



Type Script

מקורות מידע חיצוניים:

- [גוגל AI - ג'ימיני](#)
- [מכללת האקריו - קורס בניית אתרים](#)
- [התיעוד הרשמי של TypeScript](#)

מה זה TypeScript?

TypeScript היא שפת תכנות שפותחה על ידי מיקרוסופט, המבוססת על JavaScript. היא מוסיפה שכבת טיפוסים סטטיים (static typing) ל JavaScript-מה שהופך את הקוד שלך לבטוח יותר, קריא יותר וקל יותר לתחזוקה. אם אתה מכיר JavaScript, המעבר ל-TypeScript יהיה חלק וטבעי.

למה להשתמש ב-TypeScript?

- **טיפוסים סטטיים:** מאפשרים לך לתפוס שגיאות בזמן ההידור, ולא בזמן הריצה, מה שמוביל לקוד יציב יותר.
 - **תמיכה בקלאסים וממשקים:** מאפשרים לך לכתוב קוד מובנה ואלגנטי יותר.
 - **תמיכה במודולים:** מאפשרים לך לארגן את הקוד שלך בצורה מודולרית וניתנת לשימוש חוזר.
 - **תמיכה בגנריקה:** מאפשרת לך לכתוב פונקציות וטיפים גנריים, מה שמגדיל את גמישות הקוד שלך.
 - **תמיכה ב-JSX:** מאפשרת לך לכתוב קוד React בצורה חלקה ויעילה.
- *typescript בנויה על javascript לכן גם אם לא נגדיר סוגי משתנים הקובץ יעבוד אבל לא לשכוח את המטרה של שימוש בשפה שעוזרת לנו בכל הדברים הנ"ל.

מושגים בסיסיים ב-TypeScript:

- **טיפוסים ב-TypeScript:** לכל משתנה יש טיפוס מוגדר (למשל, number, string, boolean, array, object). עוזר למנוע שגיאות הקלדה בזמן ההידור.
- **אינטרפייסים:** מגדירים את צורת האובייקטים, כולל השדות והשיטות שלהם.
- **קלאסים:** מאפשרים לך ליצור תבניות לאובייקטים, עם תכונות ושיטות משלהם.
- **מודולים:** מאפשרים לך לחלק את הקוד שלך לקבצים נפרדים ולייבא אותם לפי הצורך.
- **גנריקה:** מאפשרת לך לכתוב פונקציות וטיפים שיכולים לעבוד עם מגוון סוגים של נתונים.

דוגמה פשוטה:

```
let message: string = 'Hello, TypeScript!';
```



```
• console.log(message);  
•  
• interface Person {  
•   firstName: string;  
•   lastName: string;  
• }  
•  
• function greet(person: Person) {  
•   console.log('Hello, ' + person.firstName + ' ' +  
•     person.lastName + ' !');  
• }  
•  
• let user: Person = {  
•   firstName: 'John',  
•   lastName: 'Doe'  
• };  
•  
• greet(user);
```

כלי פיתוח:

- **TypeScript Compiler** ממיר את קוד ה TypeScript לקוד JavaScript שניתן להריץ בדפדפן או בסביבת זמן ריצה של Node.js.
- **Visual Studio Code** עורך קוד פופולרי עם תמיכה מצוינת ב TypeScript, כולל השלמה אוטומטית, ניווט קוד ובדיקת שגיאות בזמן אמת.
- **Angular** מסגרת פופולרית לפיתוח אפליקציות אינטרנטיות, המבוססת על TypeScript.

בשפת JavaScript ניתן להגדיר משתנה ולשנות את המידע וסוג המידע הקיים בו אינספור פעמים, זהו מקור לשגיאות בקודים בין מפתחים או כאשר יש לנו מספר גדול של קבצי קוד יהיה לנו קשה מאוד לתפוס את השגיאה.

בשפת TypeScript אנו נותנים למשתנים לא רק מידע אלא גם מגדירים את סוג המידע שהוא מכיל למשל משתנה מסוג: Boolean, String, number. השפה עושה לנו סדר בקוד.

יש לציין כי TypeScript היא שפת פיתוח שנועדה למפתחים והדפדפן אינה מכיר בה כמו שמכיר בשפות צד לקוח כגון: HTML, CSS, JAVASCRIPT.

השפה עוזרת לנו המפתחים למנוע ולצפות אונליין בשגיאות בקוד באמצעות הטרמינל, עולים על הבעיות כבר בזמן כתיבת הקוד.

