avi';

var joi = new User();
```

```
// output Gili
console.log('joi name (will take from prototype): ' + joi.name);
console.log('user name (will take from object): ' + user.name);
```

## :JSON

```
var obj = {
  a: 1,
  b: 2,
};
var json = {
  "a": 1,
  "b": 2
};
```

האובייקט הראשון הוא אובייקט רגיל והשני מדמה איך נראה אובייקט של JSON.

נשים לב להבדלים של הכתיבה של המפתחות וגם לכך שבאובייקט רגיל אנו יכולים להשאיר פסיק בסוף (שכביכול מחכה לכניסה של עוד איברים חדשים) וב-JSON לא.

```
var jsonProduct = '{ "title": "Pr title demo", "price": 25.3}';
var product = JSON.parse(jsonProduct);

console.log(product.title);
```

כאן אנו רואים שיש לנו משתנה שמכיל string של תבנית JSON והמרנו אותו להיות אובייקט רגיל.

```
var product = {
  title: 'Pr title demo',
  price: 25.3,
};

var jsonData = JSON.stringify(product);

console.log(jsonData);
```

וזו הצורה הפוכה.

## שיעור 11

### ירושה:

דיברנו בשיעור קודם על prototype ועכשיו נדבר על ירושה, נשים לב שירושה מעבירה גם את ה-`prototype`.

לצורך הדוגמא נגדיר פונקציה אנונימית ושני prototypes אחד שהוא משתנה `string` והשני פונקציה. לאחר מכן נגדיר פונקציה אחרת שתירש את הקודמת.

```
var Gallery = function(){};
Gallery.prototype.size = '100%';
Gallery.prototype.setup = function(){
  console.log( 'setup run!' );
};

var SliderGallery = function(){};
// INHERITENCE
SliderGallery.prototype = Object.create(Gallery.prototype);
```

כך זו הדרך הישנה לרשת.

בעצם אנו מגדירים את ה-`prototype` שזה האבא להיות הפונקציה הקודמת. בדוגמא שלנו אנו מגדירים באמצעות ה-`prototype` את הפונקציה `Gallery` להיות האבא של `SlideGallery`.

נדגיש שכל שינוי יחול על כולם בצורה של `reference`.

אנו צריכים לזכור זאת משום שנראה זאת כנראה במקומות עם קוד ישן וגם כי כאשר נכתוב בקוד החדש נצטרך לזכור את העיקרון של `syntactic sugar` שבעצם מאחורי הקלעים בכל אופן מתבצע הקוד הישן.

```
var Game = function() {};

Game.prototype.start = false;
Game.prototype.play = function() {
  console.log("play run");
};

var CarGame = function(){};

CarGame.prototype = Object.create(Game.prototype); // inheritance

// create new thing that fit only to this specific game
CarGame.prototype.speed = 110;
CarGame.prototype.drive = function() {
  console.log('Car is now driving');
};

var game1 = new CarGame();
```

כאן יצרנו פונקציה שנקראת משחק והגדרנו לה שתי הגדרות (`start` ו-`play`).

לאחר מכן יצרנו פונקציה אחרת שהיא משחק מכוניות והיא יורשת ממשחק.

במחלקת משחק מכוניות יצרנו אלמנטים ששייכים רק לה משום שאין מה לעשות איתם בפונקציה הראשית.

אם היינו רוצים לדרוס אלמנט מהאבא אז פשוט היינו מגדירים אותו אחרת, נניח אם המחלקה הייתה Person והייתה בה פונקציה `GetDetails()`, אז בפונקציה הבן יכולתי לממש את הפונקציה בצורה אחרת.

■ דגש: על אף שראינו זאת, לא נהוג ליצור משתנים בתור `prototype` משום שמטרתם היא להשתנות (כמו שמם) ולכן העדיפות היא לשים אותם בתוך המחלקה ולא על האב טיפוס.

## :ES6

קצת על סוגי משתנים:

המשתנה `var` מוגדר להיות רלוונטי למרחב הגלובלי כולו.

המשתנה `let` מוגדר להיות רלוונטי למרחב שבו הוא הוגדר בתוך סקופ של סוגריים מסולסלים.

```
for (var x = 0; x < 3; x++) {
    //do something
}
//output 3
console.log(x);

for (let y = 0; y < 3; y++) {
    //do something
}
//Error: uncaught referenceError: y is not defined
console.log(y);
```

בדוגמא כאן אנו רואים זאת בצורה מובהקת כאשר את ערך ה-`x` נצליח להחזיר גם מחוץ ללולאה ואת ה-`y` לא.

המשתנה `const` דומה ל-`let` מבחינת תכונות סקופים. אבל ב-`const` אנו רואים שוני בהגדרה שלו כאשר מדובר בסוגים שונים של משתנים.

כאשר נגדיר משתנה פרימיטיבי כ-`const` התכונות תהיינה כמו שאנו מכירים וזה יוגדר כקבוע.

אבל כאשר אנו מגדירים משתנה שאינו פרימיטיבי כמו מערך או אובייקט, מה שקבוע זה ה-`reference` ולא הערך, כלומר, הקיבוע לא יאפשר לנו להצביע מחדש על משהו אחר כאובייקט גדול אבל לא ימנע מאיתנו ערך בתוך האובייקט.

## :destructuring

```
let obj = {
    a:1,
    b:2,
    c:3
};
const {a:myA, c} = obj;
console.log("myA", myA);
```

אנו רואים כאן הגדרה של הערכים מתוך האובייקט החוצה לתוך שני משתנים מסוג `const`.

בנוסף ראינו שיש לנו אפשרות לעשות `alias` (as) באמצעות ':

דרך נוספת להעתקה (הבנייה):

```
const myAry_1 = [1,2,3,4,5];
const myAry_2 = [6,7,8,9,10];

const data = [...myAry_1,...myAry_2];
console.log(data);
```

שלוש הנקודות בעצם מגדירות למערכת לקחת את כל האיברים מהמערך.

