ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΚΕΝΤΡΟ ΕΠΙΜΟΡΦΩΣΗΣ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗΣ

Typescript και Angular

Εισαγωγή στη γλώσσα Typescript

Χριστόδουλος Φραγκουδάκης

Javascript και τύποι δεδομένων

- Στη JavaScript οι μεταβλητές δεν έχουν τύπους και σε κάθε μεταβλητή μπορούμε να αναθέσουμε (και να αναθέσουμε ξανά) τιμές οποιουδήποτε τύπου

```
let foo = 42; // foo is now a number
foo = "bar"; // foo is now a string
foo = true; // foo is now a boolean
```

- Πρόκειται για επιλογή σχεδιασμού της JavaScript, καθώς είχε σκοπό να εξυπηρετεί μαζί τους προγραμματιστές και σχεδιαστές προσθέτοντας λειτουργικότητα στις σελίδες
- Είναι όμως εύκολο να εισάγονται λογικά προγραμματιστικά λάθη κατά την εκτέλεση του κώδικα που έχουν σχέση με τον έλεγχο των τύπων

Λογικό λάθος στην εκτέλεση

- Η παρακάτω συνάρτηση δέχεται τρεις παραμέτρους και επιστρέφει μια συμβολοσειρά

```
const personDescription = (name, city, age) =>
  `${name} lives in ${city}. he's ${age}. In 10 years he'll be ${age + 10}`;
```

- Αν κληθεί όπως παρακάτω δεν υπάρχει πρόβλημα

```
console.log(personDescription("Germán", "Buenos Aires", 29));
// Τυπώνει Germán lives in Buenos Aires. he's 29. In 10 years he'll be 39.
```

- Όμως αν από λάθος εισάγουμε μια συμβολοσειρά σαν τρίτο όρισμα τότε υπάρχει πρόβλημα

```
console.log(personDescription("Germán", "Buenos Aires", "29"));
// Τυπώνει Germán lives in Buenos Aires. he's 29. In 10 years he'll be 2910.
```

Λογικό λάθος στην εκτέλεση

```
console.log(personDescription("Germán", "Buenos Aires", "29"));
// Τυπώνει Germán lives in Buenos Aires. he's 29. In 10 years he'll be 2910.
```