על מנת שיוכל לעבוד עם קובץ TS (typescript) נצטרך לקמפל (להמיר) את הקובץ ל-JS.

כעת נעבור על השלבים שיש לבצע כדי להתחיל בעבודה עם TypeScript

- ראשית נתקין את [node.js lts](#) הגירסה העדכנית ביותר! לאחר ההורדה וההתקנה נוכל לבדוק שאכן מותקן במחשב ע"י פתיחת הטרמינל CMD ושם נכתוב את הפקודות הבאות:



node -v

npm -v

V מייצג version גירסה.

- ניגש לפרויקט שלנו ב- VS Code וניצור קובץ TS
 - נפתח חלון טרמינל חדש new terminal ניתן לקצר ע"י הקשה במקלדת:
ctrl+shift+backtick
 - נתנייד בטרמינל בצורה הבאה:
cd.. - תיקיה אחורה
cd folderName – קדימה לתוך התיקיה שציינו את שמה
- *יש לשים לב שאנחנו נמצא בתוך התיקיה של הפרויקט שבו נמצא הקובץ TS.*

- כעת נתקין את הספרייה TS על ידי הפקודה הבאה:

```
npm i -g typescript
```

**באותיות קטנות בלבד!*

מושגים:

new packed manager – npm

install – i

global – -g

library name – typescript

- כעת ניצור קובץ הגדרות לקומפיילר ts config על ידי הפקודה:

```
tsc -init
```

- נקמפל את הקובץ TS שלנו ל-JS ע"י הפקודה:

```
tsc filename.ts
```

לדוגמא: tsc script.ts

- נקשר ב-HTML את הסקריפט שלנו לקובץ JS שנוצר מהקימפול.

במידה ואנחנו רוצים המשך עבודה עם קובץ TS נוכל לבצע קימפול אוטומטי ע"י הפקודה הבאה:

```
tsc filename.ts -watch
```

לדוגמא: tsc script.ts -watch

כעת כל שורת קוד שנכתוב בקובץ ts שלנו תומר לקובץ JS בזמן ביצוע שמירה.

כתיבת מקוצרת של השלבים לתחילת פרויקט:

יצירת קובץ TS

לנווט את הטרמינל לתיקיה בה נמצא קובץ TS

tsc script.ts מקשר קובץ לטרמינל

node script.js מקשר קובץ JS לTS (לא חובה)

tsc -- init - פותח קובץ הגדרות CONFIG



tsc --watch עוקב אחרי השינויים ומקמפל אוטומטית קובץ TS לקובץ JS

צורת הכתיבה של type בקוד:

הגדרה של מערך:

נגדיר את שם המערך סוג למשל number ולאחר מכן נבצע השמה של הערכים. לדוגמה:

```
let numbers:number[]=[1,2,3,4,5];
```

הגדרה של פונקציה:

- חובה להגדיר מה הפונקציה מחזירה נכתוב זאת לאחר הגדרת הפרמטר
- במידה ולא מחזירה ערך נציין void

דוגמאות:

```
function showAlert (message:string):void {  
  console.log(message);  
}
```

```
function calc (x:number):number {  
  return x*10;  
}
```

פונקצית חץ arrow function:

```
let calc:function = (x:number):number => x*10;
```

יצירת מבנה type של פונקציה שמקבלת ומחזירה מספר:

```
// יצירת מבנה של פונקציה שמקבלת ומחזירה מספר  
type getReturnNumber = (num:number) => number;  
// שימוש במבנה קיים בתוך משתנה חדש  
let calc:getReturnNumber = (x:number):number => x**x;
```

Union type:

מאפשר לנו להגדיר משתנה שיכול להיות מאחד מכמה סוגים שונים. זה מאפשר גמישות רבה יותר בהגדרת משתנים ופונקציות, ובמקביל מספק בטיחות טיפוסים.

```
// הצהרה על משתנה שהסוג שלו יכול להיות מספר או מחרוזת  
let id:number = number | string;  
// השמת מספר בתוך המשתנה  
id = 12345;
```

Intersection Types:

מאפשר לנו ליצור סוג חדש שמשלב את המאפיינים של שני סוגים קיימים.



שימושי כאשר אובייקט צריך לעמוד בדרישות של שני סוגים שונים בו זמנית. לדוגמה:

```
interface Person {
  name: string;
  age: number;
}

interface Developer {
  skills: string[];
}

type PersonWithSkills = Person & Developer;
```

Literal Types:

מאפשר לנו להגדיר סוג שמקבל רק ערך ספציפי.

