פונקציות – Functions

מהי פונקציה

פונקציה בתכנות מאפשרת לנו לבצע פעולות דומות בקוד.

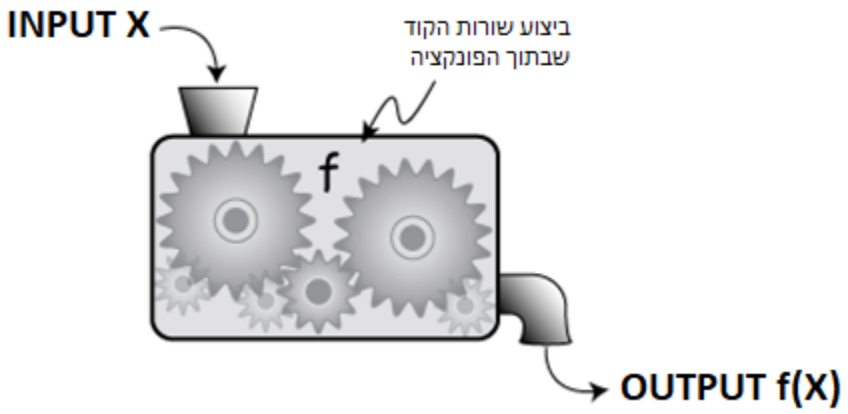
למעשה, פונקציה היא קטע קוד שיכול לפעול מתי שנרצה וכמה שנרצה.

הגדרת פונקציה היא לכתוב שורות קוד במקום אחד תחת שם שנגדיר לפונקציה.

כך נוכל להפעיל את קטע הקוד הזה ע"י קריאה לשם שהגדרנו, מתי שנרצה ומאיפה שנרצה.

- אפשר **להעביר מידע** (Arguments/Parameters) שעליו יבוצעו פעולות, או **שלא להעביר כלום**.
- אפשר **לקבל חזרה ערכים** מהפונקציה (מספר או מחרוזת למשל), או **לא להחזיר כלום**.

במילים אחרות פונקציה היא קטע קוד שניתן שיהיה לו קלט Input ופלט Output .



שימוש בפונקציות

היתרון בשימוש בפונקציה הוא יעילות, חיסכון במקום ושמירה על סדר.

במקום לרשום מחדש את הפקודות שנרצה שהמערכת תבצע שוב מחדש, נוכל לשמור אותם בתוך פונקציה לה נקרא בכל פעם כשנרצה.

כאשר יש שימוש בלולאות רבות, בד"כ ההעדפה היא להכניס כל איטרציה אל תוך פונקציה בפני עצמה.

מבנה הפונקציה

כל פונקציה נפתחת בהוראה **function** (מילה שמורה המגדירה פונקציה) , **שם הפונקציה**, **סוגרים וסוגריים מסולסלים**.

בתוך הסוגריים המסולסלים, אנו רושמים את גוף הפונקציה – קטע הקוד שיפעל.

אם הפונקציה מחזירה ערך אז היא תסתיים בפקודת return.

```
function name(parameters) {
  ...body...
}
```

- **name**- הוא שם הפונקציה אותה מקליד המתכנת.
  - Case-Sensitivity - יש הבדל בין אותיות גדולות וקטנות (לדוגמה, פונקציה בשם f ופונקציה בשם F הן שתי פונקציות שונות).
  - מומלץ להתחיל פונקציה עם אות קטנה, אם נרצה שם ארוך נוכל להיעזר בקו תחתון (_) .
- **Parameters**– ערכים שניתן להעביר לתוך הפונקציה וניתן יהיה להשתמש בהם בגוף הפונקציה.
  - ניתן להעביר ערכים וניתן שלא להעביר כלום. לשיקול המתכנת בהתאם למטרת הפונקציה.
- **Documentation**- מומלץ לכתוב תיעוד של הפונקציה – מה המטרה, מה הפונקציה עושה.
- **Body** - גוף הפונקציה - רצף של פקודות רצופות אותם מקליד המתכנת.
- **return**- משפט שמורה לפונקציה להחזיר ערך מסוים או מספר ערכים לאותו מקום שממנו קראו לפונקציה.
  - return היא מילה שמורה, לאחריה נדרש לכתוב את הערך או ערכים שיוחזר/ו.
  - ניתן להחזיר ערך ע"י המילה return מכל מקום בתוך גוף הפונקציה, יש לזכור שלאחר משפט return , ביצוע קוד הפונקציה נפסק וממשיך מאותו מקום שממנו הפונקציה נקראה.
  - חשוב לציין – בפונקציות שלא מחזירות ערך, לא כותבים משפט return.