```
const [myFirstElement, my2ndElement,,,mySomeElement] = data;
```

כאן אנו רואים, כמו שראינו לעיל, אפשרות לחלץ ערכים החוצה למשתנים מסוג const. עכשיו לגבי האיבר mySomeElement, הוא מקבל את הערך של האיבר במיקום כמו המיקום שלו, זה אומר בעצם שלפי מספר הפסיקים פלוס אחד, כלומר, אם יש לנו כאן חמישה פסיקים (גם אם לא היינו מגדירים את myFirstElement ואת השני), אז נקבל ב-mySomeElement את האיבר השישי.

💡 נזכיר כאן את האפשרות לכתיבת string בצורה הבאה:

```
let solution = 1+1;
let myString = `1 + 1 = ${solution}`;
```

■ דגש צדדי – ניתן לתת ערך ברירת מחדל כארגומנט לפונקציה, אך אם ישנם כמה ארגומנטים ורוצים גם אחד ברירת מחדל אז הוא חייב להיות האחרון.

כעת נשים לב בהעתקות לדבר הבא:

```
let israelCitizen = {
  name: 'Moshe',
  id: 1234565,
  address: {
    street: 'Byalik',
    houseNum: 50,
    city: 'Nes Ziona'
  }
};

let copyOfCitizen = {
  ...israelCitizen,
  address: { ...israelCitizen.address }
};

console.log('copyOfCitizen', copyOfCitizen);
israelCitizen.address.street = 'Lochamei Hagetaot';
console.log('copyOfCitizen.address.street', copyOfCitizen.address.street);
```

אם במשתנה copyOfCitizen לא הייתי מבצע את ההעתקה של address בצורה שאנו רואים כאן, אז היה נוצר לי מצביע בלבד ולא משתנה חדש.

מה שראינו עכשיו זו דוגמא טובה ל-deep copy (לעומת shallow copy שבסך הכל היה יוצר לנו מצביע).

כעת נראה את השימוש עליו דיברנו ב-syntactic sugar.

לאחר שהשפה עברה שדרוג הדברים התחילו להיכתב בצורה נוחה יותר אך נזכור כי מאחורי הקלעים כל מה שראינו עד עכשיו זה מה שקורה.



## מחלקות:

```
class Person {
  constructor(address) {
    this.address = address;
  }
  name = 'Gil';
  getAge = () => 31;

  justOtherMethod({first, last}) {
    console.log('first', first);
    console.log('last', last);
  }
  static STATIC_METHOD() {
    console.log('this is static');
  }
}
```

נשים לב שבפונקציה justOtherMethod בעצם מתבצע destructuring והאובייקט שהכנסתי כארגומנט מתפרש כשני משתנים first ו-last.

בנוסף הפונקציה בסטטית מתנהגת בדיוק כמו שאנו מכירים מסי שארפ.

וכך תיראה ירושה:

```
class SpecialPerson extends Person {
  constructor() {
    super("Some Address");
  }

  special = true;
}
```

:Async

```
setTimeout( () => {
  console.log('timer finished!')
}, 3000);

console.log("script started");
```

באמצעות הפונקציה הזו בעצם יצרנו הדמיה של תזמון של דברים, מה שיכול לקרות נניח כשמשהו אחד מתעכב יותר ממה שהיה אמור.

נניח שהבעיה שלנו היא ביצירה של שלושה פוסטים והשלישי מתעכב:

```
const posts = [{
  title: 'Post 1 demo title'
},
{
  title: 'Post 2 demo title'
}
];
```

אחד הפתרונות הוא callback hell (אסון הקולבקים) שזה אומר קינון פונקציות ובכל פעם אנו גורמים לפונקציה לחכות.

כך אני מתחיל:

```
const createPost = (post, callback1) => {
  setTimeout(() => {

    posts.push(post);
    // קורא למה שיתרחש בעתיד = כלומר רפרנס לפונקציה מסוימת
    callback1();
  }, 3000);
};
```

יצרתי פונקציה שמקבלת פוסט וקולבק.

■ המתודה push מכניסה את הפוסט לתוך המערך.

אנו קוראים לפונקציה שתיצור את הפוסט השלישי:

```
createPost({ title: 'Post 3 demo' }, renderPosts);
```

כאשר:

```
const renderPosts = () => {
  posts.forEach(post => {
    document.getElementById('posts-content').innerHTML += '<h3>' + post.title + '</h3>';
  });
};
```

מה שקרה כאן בעצם זה שהפונקציה של יצירת הפוסט קיבלה את הפוסט השלישי, מחכה לסיום העיכוב ואז מבצעת callback שדרכו היא מפעילה את הפונקציה שבעצם מפעילה את היצירה של כל הפוסטים.

## שיעור 12

קצת השלמות ונעבור ל-Angular.

יש לנו עמוד אינטרנט ויש לנו שני פוסטים שאנו יודעים מראש.

בנוסף כאשר העמוד נטען יגיע לאחר זמן מסוים הפוסט השלישי ואני לא מעוניין ששני הפוסטים יוצגו למשתמש לפני שהשלישי יגיע.