שימושי להגדרת קבועים או מצבים מוגדרים מראש. למשל:

```
• type Direction = "up" | "down" | "left" | "right";
```

Tuple Types:

מערך עם אורך קבוע וסוגים מוגדרים מראש לכל איבר.

שימושי לייצוג נתונים בעלי מבנה קבוע. לדוגמה:

```
let coordinates: [number, number] = [10, 20];
```

הגדרת מערך עם סוגים שונים של type:

```
let product: [string, string, string, number] = ["1", "iphone", "13 Pro", 500];
```

Generic Types:

מאפשר לנו ליצור פונקציות או מחלקות שפועלות על סוגים שונים, מבלי לציין את הסוגים הספציפיים מראש.

שימושי ליצירת קוד גנרי וניתן לשימוש חוזר. לדוגמה:

```
function identity<T>(arg: T): T {
  return arg;
}
```

Type Aliases:

מאפשרים לנו לתת שם לסוג קיים, לשיפור הקריאות. לדוגמה:

```
type UserId = number;
```



```
let userId: UserId = 123;
```

:any type

מאפשר לנו להגדיר type שיכול להיות מכל סוג לדוגמה:

```
let content:any = 10;
```

:unknown type

מאפשר לנו להגדיר type שאינו ידוע מראש לדוגמה:

```
let content:unknown = 10;
```

תחילה הטייפ לא ידוע עד שהוגדר ומעתה יהיה number לפי הדוגמא הנ"ל.

Mapped Types:

מאפשר לנו ליצור סוגים חדשים על בסיס סוג קיים, על ידי שינוי או הוספת מאפיינים.

Conditional Types:

מאפשר לנו ליצור סוגים שמשתנים בהתאם לתנאי מסוים.

מתי להשתמש באיזה סוג?

- **Union Types:** כאשר משתנה יכול להיות מאחד מכמה סוגים.
- **Intersection Types:** כאשר משתנה צריך לעמוד בדרישות של שני סוגים שונים.
- **Literal Types:** להגדרת קבועים או מצבים מוגדרים מראש.
- **Tuple Types:** לייצוג נתונים בעלי מבנה קבוע.
- **Generic Types:** ליצירת קוד גנרי וניתן לשימוש חוזר.
- **Type Aliases:** לשיפור הקריאות של הקוד.
- **Mapped Types:** ליצירת סוגים חדשים על בסיס סוג קיים.
- **Conditional Types:** ליצירת סוגים שמשתנים בהתאם לתנאי מסוים.

Casting ב TypeScript הוא תהליך המרת משתנה מסוג אחד לסוג אחר. זה מאפשר לנו לבצע פעולות על משתנה שדורשות סוג מסוים, גם אם הסוג המקורי שלו שונה. יש לציין היא לא משנה את סוג המשתנה! אלא אומרת תתייחס למשתנה מסויים כסוג מסויים.

מדוע משתמשים ב-Casting?

- **גמישות:** מאפשרת לנו לבצע פעולות שונות על אותו משתנה, בהתאם לצורך.



- **אינטראקציה עם ספריות או-API ים:** כאשר ספרייה או API מצפים לסוג מסוים, אנו יכולים לבצע Casting כדי להתאים את הנתונים שלנו.
- **התמודדות עם נתונים לא ידועים:** כאשר איננו בטוחים לגבי הסוג המדויק של משתנה Casting, מאפשר לנו לבצע פעולות בצורה בטוחה יותר.

סוגי Casting ב-TypeScript-

Casting צר (Narrowing):

- המרת משתנה מסוג כללי לסוג ספציפי יותר.
- דוגמה: המרת משתנה מסוג any לסוג string.
- **שימוש:** כאשר אנו בטוחים שהערך בפועל של המשתנה תואם לסוג החדש.

Casting רחב (Widening):

- המרת משתנה מסוג ספציפי לסוג כללי יותר.
- דוגמה: המרת מספר שלם (integer) למספר צף (float).
- **שימוש:** כאשר אנו רוצים לאבד מידע מסוג, למשל, כאשר מעבירים ערך לפונקציה שמקבלת סוג כללי יותר.