קריאה לפונקציה

לאחר שנכתוב את הפונקציה, היא תשמר בזיכרון אך היא לא תפעל בכוחות עצמה.

אם נרצה להשתמש בה, נדרש לקרוא לה וכך היא תפעל.

לשם כך נקליד את שם הפונקציה ואחריו סוגריים ()function_name. לדוגמא:

```
function showMessage() {
  alert( 'Hello everyone!' );
}

showMessage();
showMessage();
```

משתנים מקומיים - Local

משתנה המוצהר בתוך פונקציה "חי ונושם" רק בתוך פונקציה זו.

בסיום הפונקציה המשתנה "נמחק מהזיכרון".

```
Function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!

Alert( message ); // ← Error! The variable is local to the function
```

משתנים חיצוניים

ניתן לגשת מתוך הפונקציה גם למשתנה חיצוני, למשל:

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

כמו כן, מתוך הפונקציה ישנה גישה מלאה למשתנה החיצוני – כולל אפשרות לעדכון.

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function
```

המשתנה החיצוני יהיה בשימוש רק אם אין משתנה מקומי באותו שם.

אם מוכרז משתנה עם אותו שם בתוך הפונקציה אז הוא "מתעלם" מהמשתנה החיצוני.

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable
```

פרמטרים וארגומנטים

כאשר יש לנו פונקציה שמבצעת קוד מסוים, ונרצה להעביר ערכים מחוץ לפונקציה אל תוך הפונקציה נשתמש בארגומנטים. בתוך הסוגריים של הפונקציה, יצוינו רשימת הארגומנטים; רשימה זו יכולה להיות ריקה. ניתן להעביר כל סוג טיפוס שנרצה (int, string, objects, array וכו').

```
function showMessage(from, text) { // arguments: from, text
  alert(from + ': ' + text);
}
let name = "Ann";
showMessage(name, 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

בדוגמה זו ניתן לראות כי מוגדרת פונקציה בשם showMessage ולה 2 ארגומנטים - הראשון בשם from והשני בשם text .

הפעלנו את הפונקציה ע"י קריאתה והעברה של 2 פרמטרים (ערכים) – את המשתנה name שבתוכו המחרוזת "Ann" שיכנס לארגומנט from והערך "Hello" שמוזן לארגומנט text .

בפועל, הפרמטרים שמועברים לפונקציה מועברים אליה בדיוק כמו **השמה רגילה של משתנה**.

חשוב להבין - ההבדל הוא שהמשתנים ה"חדשים" שמוגדר בתוך הפונקציה לא קשורים למשתנים שהעברנו לפונקציה מבחוץ.

דוגמא נוספת:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // make "from" look nicer

  alert( from + ': ' + text );
}

let from = "Ann";

showMessage(from, "Hello"); // *Ann*: Hello

// the value of "from" is the same, the function modified a local
copy
alert( from ); // Ann
```

ברירת מחדל של פרמטרים (parameters default value)

ניתן להגדיר ארגומנט בפונקציה שבמידה ולא מעבירים אליו שום ערך בעת קריאה לפונקציה, הארגומנט יקבל **ערך כברירת מחדל** אותו הגדיר המתכנת בעת כתיבת הפונקציה.

את הפרמטרים של ברירת המחדל כותבים בד"כ בסוף לאחר כתיבת כל הארגומנטים הרגילים בהכרזת הפונקציה. מגדירים שם לפרמטר ולאחר מכן סימן השוואה (=) ולאחריו את הערך שיוזן לפרמטר כברירת מחדל במידה ובקריאה לפונקציה לא ישלח ארגומנט.

במידה והועבר ערך לפרמטר, הערך **שיושם במשתנה** הוא זה **שהועבר**. לדוגמא:

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

בפרמטרים רגילים, במידה ולא יועבר ערך לארגומנט, מה שיהיה במשתנה הוא undefined .