```
{title: ' Post 1 ', body: 'This is post 1'},
{title: ' Post 2 ', body: 'This is post 2'}
];

function getPosts(){
  let output = '';
  posts.forEach((post, index) => {
    output += `<li>${post.title}</li>`;
  });
  document.body.innerHTML = output;
}
```

כאן אנו רואים פונקציה שאמורה להדפיס על המסך את שני הפוסטים, כמובן שזה לשם ההדמיה.

■ על הדרך ראינו כאן את הפקודה `foreach` שיש אותה גם ב-LINQ והיא בעצם רצה בלולאה על המערך אליו היא מוצמדת ומבצעת את המתודה הפעולות שנכתוב לה לבצע בסוגריים.

### נקודה לפני שנמשיך:

```
function step1(nextStepFunction){
  console.log("start engine!");
  nextStepFunction();
}

function step2(){
  console.log("drive!");
}

step1(step2);
```

נשים לב למה שעשינו כאן:

יצרנו פונקציה שמקבלת כארגומנט פונקציה אחרת, עשינו זאת כי רצינו להבטיח שהצעד השני יקרה בוודאות לאחר שהצעד הראשון יקרה. (יכולנו גם להעביר פונקציה בצורה של למבדא ולא בהכרח ליצור את `step2`)

נשים לב שכאשר קראנו לפונקציות לא שמנו ל-`step2` סוגריים משום שהם יצרו `invoke` כלומר, הם יגרמו לקריאה לפונקציה ואנו לא רוצים זאת אלא רק להעביר אותה כפרמטר לפונקציה `step1`.

נהוג לקרוא לארגומנט `callback` ולא `nextStepFunction`. שהמשמעות היא שאנו אומרים לפונקציה "כשתסיימי – תחזרי אליי..".

```
function createPost(post){
  setTimeout(() =>{
    posts.push(post);
  }, 4000);
}
```

בפונקציה הזו ראינו בעצם קריאה ליצירת פוסט שלישי ודימינו שרת איטי יותר באמצעות המתודה `setTimeout` שמששה את ביצוע הפעולה עד למעבר מספר השניות שאמרנו לה.

כעת נראה איך נטפל בבעיה באמצעות הנאמר בתיבה הירוקה:

```
function createPost(post, callback){
  setTimeout(() =>{
    posts.push(post);
    callback();
  }, 4000);
}

createPost({title: ' Post 2 ', body: 'This is post 2'}, getPosts);
```

זו הייתה הצורה הכי "מסובכת" להבנה ופתרון.

בשיעור הקודם דברנו על מושג שנקרא `callback hell` (גיהנום הקולבקים).

זה אומר שבשיטה הזו, אם נרצה לבצע הרבה פעולות ולא רק שתיים, נצטרך ליצור קינון גדול מאוד של פונקציות מה שיסבך גם אותנו וגם כל אחד אחר בקריאה של הקוד.

לכן נראה פתרון נוסף, ניצור אפשרות כרגע שבעצם תבטיח לי שהפוסט ייוצר בעתיד, רק נזכור שכמובן יכולות לקרות תקלות והפעולה לא תתממש.

```
function createPost(post){
  return new Promise((resolve, reject) => {
    setTimeout(() =>{
      posts.push(post);
      let error = false; // מדמה מצב של תקלה

      if(!error){
        resolve();
      }else{
        reject();
      }
    }, 4000);
  });
}
```

כאן אנו רואים את הפונקציה מקודם שהפעם מחזירה לנו הבטחה – זה בעצם אובייקט מובנה ב-JS שמקבל למבדא ואם אין תקלה והכל תקין נעשה `resolve` ואם יש תקלה נעשה `reject`.

כעת נראה איך משתמשים בהבטחה (`promise`) שחוזרת אלינו.

```
createPost({title: ' Post 2 ', body: 'This is post 2'}).then(getPosts);
```

המתודה `then` בעצם אומרת למערכת לקרוא לפונקציה אם ההבטחה מתממשת.

```
createPost({title: ' Post 2 ', body: 'This is post 2'})
  .then(getPosts)
  .catch((errorDetails) => console.log("Error: " + errorDetails));
```

וכך גם נוכל לבצע פעולה מסוימת אם אכן תגיע שגיאה מהמערכת.

כאשר `errorDetails` זהו פרמטר המועבר דרך:

```
reject("An error occurred!");
```

היתרון הוא שנוכל לשרשר כמה פעמים then שכל אחד לוקח את הזמן שלו ומחזיר הבטחה, דבר שנותר מצב דומה ל-callback אבל הרבה יותר קריא.

נניח שיש לי כמה הבטחות ואני רוצה שכולן תתקיימנה אבל לא רוצה לעשות כמה פעמים then.

במקרה כזה, נוכל להשתמש בפקודה all שתכלול ב-promise אחד את כל האחרים:

```
const promise1 = Promise.resolve("hello world");
const promise2 = 10;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'good bye')
});
Promise.all([promise1, promise2, promise3]).then(values => console.log(values));
```

בסוף בפקודה האחרונה אמרנו לו בפקודה אחת להריץ את כל ההבטחות ואז בסוף להדפיס את הערכים שהגיעו מכל ההבטחות (שעטפנו במסגרת של מערך) לתוך הקונסול.

■ נשים לב שלא חייב להכניס רק promise לתוך all, נכניס מה שנרצה והוא פשוט יפרש את זה כתוצאה המוחזרת מ-promise.

דרך נוספת לפתרון הבעיה היא שימוש ב-async ו-await, העניין הוא שגם כאן זה מקרה של syntactic sugar משום שמאחורי הקלעים קורות ההבטחות שראינו קודם.