- Η JavaScript δεν αναφέρει κάποιο λάθος γιατί δεν έχει τρόπο να γνωρίζει τίποτα σχετικά με τον τύπο των δεδομένων που δέχεται η συνάρτηση
- Είναι εύκολο να υπάρξουν τέτοια λάθη, κυρίως σε προγράμματα με πολύ κώδικα, ειδικά αν δεν είμαστε εξοικειωμένοι με τους τύπους των παραμέτρων των συναρτήσεων ή των API
- Η Typescript αντιμετωπίζει αυτό ακριβώς το πρόβλημα

```
const personDescription = (name: string, city: string, age: number) =>
  `${name} lives in ${city}. he's ${age}. In 10 years he'll be ${age + 10}`;

console.log(personDescription("Germán", "Buenos Aires", "29"));
// error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

Typescript

- Δημιουργήθηκε το 2012 και συντηρείται από τη Microsoft
- Οι μεταβλητές της γλώσσας είναι απαραίτητο να έχουν τύπους όπως στη Java ή τη C#
- Η δήλωση των τύπων των μεταβλητών εξασφαλίζει τον έλεγχο στις μετέπειτα αναθέσεις τιμών στον κώδικα
- Αν δεν υπάρχει συμβατότητα των τιμών και των τύπων των μεταβλητών *τότε η Typescript δεν μετατρέπεται (transpile) σε Javascript*
- Τελικά παράγεται ένα αρχείο Javascript που εκτελείται είτε στον browser, είτε με το Node.js
- Εγκατάσταση:

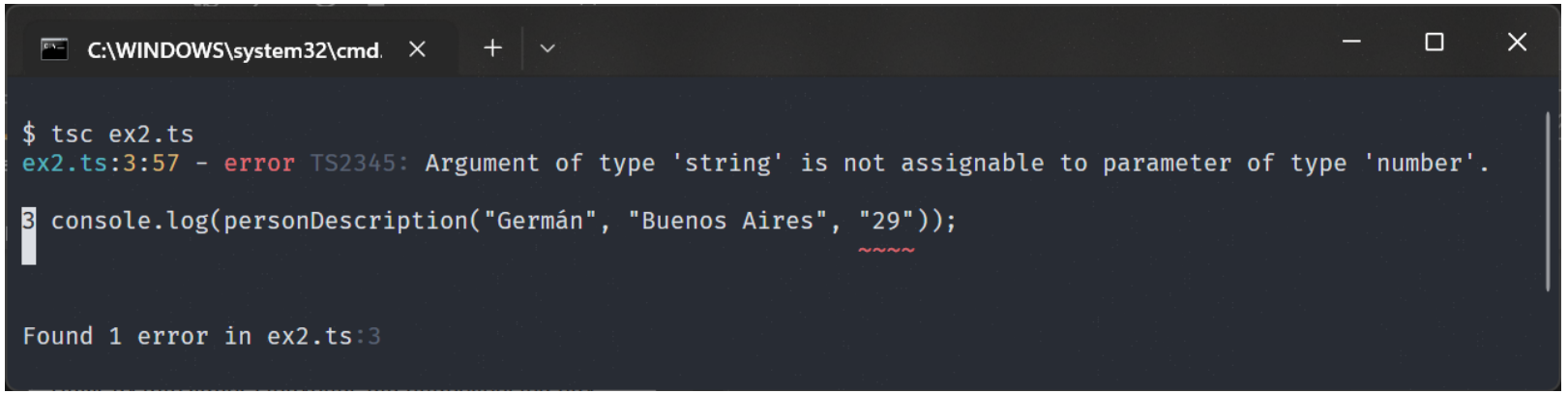
```
npm i -g typescript
```

Typescript σε χρήση

- Η προηγούμενη συνάρτηση σε Typescript

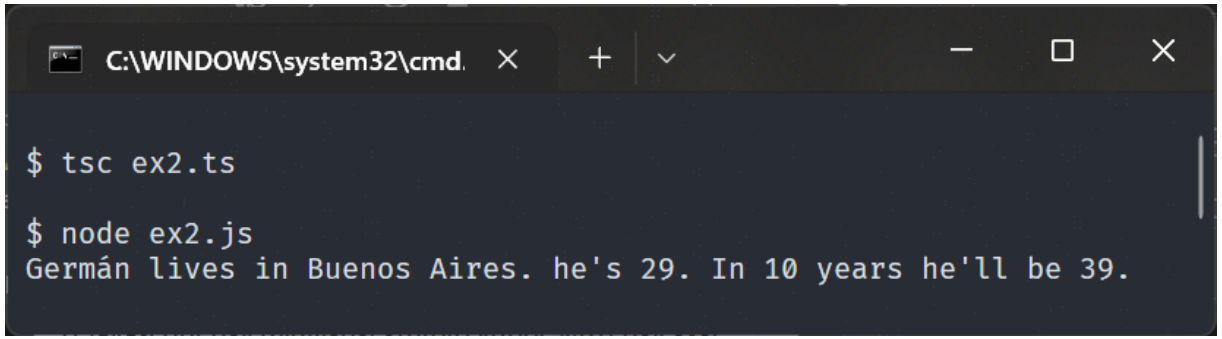
```ts
const personDescription = (name: string, city: string, age: number) =>
  `${name} lives in ${city}. he's ${age}. In 10 years he'll be ${age + 10}`;

console.log(personDescription("Germán", "Buenos Aires", "29"));
// error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