דוגמה:

```
• // Casting צר
• let value: any = "hello";
• let str: string = value as string; // Casting ל-string
•
• // Casting רחב
• let num: number = 42;
• let anyNum: any = num;
```

interface

יצירת אובייקט ליטרלי יחיד מסוג יוזר:

- אות ראשית גדולה חובה!
- Interface נותן מבנה מסוים (חוקיות) לאובייקט.
- במידה ונשמך במבנה של אובייקט interface מסוים מבלי שהגדרנו ערך למפתח אותו מפתח יחזיר undefined, ניתן להגדיר במבנה interface מפתח רשות ע"י סימן שאלה ובכך למנוע ממנו להופעה בתוך האובייקט במידה ולא קיים ערך.

```
//אות ראשית גדולה
interface User {
  id: string;
  userName: string;
  email: string;
  password: string;
```



```
// מפתח רשות לא חובה מוגדר ע"י סימן שאלה/  
age?: number;  
isLoggedIn: boolean;  
}  
  
let user: User = {  
  id: "1",  
  userName: "maorgross",  
  email: "maorgross247@gmail.com",  
  password: "123456789",  
  // לא הוגדר ערך לפמתח גיל ולכן לא יציג אותו/  
  isLoggedIn: true,  
};
```

יצירת מערך של אובייקטים ליטרליים מסוג יוזר:

```
// יצירת מערך של יוזרים  
let users: User[] = [  
  {  
    id: "1",  
    userName: "maorgross",  
    email: "maorgross247@gmail.com",  
    password: "123456789",  
    age: 30,  
    isLoggedIn: true,  
  },  
  {  
    id: "1",  
    userName: "maorgross",  
    email: "maorgross247@gmail.com",  
    password: "123456789",  
    age: 35,  
    isLoggedIn: true,  
  },  
  {  
    id: "1",  
    userName: "maorgross",  
    email: "maorgross247@gmail.com",  
    password: "123456789",  
    age: 40,  
    isLoggedIn: true,  
  },  
];
```

מתודה enum ב-type script :

מה זה Enum?

Enum (קיצור של enumeration) ב-TypeScript הוא סוג נתונים מיוחד המאפשר לך להגדיר קבוצה קבועה של ערכים נקובים. זה כמו ליצור סוג נתונים חדש ומותאם אישית, המכיל רק את הערכים שאתה מגדיר.



למה להשתמש ב-Enum?

קוד קריא יותר: במקום להשתמש במספרים או מחרוזות גנריות, אתה יכול לתת שמות משמעותיים לערכים שלך, מה שהופך את הקוד שלך לקל יותר להבנה ולתחזוקה. מניעת טעויות הקלדה: כאשר אתה משתמש ב-Enum, המערכת תדע בדיוק אילו ערכים חוקיים עבור משתנה מסוים, מה שמפחית את הסיכוי לטעויות הקלדה. שיפור אימות הנתונים: אתה יכול להשתמש ב-Enum כדי לבצע אימות על נתונים המתקבלים ממשתמש או ממערכת חיצונית. תמיכה בתכונות מתקדמות: Enum יכול לשמש בשילוב עם תכונות אחרות של TypeScript, כמו סוגים מותאמים אישית ואינטרפייסים, כדי ליצור קוד מסודר ומבוסס יותר.

סוגי Enum:

:Numeric Enum

כל ערך מקבל מספר באופן אוטומטי, החל מ-0. אתה יכול גם להגדיר את הערך הראשוני באופן ידני, והערכים הבאים יגדלו באופן אוטומטי. דוגמה:

```
enum Direction {  
  Up,  
  Down,  
  Left,  
  Right,  
}
```

ב-Enum הזה, Up יהיה שווה ל-0, Down יהיה שווה ל-1 וכן הלאה.

:String Enums

כל ערך הוא מחרוזת. דוגמה:

```
enum Color {  
  Red = "red",  
  Green = "green",  
  Blue = "blue",  
}
```

תכונות מתקדמות:



Enums מחושבים: אתה יכול להשתמש בביטויים כדי לחשב את ערך ה-Enum.

Enums רקועים: Enums רקועים מאפשרים לך להגדיר קבוצה של קבועים.

Enums היברידיים: Enum יכול לשלב בין ערכים מספריים ומחרוזות.

מתי להשתמש ב-Enum?

- כאשר יש לך קבוצה מוגדרת מראש של ערכים קבועים.
- כאשר אתה רוצה לשפר את קריאות הקוד ולמנוע טעויות.
- כאשר אתה רוצה לבצע אימות על נתונים.