```
async function init() {
  try {
    await createPost({ title: ' Post 2 ', body: 'This is post 2' });
    getPosts();
  } catch (error) {
    console.log("Error: " + error);
  }
}

init();
```

השתמשנו כאן ממש בפונקציה הקודמת שבנינו מבחינת הפוסטים רק שהפעם יצרנו לקריאה לפונקציה מעטפת אחת של async-await ומעטפת נוספת של try-catch.

## :Angular

למה בעצם אנחנו צריכים כל מיני תבניות\frameworks?

לרוב זה עוזר לנו לדלג על העבודות הסיזיפיות שתופסות לנו זמן מיותר.

אז angular היא ממש framework שמכילה בתוכה המון אפשרויות כדי לעזור לנו ליצור את האתר שלנו כמה יותר טוב, איכותי וגם דינמי.

יש מודל שנקרא **Model View Controller**.

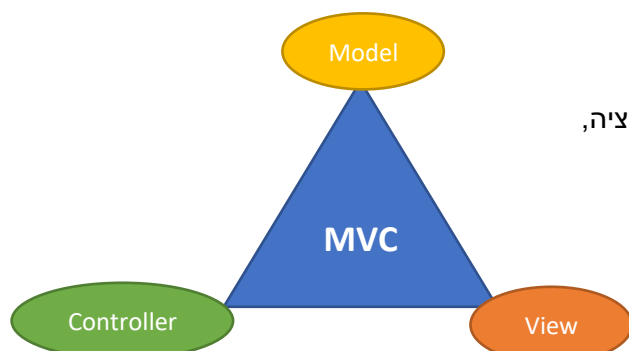
**Model**: כל המחלקות שעוסקות בישויות השונות של האפליקציה,

שהן בעצם ה-business logic.

**View**: תצוגה על המסך – מה שבאמת רואים (UI).

**Controller**: החלק שמאחד בין שניהם ואיתו אנחנו בדרך

כלל נתקשר ישירות.



נניח שיש לנו חנות, ואני רוצה לממש חשבוניות, מלאי, מוצרים. אז כל הישויות (המחלקות) יהיו ב- Model שזה כולל את כל המשחקים והחישובים הקשורים לנתונים.

View- יהיו כל החלקים החזותיים כמו איך תיראה הקופה והחשבונית ואיך המשתמש יראה אותם.

ואז בעצם ה-Controller הוא זה שידע לקחת מידע מהמודל ולשלוח אותו ל-UI וההיפך.

אז לצורך הנושא שלנו, angular נותן מענה לתבנית הזו בכללותה ו-react (שנלמד בהמשך) נותן מענה רק לחלק של ה-View.

npm – זה כמו עולם ה-Nuget של C#, איתו אנחנו בעצם מתקינים חבילות רצויות.

כאשר אנו מתחילים ויוצרים פרויקט ב-JS ה-npm יוצר קובץ JSON שהוא המקביל לקובץ csproj שנוצר בפרויקט ב-C#.

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course/angular (master)
$ #npm install -g @angular/cli
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course/angular (master)
$ npm --version
8.3.1
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course/angular (master)
$ explorer .
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course/angular (master)
$ cd ..
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course (master)
$ mkdir newLib
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course (master)
$ cd newLib/
```

```
USER@DESKTOP-77CPKAS MSYS ~/source/repos/learn/net-course/newLib (master)
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
```

```
See 'npm help init' for definitive documentation on these fields
and exactly what they do.
```

```
Use 'npm install <pkg>' afterwards to install a package and
save it as a dependency in the package.json file.
```

בשתי הפקודות הראשונות ראינו את ההתקנה של חבילה גלובלית כללית למחשב.

לאחר מכן יצרנו תיקייה חדשה לצורך הפרויקט בשם "newLib", נכנסנו אליה והתחלנו יצירת פרויקט באמצעות הפקודה "npm init".

ברגע הזה התחלנו יצירה של פרויקט ואנו נשאלים על ידי המערכת את השאלות הבאות:

```
Press ^C at any time to quit.
package name: (newlib) Bubul
Sorry, name can no longer contain capital letters.
package name: (newlib) bubul
version: (1.0.0)
description: my first lib
entry point: (index.js)
test command:
git repository:
keywords:
author: Eyal Bardugo
license: (ISC) MIT
```

נותנים שם לפרויקט שלנו (רק באותיות קטנות).

מגדירים לו גרסה (אפשר לקחת את מה שמציע ומקסימום לשנות אחר כך).

ניתן להוסיף תיאור ובחרים נקודת פתיחה לספריה בעת הפעלה (בתמונה זה הקובץ index.js).

ניתן להוסיף כלים שיריצו טסטים על קטעי תוכנה, כתובת ל-git להעלות לריפוזיטורי ומילות מפתח שבדרך כלל משמשות להפצת הספרייה.

בנוסף ניתן להוסיף שם יוצר ורישיון לצורך הפצה בקוד פתוח.

לאחר שסיימנו המערכת יוצרת את הקובץ JSON עליו דיברנו לעיל.

```

1  = {
2    "name": "bubul",
3    "version": "1.0.0",
4    "description": "my first lib",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "Eyal Bardugo",
10   "license": "MIT"
11 }
12

```

זה הקובץ של ה-JSON.

נוכל להיכנס ל-npm (אתר) ולמצוא שם עוד ספריות שניתן להוריד ולהתקין על מנת להשתמש בה. אם נרצה להתקין חבילה נוספת לספרייה שלנו, נניח שנרצה את החבילה rxjs (ספרייה קיימת) נכתוב ב-cmd **בתוך** התיקייה את הפקודה: `npm install rxjs`. לאחר התקנת החבילה, נוסף לי לקובץ ה-JSON השורות הבאות:

```

"dependencies": {
  "rxjs": "^7.5.4"
}

```

■ כדאי לשים לב שהספריות תופסות מקום המחשב.