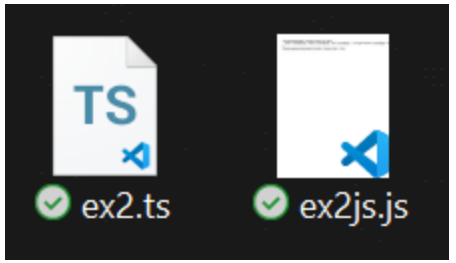
- Το transpile σε Javascript γίνεται με χρήση του `tsc`:



Typescript σε χρήση

- Ο έλεγχος στον τύπο δεν επέτρεψε τη δημιουργία του αρχείου ex2.js που θα ήταν επιδεκτικό στο λογικό λάθος εκτέλεσης με πέρασμα λάθος τύπου στην τρίτη παράμετρο
- Αν διορθωθεί το λάθος τότε το transpile ολοκληρώνεται





- Το αρχείο `ex2.js` που προέκυψε

```
var personDescription = function (name, city, age) {
    return "".concat(name, " lives in ").concat(city, ". he's ").concat(age, ". In 10 years he'll be ").concat(age + 10, ".");
};
console.log(personDescription("Germán", "Buenos Aires", 29));
```

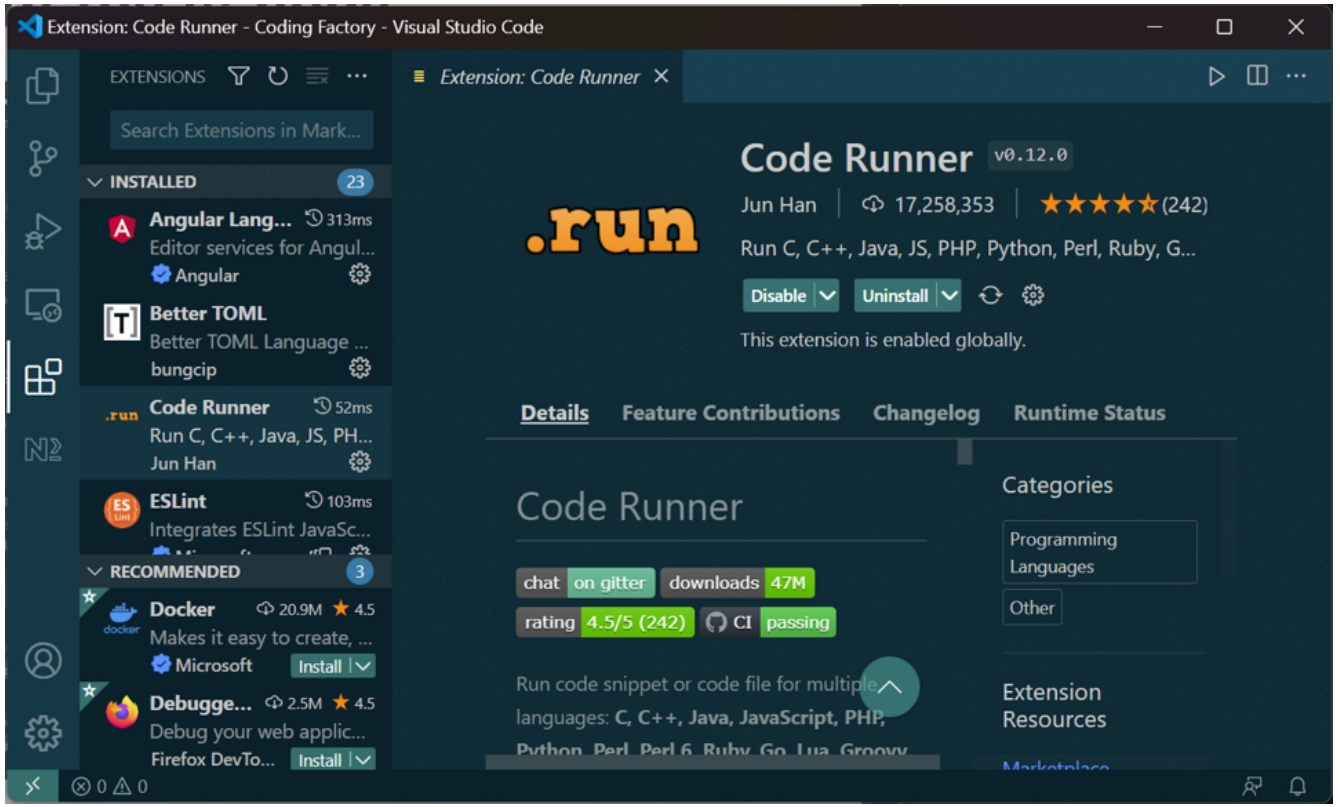
Χρήση του `ts-node`

- Το `ts-node` επιτρέπει την εκτέλεση του προγράμματος Javascript, που προκύπτει από ένα συντακτικά ορθό αρχείο Typescript, με μία μόνο εντολή

```
C:\WINDOWS\system32\cmd.

$ npm i -g ts-node

changed 19 packages, and audited 20 packages in 2s

found 0 vulnerabilities

$ ts-node ex2.ts
C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:
859
    return new TSError(diagnosticText, diagnosticCodes, diagnostics);
           ^
TSError: ⨯ Unable to compile TypeScript:
ex2.ts:3:57 - error TS2345: Argument of type 'string' is not assignable to pa
rameter of type 'number'.

3 console.log(personDescription("Germán", "Buenos Aires", "29"));
                                                          ~~~~
```

Code Runner και `ts-node`

- Το `ts-node` χρησιμοποιείται από το extension Code Runner του Visual Studio Code



Το Code Runner σε χρήση

ex2.ts - Coding Factory - Visual Studio Code

ex2.ts 1

2023 > Typescript > ex2.ts > ...

```
1  const personDescription = (name: string, city: string, age: number) =>
2    `${name} lives in ${city}. he's ${age}. In 10 years he'll be ${age + 10}.`;
3  console.log(personDescription("Germán", "Buenos Aires", "29"));
4
```

PROBLEMS 1 SEARCH OUTPUT DEBUG CONSOLE TERMINAL Code

```
[Running] ts-node "c:\Users\christodoulos\OneDrive - Εθνικό Μετσόβιο Πολυτεχνείο\Documents\Coding