להראות דוגמא נוספת :

```
function showMessage(text) {
  if (text === undefined) {
    text = 'empty message';
  }

  alert(text);
}

showMessage(); // empty message
```

תרגיל – מי יצליח לעשות עוד דרך :

```
// if text parameter is omitted or "" is passed, set it to 'empty'
function showMessage(text) {
  text = text || 'empty';
  ...
}
```

או :

```
// if there's no "count" parameter, show "unknown"
function showCount(count) {
  alert(count ?? "unknown");
}

showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

החזרת ערך – Return value

פונקציה יכולה להחזיר ערך מסוים או מספר ערכים לאותו מקום שממנו קראו לפונקציה.

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert( result ); // 3
```

מיד לאחר פקודת return הפונקציה נעצרת והערך מוחזר.

יתכנו מקרים בהם נכתוב מס' return בפונקציה אחת, לדוגמא:

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

ניתן להשתמש בפקודת return גם **ללא ערך**. במקרה זה הפונקציה תסתיים ויוחזר הערך undefined.

גם במקרה שלא נכתוב פקודת return – הערך ש"יחזור" יהיה undefined.

נושאים חשובים

מתן שמות לפונקציות

פונקציות הן בד"כ פעולות, ולכן נדרש לתת שם לפונקציה שמתאר את הפעולה שנעשית בתוך הפונקציה.

השם צריך להיות קצר, מדויק ככל האפשר ולתאר מה הפונקציה עושה, כך שמי שקורא את הקוד יקבל אינדיקציה לגבי מה שהפונקציה עושה.

```
showMessage(..)     // shows a message
getAge(..)          // returns the age (gets it somehow)
calcSum(..)         // calculates a sum and returns the result
createForm(..)      // creates a form (and usually returns it)
checkPermission(..) // checks a permission, returns true/false
```

פונקציה אחת - פעולה אחת

פונקציה צריכה לעשות בדיוק את מה שנרמז בשמה, לא יותר.

שתי פעולות עצמאיות בדרך כלל ראויות לשתי פונקציות, גם אם בדרך כלל הן נקראות יחד

(במקרה זה נוכל לבצע פונקציה שלישית המכילה שתי קריאות לשתי הפונקציות).

לתת דוגמא : מסך התחברות – בדיקת הרשאות, בדיקת איות, בדיקת סיסמא.

יתרון נוסף לפונקציה קצרה שעושה פעולה מסויימת היא במקרה של באגים – נדע לטפל בקלות ובמהירות.

Function expressions

ב- JavaScript, פונקציה היא סוג מיוחד של ערך.

התחביר בו השתמשנו בעבר נקרא הצהרת פונקציות:

```
function sayHi() {
  alert( "Hello" );
}
```

יש תחביר נוסף ליצירת פונקציה הנקראת function expression.

```
let sayHi = function() {
  alert( "Hello" );
};
```

כאן, הפונקציה נוצרת ומוקצה למשתנה באופן מפורש, כמו כל ערך אחר.

לא משנה כיצד מוגדרת הפונקציה, זהו רק ערך המאוחסן במשתנה sayHi.

המשמעות של דוגמאות קוד אלה זהה: "צור פונקציה והכניס אותה למשתנה sayHi".

אנו יכולים אפילו להדפיס את הערך:

```
function sayHi() {
  alert( "Hello" );
}

alert( sayHi ); // shows the function code
```

שימו לב שהשורה האחרונה לא מפעילה את הפונקציה, מכיוון שאין סוגריים אחרי sayHi.

ישנן שפות תכנות בהן כל אזכור של שם פונקציה גורם לביצועו, אך JavaScript אינו כזה.

ב- JavaScript פונקציה היא ערך, ולכן נוכל להתמודד איתה כערך.

אין ספק, פונקציה היא ערך מיוחד, במובן שאנחנו יכולים לקרוא לזה כמו sayHi().

אבל זה עדיין ערך. כדי שנוכל לעבוד עם זה כמו עם סוגים אחרים של ערכים.

אנו יכולים להעתיק פונקציה למשתנה אחר:

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

להסביר מה שקורה פה שורה אחר שורה.

Callback functions

פונקציות callback מאד שימושיות ומהוות דוגמא נוספת להעברת פונקציות כערכים ושימוש ב function expression.