כאן נכנסת נקודה חשובה:

כאשר אנו עובדים עם הפרויקט ואני רוצה להעביר אותו, אז אני לא מעביר אותו יחד עם תיקיית `node_modules` (כמו שלא מעבירים `C#-abin`).

אם אני אוסיף ל-`dependencies` בקובץ ה-JSON שם של ספרייה עם הגרסה שלי, מספיק שאכנס ל-cmd שלי בתיקייה ואכתוב `npm install` זה יתקין את כל מה שכתוב בקובץ ועדיין לא קיים אצלי.

בנוסף נוצר לי קובץ JSON נוסף שנקרא `package-lock` ששם הוא שומר את ה-`dependencies` ששקיימים עם הגרסאות שהותקנו ומספיק שמי שקיבל את הפרויקט ייכנס דרך הספרייה ל-cmd ויכתוב אצלו `npm install` זה יתקין את כל החבילות הרצויות.

כשהתקנו את החבילה הגלובלית, על הדרך התקנו גם כלי לשורת פקודה שעוזר לנו ליצור פרויקטים.

הכלי הזה מאפשר לנו לכתוב `ng new <proj name>` ואז נישאל כמה שאלות ליצירת הפרויקט (אם נרצה להוסיף משהו מאנגולר ובאיזו פלטפורמת עיצוב נרצה להשתמש – בחרנו כרגע CSS).

זה יתקין כמה דברים.

```

C:\Users\gilal\OneDrive\... .NET\6 ... HTML\Angular\firstProj>cd myFirstProj
C:\Users\gilal\OneDrive\... .NET\6 ... HTML\Angular\firstProj\myFirstProj>ng serve -o

```

לאחר מכן ניכנס לתיקייה ונריץ את הפקודה שבתמונה.

```

Initial Chunk Files | Names          | Raw Size
vendor.js           | vendor         | 1.70 MB
polyfills.js        | polyfills      | 294.85 kB
styles.css, styles.js | styles         | 173.23 kB
main.js             | main           | 47.99 kB
runtime.js          | runtime        | 6.52 kB
| Initial Total | 2.21 MB

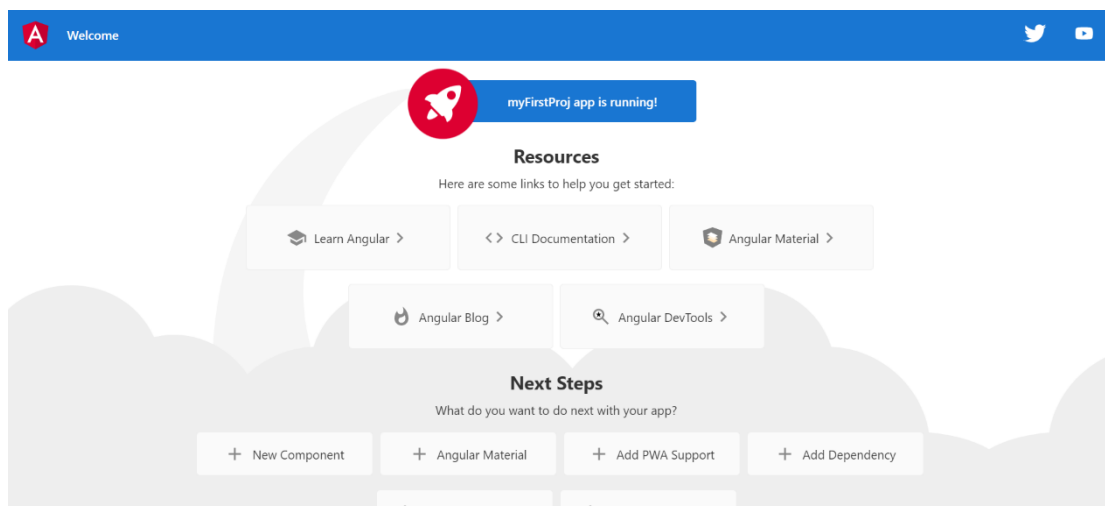
Build at: 2022-03-06T04:48:00.829Z - Hash: 051a34585ea88a73 - Time: 12962ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✔ Compiled successfully.

```

מה שזה יעשה, זה בעצם יצור האזנה לקוד שלי לעמוד האינטרנט שאני בונה (דומה ל-`watch live` server) – בתמונה <=

כעת ייפתח לנו העמוד שמאזין לנו בעצם:

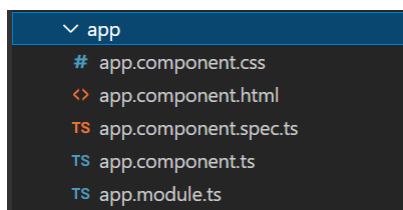


כל פעם שנחליט לשנות משהו זה יראה את מה שרואים בתמונה הקודמת שוב מחדש עם שינוי בהתאם לשינויים.

כמובן שאני יכול גם למחוק את כל העמוד ולהתחיל מאפס.

### נחזור ל-MVC:

החלק של ה-View באנגולר, מיוצג באמצעות רכיבים שנקראים קומפוננטות. קומפוננטה זה מושג, היא מורכבת מכמה קבצים וכמובן שבעולם האתרים הכוונה לקבצי HTML, CSS, SASS, ACSS, JS וכו'.



קומפוננטה זה רכיב ויזואלי ואלו הקבצים הקיימים בחלק הזה:

### הקבצים הקיימים בהגדרות:

#### ":.gitignore"

הכוונה היא, מאילו קבצים גיט יכול להתעלם.