Factory\2023\Typescript\ex2.ts"
C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:859
    return new TSError(diagnosticText, diagnosticCodes, diagnostics);
           ^
TSError: ⨯ Unable to compile TypeScript:
2023/Typescript/ex2.ts(3,57): error TS2345: Argument of type 'string' is not assignable to parameter of type
'number'.

    at createTSError (C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:859:12)
    at reportTSError (C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:863:19)
    at getOutput (C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:1077:36)
    at Object.compile (C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:1433:41)
    at Module.m._compile (C:\Users\christodoulos\AppData\Roaming\npm\node_modules\ts-node\src\index.ts:1617:30)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
```

⊗ 1 ⚠ 0 Ln 3, Col 61 Spaces: 2 UTF-8 CRLF {} TypeScript ✓ Prettier

Το Code Runner σε χρήση

ex2.ts - Coding Factory - Visual Studio Code

ex2.ts

2023 > Typescript > ex2.ts > ...

```ts
const personDescription = (name: string, city: string, age: number) =>
  `${name} lives in ${city}. he's ${age}. In 10 years he'll be ${age + 10}.`;
console.log(personDescription("Germán", "Buenos Aires", 29));

```

PROBLEMS SEARCH OUTPUT DEBUG CONSOLE TERMINAL

Code

```
[Running] ts-node "c:\Users\christodoulos\OneDrive - Εθνικό Μετσόβιο Πολυτεχνείο\Documents\Coding Factory\2023\Typescript\ex2.ts"
Germán lives in Buenos Aires. he's 29. In 10 years he'll be 39.

[Done] exited with code=0 in 1.045 seconds
```

0 0 Ln 3, Col 62 Spaces: 2 UTF-8 CRLF {} TypeScript ✓ Prettier

Typescript → Javascript

- Η ανάπτυξη κώδικα σε Typescript ενσωματώνει δικλείδες ασφάλειας για την *παραγωγή στιβαρού κώδικα Javascript* που δεν είναι επιρρεπής σε λογικά λάθη εκτέλεσης σχετικά με τους τύπους των δεδομένων
- *Τελικά εκτελείται ο κώδικας της Javascript*
- Η χρήση του ts-node κάνει τη διαδικασία της μετατροπής σε Javascript *διάφανη* για το χρήστη (μετά την εκτέλεση δεν παραμένει το αρχείο της Javascript).