דוגמאות נוספות:

```
enum Direction {
  north,
  south,
  east,
  west,
}

console.log (Direction.west);
//3 ידפיס
```

```
enum StatusCode {
  notFound = 404,
  seccess = 200,
  created = 201,
  serverError = 501,
}

console.log(StatusCode.notFound);
//404 ידפיס
```

Class ב-typescript

יש להגדיר בקונסטרוקטור גישה למאפיינים באמצעות מילים שמורות הבאות:

privet – מאפשר לנו לגשת למשתנה אך ורק מתוך מחלקה.

protected – ניתן לגשת למשתנה מתוך המחלקת או מחלקה יורשת.

readonly – אפשר לקרוא את המידע השמור במשתנה, לא ניתן לעדכן את ערכו.

public – ניתן לגשת למשתנה ולעדכן את ערכו מכל מקום.

דוגמה לכתיבת קלאס ב-typescript:

```
class Person {
```



```
    constructor (private id: string, protected name: string, readonly
    birthday: string, public age: number){
        this.id = id;
        this.name = name;
        this.birthday = birthday;
        this.age = age;
    }

    getId(): string {
        return this.id;
    }

    setId(newId: string) : void{
        this.id = newId;
    }
}

let p1: Person = new Person("1", "Avi", "13/03/1982", 42);
// console.log(p1.id) - error
console.log(p1.getId());
```

ירושת חוקיות מבנה של מחלקה

implements

- חובה להשתמש בכל המאפיינים הקיימים
- לא ניתן להסיר מאפיינים
- ניתן להוסיף על הקיים
- בקונסטרוקטור מזכור להגדיר את הגישה למאפיינים (public, readonly, protected,)
- כפי שדיברנו קודם. (private)
- במידה ונרצה ליצור מופע חדש שהוגדר במחלקה ספציפית נגדיר למשתנה type של המחלקה.
- במידה ונרצה ליצור מופע חדש שמכיל הגדרות של מספר מחלקות יורשות נגדיר למשתנה type של האינטרפייס הראשי (מבנה מחלקת האב).
- type צריך להיות תואם למחלקה.

דוגמה לכתיבת הקוד:

```
// מחלקה שממשת ממשק מבנה מסוים
interface Shape{
    width: number;
    height: number;
    // חתימה של המתודה
    calcArea: () => number;
}
```



```
//מחלקה יורשת מבנה מסוים
class Squere implements Shape {
  constructor( public width: number, public height: number ){
    this.width = width;
    this.height = height;
  }
  calcArea(): number{
    return this.width * this.height;
  }
}

//מחלקה נוספת שיורשת את אותו המבנה
class Circle implements Shape {
  constructor( public width: number, public height: number, public
radius: number){
    this.width = width;
    this.height = height;
    this.radius = radius;
  }
  calcArea(): number {
    return this.radius ** 2 * Math.PI;
  }
}

//מערך המכיל מופעים של מחלקות שונות שיורשת מאותו מבנה
let x: Shape[] = [
  new Squere(3, 3),
  new Squere(2,2),
  new Circle (1, 2, 3),
  new Circle (2, 2, 4),
];

//מופע של מחלקה ספציפית
let s1: Squere = new Squere (2,3)
console.log(s1.calcArea());

//מופע של מחלקה ספציפית
let c1: Circle = new Circle(1, 2, 3);
console.log(c1.calcArea());
```

abstract class

מחלקה שאנחנו לא יכולים ליצור ממנה אובייקטים ישירות אלא ממחלקות יורשות הסבר:

Class – אפשר ליצור מופע ואי אפשר להוריש

Interface – אי אפשר ליצור מופע אפשר להוריש

Abstract class – אפשר להוריש אי אפשר ליצור מופע



abstract יצירת חותמת מבנה חוקיות.

כאשר נשתמש ב-abstract class ונרצה להוסיף חוקיות של מתודה מסוימת נצטרך להשתמש במתודת abstract.

מתודה לא אבסטרקטית – יורשים אוטומטית בצורה סטטית כמו ב-JS ולא חייב לציין אותה במחלקה היורשת, ניתן לדרוס אותה ע"י הגדרתה בצורה אחרת.

מתודה אבסטרקטית – חייבת להופיע במחלקה היורשת.

ניתן להגדיר בירושה שלה ערך אחר.

במחלקה אבסטרקטית נגדיר את המבנה ולאחר מכן במחלקה היורשת נגדיר את הפעולות שמתודה מבצעת.