לדוגמא מהתיקייה node\_modules שדיברנו עליה קודם לכן, היא יכולה להיות גדולה מאוד ולכן היא תהיה ברשימת ההתעלמות של גיט.

#### ":angular.json"

מפרט על כל קבצי הפרויקט.

#### ":karma.conf.js"

מכיל ספרייה של טסטים שנקראת קארמה.

#### ":README.md"

קובץ מקודד שמתפרש להיות קובץ טקסט רגיל.

**assets:** זו תיקייה של כל הדברים הסטטיים בפרויקט כמו תמונות.

**environment:** אפשרות להגדיר כמה סביבות עבודה, כלומר נניח מגדירים מצב מסוים על מחשב אחד ומצב אחר על מחשב אחר.

**Steps to create new Angular App:**

- install (global) @angular/cli  
`npm install -g @angular/cli`
- create new app  
`ng new [YOUR APP NAME]`
- enter the new folder  
`cd [YOUR APP NAME]`
- serve your app  
`ng serve`
- open your browser and browse to localhost:4200

סיכום שלבי יצירת פרויקט אנגולר



## שיעור 13

נושא קטן להשלמה ב-JS:

עם הזמן נוצר מצב שבו יש לנו הרבה תגיות סקריפט וזה לא כל כך יפה ונוח.

בנוסף היה שינוי מסוים ואנשים ראו שאפשר לעשות כמו בצד השרת node ולהגיד שקובץ מסוים מכיל תוכן מקבצים אחרים – מכאן השינוי שנעשה ב-ES6 שהוסיפו את האפשרות לעשות import.

כלומר, אפשר לעשות מעין using בין דפים שונים של JS.

כדי להשתמש באובייקטים\מחלקות שנמצאים בעמוד אחר נצטרך להגדיר אותם בעמוד שלהם בתור export ולעשות להם import בדף המייבא.

ניתן גם לייצא אובייקט אחד (בלבד) כברירת מחדל.

נניח שיצרנו קובץ ראשון בשם 'mdl1' (כלומר מודל 1):

```
const Person = {
  name: 'John',
  family: 'Dou',
  id: 123,

  shut() {
    alert("HI!!");
  }
}

const Car = {
  provider: 'Tesla',
  model: '3',
}

//export default Person; => מייצא את פרסון כברירת מחדל,
//                          כלומר מספיק שנייבא את הקובץ .
//                          פרסון יהיה בשימוש

export {
  Person,
  Car
}
```

נשים לב שראינו שני אובייקטים בשם person ו-car, והכנסנו אותם לתוך פקודת export על מנת שיהיו זמינים לכל מי שייבא את העמוד הזה ויקח ממנו יכולות.

בנוסף ראינו דוגמה שנשארה נכון לעכשיו ב-comment, והיא האפשרות ליצור אובייקט דיפולטיבי לייצוא.

כמובן נוכל לייבא קובץ מכל מקום במחשב כל עוד ננקוט בשם כל ה-path שלו אבל כן נהוג שהם יהיו קרובים ואפילו באותה התיקיה.

כעת נוכל ליצור קובץ נוסף נניח ששמו יהיה 'mdl2':

```
import { Person, Car } from "./mdl1.js";
// מייבא ערך ברירת מחדל => import MyPerson from "./mdl1.js";

export class PersonWithCar {
  a = 1;
  b = 2;
  name = Person.name;
  HisCarSupplier = Car.provider;
}

var instance = new PersonWithCar();
console.log(instance);
```

כאן אנו רואים בשורה הראשונה את היבוא של העמוד 'mdl1' ושימוש באובייקטים הקיימים שבו.

- כאן המקום להזכיר\להדגיש את השוני בין אובייקט למחלקה, הגדרת איברים באובייקט מתבצעת באמצעות נקודתיים (':') ובמחלקה באמצעות שווה ('=').

וכאשר נרצה לייבא אותם לעמוד הראשי, נוסיף:

```
<script src="./mdl1.js" type="module"></script>
<script src="./mdl2.js" type="module"></script>
```

#### דגשים נוספים:

- ניתן לשרשר ייבוא טפסים, כלומר, ניתן לייבא מ-mdl1 ל-mdl2 ואם נייבא מ-mdl2 לקובץ חדש mdl3 אז mdl3 יקבל גם את מה שיובא מ-mdl1 מבלי לקרוא לו.
- ניתן לכתוב \* import ואז בעצם הכוכבית אומרת שנייבא כל מה שיש עליו export בעמוד ממנו אנו מייבאים.
- נוכל להשתמש ב-alias (as) כדי לקרוא לאובייקט\מחלקה המיובאת בשם אחר.

#### נחזור ל-Angular:

נזכיר שיש קובץ שנוצר בעת פתיחת פרויקט ומתעדכן עם כל הוספת ספרייה שנקרא package-lock.json ששומר בתוכו תחת השם dependencies את הספריות והגרסאות שהותקנו, כך שכל מי שיקבל את הפרויקט ממני יוכל לפתוח את ה-cmd בתיקייה ופשוט לכתוב npm install וזה ירוץ ויתקין לו את כל מה שאמור להיות בשימוש בפרויקט.

אז ראינו שביצירת הפרויקט נוצרים לנו 4-5 קבצים שמרכיבים את העמוד והתוכן כמובן יהיה בקובץ ה-HTML.

בקובץ הזה נוכל גם למחוק את כל מה שכתוב ולהתחיל מאפס (מה שמגיע שם הוא התבנית הראשונית של אנגולר)

דברנו על כך שהמסך יכול להתחלק לקומפוננטות, כלומר כל חלק וחלק, אפילו ברמת הפרסומת או כפתור.