נכתוב פונקציה ask(question, yes, no) עם שלושה פרמטרים:

question - טקסט השאלה

yes - פונקציה שתופעל אם התשובה היא "כן"

no - פונקציה שתופעל אם התשובה היא "לא"

הפונקציה תשאל את השאלה ובהתאם לתשובת המשתמש תפעיל את פונקציה yes() או no() :

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

function showCancel() {
  alert( "You canceled the execution." );
}

ask("Do you agree?", showOk, showCancel);
```

הארגומנטים showOk ו- showCancel של ask נקראים פונקציות callback או סתם callbacks.

הרעיון הוא שאנו מעבירים פונקציה ומצפים שהיא "call back" מאוחר יותר במידת הצורך.

במקרה שלנו, showOk הופך ל-callback לתשובה "yes", ו- showCancel לתשובה "no".

אנו יכולים להשתמש ב- function expression כדי לכתוב את אותה פונקציה בצורה קצרה בהרבה:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

להסביר שהפונקציות מוכרזות ממש בתוך הקריאה של ask(...), אין להם שם, ולכן הם נקראים אנונימיים.

פונקציות כאלה אינן נגישות מחוץ ל-ask (מכיוון שהן לא מוקצות למשתנים), אבל זה בדיוק מה שאנחנו צריכים כאן.

Function Expression vs Function Declaration

- **התחביר**

Function declaration: פונקציה, המוצהרת כהצהרה נפרדת, בזרימת הקוד הראשית.

Function expression: פונקציה, שנוצרה בתוך ביטוי או בתוך מבנה תחבירי אחר.

```
// Function Declaration
function sum(a, b) {
  return a + b;
      }
```

vs

```
// Function Expression
let sum = function(a, b) {
  return a + b;
      };
```

- **יצירת הפונקציה**

Function expression: פונקציה נוצרת כאשר הביצוע מגיע אליה והוא שמישה רק מאותו הרגע.

Function declaration: ניתן לקרוא לפונקציה עוד לפני שהיא מוגדרת.

פונקציה מוצהרת הינה גלובלית ונראית בכל הקוד, לא משנה היכן היא נמצאת.

זה נובע "מאלגוריתמים פנימיים" של JavaScript – שכאשר JS מתכונן להפעיל את הקוד, תחילה הוא מחפש בו פונקציות מוצהרות גלובליות ויוצר את הפונקציות.

ניתן לחשוב על זה כעל "שלב אתחול". ולאחר עיבוד כל הצהרות הפונקציות, הקוד מבוצע.

ולכן יש לו גישה לפונקציות האלה.

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
      }
```

ב – Function expression זה לא יעבוד

```
sayHi("John"); // error!

let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
      };
```

פונקציות החץ – Arrow Function

תחביר נוסף פשוט מאוד ותמציתי ליצירת פונקציות, שהוא לרוב טוב יותר מ- function expression. זה נקרא "פונקציות חץ" :

```
let func = (arg1, arg2, ..., argN) => expression
```

הקוד יוצר פונקציית func שמקבלת ארגומנטים arg1..argN, ואז מבצעת את הביטוי/הקוד בצד ימין תוך כדי השימוש באותם ארגומנטים ומחזירה את התוצאה .

במילים אחרות, זו הגרסה הקצרה יותר של:

```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

דוגמא מוחשית יותר:

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

להסביר את הקוד וההבדלים.

אם יש לנו רק ארגומנט אחד, ניתן להשמיט סוגריים סביב הפרמטרים, מה שהופך לקצר עוד יותר.

```
let double = n => n * 2;
// roughly the same as: let double = function(n) { return n * 2 }

alert( double(3) ); // 6
```

אם אין לנו ארגומנטים כלל, נדרש להשאיר את הסוגריים ריקים

```
let sayHi = () => alert("Hello!");

sayHi();
```

ניתן להשתמש בפונקציות חץ באותו אופן כמו Function expression.

לדוגמה, כדי ליצור פונקציה באופן דינמי:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");

welcome();
```

פונקציות חץ עשויות להיראות לא מוכרות ולא קריאות במיוחד בהתחלה, אך זה משתנה במהירות ככל שהעיניים מתרגלות למבנה.

הן נוחות מאוד לפעולות פשוטות בשורה אחת, כשאנחנו עצלנים לכתוב מילות קוד רבות.

multiline arrow functions

לפעמים אנו זקוקים למשהו מורכב מעט יותר, כמו ביטויים מרובים או הצהרות.

זה גם אפשרי, אך עלינו לסגור אותם בסוגרים מסולסלים.

```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit
"return"
};

alert( sum(1, 2) ); // 3
```