תכונה שקיימת בספרייה של אנגולר בשונה מכתביה רגילה של HTML, זו האפשרות להציג שגיאות בזמן אמת באמצעות ה-server שפתוח ומאזין ברקע.

כדי לפתוח את ה-ngServer שמאזין בצורה טובה, כלומר, בצורה שתבטיח שאנו מאזינים עם השרת למקום הנכון, אפשר לפתוח טרמינל ב-Visual Code ושם יש אפשרות לבחור שהטרמינל יהיה cmd

כמו זה של המחשב אליו אנחנו רגילים (יכול להיות גם powershell או כל אחד שמעדיפים ממה שקיים שם).

- דגש ביניים: אם נכתוב בפקודה "ng sever" ולא "ng server -o" אז זה יפתח את ההאזנה מבלי לפתוח את העמוד האינטרנטי עצמו.

כדי ליצור קומפוננטה חדשה נוכל לכתוב בטרמינל של ה-Visual Code את הפקודה: "ng generate component <name>".

זה מה שיופיע לנו:

```
C:\Users\USER\source\repos\learn\hackeru\ng\Angular2>ng generate component MyFirst
CREATE src/app/my-first/my-first.component.html (23 bytes)
CREATE src/app/my-first/my-first.component.spec.ts (634 bytes)
CREATE src/app/my-first/my-first.component.ts (282 bytes)
CREATE src/app/my-first/my-first.component.css (0 bytes)
UPDATE src/app/app.module.ts (402 bytes)
```

אז מה אלו כל הקבצים שהוא יצר?

נוצרה תיקייה שנקראת בשם שנתנו לה, ובתוך הקובץ תופיע פסקה בתגית <p> שנראית כך:

"<p>(NAME) works!</p>"

- נשים לב שאנו לא צריכים את כל המעטפת שדברנו עליה בשיעורים הראשונים של DOCTYPE, משום שכל קובץ שנפתח יהיה חלק מהעמוד הגדול, קובץ לקומפוננטה.

כעת, נניח שקראנו לקומפוננטה החדשה שלנו "myFirst". אם נלך לקובץ הראשי של אנגולר ונכתוב שם: "<app-my-first></ app-my-first>" – זה יכתוב לנו על המסך את מה שמופיע בקובץ של myFirst.

- לכל קומפוננטה (כמו שרואים בתמונה) נוצר גם קובץ CSS, זה אומר שניתן לתת לחלק הזה עיצוב משלו.

נניח שיצרנו שוב קומפוננטה (אפשר גם פקודה מקוצרת – "ng g c my2") וקראנו לה my2 ואפילו עוד אחת וקראנו לה my3.

לכל הקומפוננטות יש את המקדם app אותו ניתן לשנות, אבל נגיע לזה אולי אחר כך..

כעת נראה שרשור בין הקבצים, כלומר, אם ב-myFirst נכתוב כך:

```
<p>my-first works!</p>
<p>Hello</p>
<app-my2></app-my2>
```

ובתוך my2 נכתוב כך:

```
<p>my2 works!</p>
<app-my3></app-my3>
```

ובתוך my3 נכתוב רק:

```
<p>my3 works!</p>
```

אז בעצם מה שיוצג בעמוד יהיה:

## Hello Angular

my-first works!

Hello

my2 works!

my3 works!

TypeScript:

נפתח את הקובץ של typescript (רנדומלית לקחנו את של my2) ונראה מה יש בו:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-my2',
  templateUrl: './my2.component.html',
  styleUrls: ['./my2.component.css']
})
export class My2Component implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

בחלק העיקרי יש לנו שם class שנקרא My2Component עם בנאי ומתודה שנקראת ngOnInit כאשר ניתן לייצא את המחלקה.

החלק שמעל המחלקה נקרא decorator – עליו נשבר בהמשך.

ובשורה הראשונה אנו רואים import מספרייה של אנגולר.

Decorator – זה בעצם מאפיין כמו attribute ב-C#.

אצלנו כאן אנו רואים שבחלק זה הגדרנו לו את הסלקטור שזה בעצם איך נקרא לחלק הזה בתגית (כמובן נוכל לשנות את השם כרצוננו).

כדי ליצור קומפוננטה בצורה ידנית ניצור קובץ typescript חדש בשם "manual components" (רק לשם הדוגמא) שיראה כך (בצורה הכי מינימלית):

```
import { Component } from "@angular/core";

@Component({
  selector: 'app-manual-component',
  template: `<h1>this is manual-component</h1>`
})
export class ManualComponent {

}
```

ובקובץ app.modules.ts נראה ניצור הצהרה חדשה (declarations) וזה יצור אוטומטית גם יבוא חדש:

```
import { AppComponent } from './app.component';
import { ManualComponent } from './manual-component/man
import { MyFirstComponent } from './my-first/my-first.c
import { My2Component } from './my2/my2.component';
import { My3Component } from './my3/my3.component';

@NgModule({
  declarations: [
    AppComponent,
    MyFirstComponent,
    My2Component,
    My3Component,
    ManualComponent
  ],
})
```

כעת אם נקרא לתגית שלו בעמוד הראשי נראה בעמוד את מה שאמרנו לו לכתוב שורה של template בחלק של ה-decorator:



כעת נשים לב לשוני שהיה ב-decorator שלנו ממה שראינו באלו שנוצרו באופן אוטומטי,

```
@Component({
  selector: 'app-my2',
  templateUrl: './my2.component.html',
  styleUrls: ['./my2.component.css']
})
```

בידני יצרנו פשוט template, וזה בעצם יצירה של אלמנט פשוט וזה מצב שבו הקומפוננטה מאוד קטנה ואני לא להבדיל אותה לקובץ נפרד.

באוטומטי הוא יוצר קובץ HTML נפרד בתוך התיקייה ולכן השם של המאפיין נקרא templateUrl.

גם לגבי העיצוב נוכל להוסיף מערך סטרינגי של עיצובים עם שם המאפיין styles כאשר גם כאן באוטומטי הוא יוצר קובץ נפרד.

Module לעומת component:

אנגולר הוא framework מובנה מאוד ויש בו מונח שנקרא module.

Module אחד מכיל בתוכו כמה קומפוננטות וגם עוד כל מיני הגדרות.

לפעמים יש לנו יכולת שמתרכזת בכמה קומפוננטות בבת אחת ולא נרצה שיהיה משהו ענק שמרכז הכל בבת אחת אלא נרצה שיהיה לכל קבוצה את הגג שלה. כלומר, אם יש קומפוננטות שמשתמשות בניח ביכולות צד שלישי בגישה לשרתים אז אפשר לרכז את זה במודול.

כך נראה הקובץ של ה-module:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { MyFirstComponent } from './my-first/my-first.component';
import { My2Component } from './my2/my2.component';
import { My3Component } from './my3/my3.component';

@NgModule({
  declarations: [
    AppComponent,
    MyFirstComponent,
    My2Component,
    My3Component
  ],
```

```
imports: [
  BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

נראה כעת קצת דברים אמיתיים ובהמשך נשוב לתיאוריה.

נניח שנכתוב בעמוד הראשי שלנו את הביטוי הבא:

```
<h1>1 + 1 = {{1+1}}</h1>
```

השיטה הזו נקראת string interpolation.

נקבל בעמוד  $1 + 1 = 2$  <= זה קרה כי הסוגריים המסולסלים יכולים להחזיק ביטוי JS שמחזיר משהו, אמנם כאן זה לא כל כך משמעותי אבל בקודים גדולים יש לזה שימוש יותר משמעותי.

נניח שהוספתי לקובץ ה-typeScript משתנה בצורה הבאה:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular2';
  myName = 'Gil';
  age: number = 31;
}
```

כלומר הוספתי את השם שלי ואת הגיל למחלקה.

אם אקרא לו כך:

```
<p>My name is: {{myName}} and my age is: {{age}}</p>
```

אז גם כאן השם שכתוב במחלקה יופיע וגם הגיל. וכל פעם שהשם ישתנה בתוך המשתנה הוא ישתנה גם במיקום בו קראנו לו.

נשים לב שהגדרנו את age עם type וקראנו לו number ולא נוכל להכניס לו משהו שהוא לא מספר נניח string. הדבר הזה הוא אכן חידוש משום שמשתמשים פה בשפה של JS וזו תוספת.

typescript מגבילה אותנו בזמן הפיתוח ממש בשונה מ-JS כאשר בפועל מאחורי הקלעים מיוצר קוד של JS לגמרי – אבל רק בזמן הפיתוח, כלומר אם אני מושך מידע מהמשתמש אני צריך להיות אחראי גם לבדיקת המידע שיגיע מהמשתמש.

נוכל גם להוסיף אובייקט למחלקה שלנו:

```
candy = {
  color: 'yellow',
  taste: 'sweet',
};
```

ובקובץ הראשי:

```
<h3>
  I love my candy, which is {{candy.color}}
  <br>
  and the taste is {{candy.taste}}
</h3>
```

אז גם כאן יקרה אותו אפקט שראינו קודם.

עכשיו אם הייתי בונה בנאי במחלקה שמשנה משתנה:

```
constructor(){
  this.age = 32;
}
```

אז הוא היה דורס את המשתנה שהגדרתי קודם.

באנגולר ספציפית יש דבר שנקרא מחזור חיים של קומפוננטה ויש אירועים שקורים אחרי הבנאי כמו OnInit שנלמד עליה יותר בהמשך, אבל חשוב להבין שאם שמנו משתנה ודרסנו אותו בבנאי אז הבנאי יתפוס.

נשים לב לפרט חשוב שה-framework יוצר את המופע של המחלקה שלנו ולא ניתן ליצור מופע נוסף משל עצמנו.

כעת נשים לב שנוכל גם ליצור מתודות בתוך המחלקה:

```
getHairColor(color:string){
  return "*" + color + "*";
}
```

נוכל לקרוא לה בצורה הבאה:

```
<h4>Hair color: {{getHairColor("yellow")}}</h4>
```

דבר נוסף, אנו יכולים להוסיף משתנה תמונה:

```
imageSrc: string = 'https://images.unsplash.com/photo-1645902718013-370a382f14e9?ixlib=rb-1.2.1&ixid=MnwxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8&auto=format&fit=crop&w=387&q=80';
```

מעתיקים פשוט את כתובת התמונה ומציגים בצורה הבאה:

```

```

דבר אחרון, ראינו שבהגדרת המחלקה אנו מוסיפים את החלק של implements:

```
export class My3Component implements OnInit
```

זה בעצם אומר שהמחלקת אוכפת interface שמחזיק פונקציה בשם ngOnInit().

השוני ממה שראינו במחלקות בשם extends הוא ש-extends זה ירושה ו-implements אכיפה של ה-interface.

אם נרצה להגדיר prefix (מקדם) אחר במקום app, בעת היצירה של הקומפוננטה, נוכל לכתוב בפקודה: "ng g c my4 -prefix <name>" כאשר <name> זה השם של המקדם אותו נרצה.

דבר זה שימושי במקרים שבהם נרצה לעטוף בשם מקדם מסוים קבוצה של קומפוננטות שקשורות לאותו הנושא.