Τα παραδείγματα στην τάξη θα βρίσκονται όλα μαζί σε ένα κατάλογο. Όμως ο transpiler της Typescript θεωρεί τότε πως ο κώδικας όλων των αρχείων ανήκει στο ίδιο global scope. Αυτό ενδεχομένως μπορεί να προκαλέσει συγκρούσεις στα ονόματα μεταβλητών, κλάσεων κτλ. Για να αποφύγουμε τα προβλήματα με τις συγκρούσεις στα ονόματα προσθέτουμε στην αρχή ένα `export {}` *και έτσι κάθε αρχείο έχει το δικό του scope χωρίς να επηρεάζει το global scope δημιουργόντας συγκρούσεις ονομάτων.*

Το αρχείο `tsconfig.json`

Το αρχείο `tsconfig.json` περιέχει τις παραμέτρους που ρυθμίζουν τη συμπεριφορά του Typescript transpiler και καθορίζουν τις ιδιότητες του *παραγόμενου* κώδικα σε Javascript. Για να αναγκαστούμε να χρησιμοποιούμε *βέλτιστες πρακτικές* συμφωνούμε να χρησιμοποιούμε *τουλάχιστον το* `"strict":true`.

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "sourceMap": true
  }
}
```

- `target` είναι το πρότυπο που θα ακολουθεί η παραγόμενη Javascript
- `module` είναι το σύστημα που θα χρησιμοποιηθεί για τις ενότητες κώδικα (χρήση `require` ή `import`)
- `strict` επιβάλλει ολοκληρωτικά την αυστηρότητα των τύπων
- `sourceMap` χρήσιμο στο debugging (δείχνει το ενδεχόμενο λάθος στην Typescript και όχι στην Javascript)

Τύπος δεδομένων `boolean`

Βασικός τύπος από τη Javascript

```
let isReady: boolean = true;
let isLoggedIn: boolean = false;

...

let isAdult: boolean = true;
let hasLicense: boolean = false;

if (isAdult && hasLicense) {
  console.log("You can drive.");
} else {
  console.log("You can't drive.");
}

const canDrive = () => isAdult && hasLicense
if (canDrive()) {
  console.log("You can drive.")
}
```

```
let isLightOn: boolean = true;

if (isLightOn) {
  console.log("The light is on.");
} else {
  console.log("The light is off.");
}

...

function hasPermission(role: boolean): void {
  if (role) {
    console.log("Access granted.");
  } else {
    console.log("Access denied.");
  }
}

hasPermission(true); // Τυπώνει Access granted.
hasPermission(false); // Τυπώνει Access denied.
```

Τύπος δεδομένων `number`

Στην Typescript ο τύπος `number` χρησιμοποιείται και για τους ακέραιους και για τους αριθμούς κινητής υποδιαστολής. Μερικά παραδείγματα χρήσης είναι τα παρακάτω:

Δήλωση μεταβλητών

```
let age: number = 30;
let price: number = 4.99;
```

Χρήση σε αριθμητικές εκφράσεις

```
let x: number = 10;
let y: number = 5;
console.log(x + y); // Τυπώνει 15
console.log(x - y); // Τυπώνει 5
console.log(x * y); // Τυπώνει 50
console.log(x / y); // Τυπώνει 2
```

Έλεγχος αν μια τιμή είναι αριθμός

```
let c: number = NaN;
let d: number = 10;

console.log(isNaN(c)); // Τυπώνει true
console.log(isNaN(d)); // Τυπώνει false
```

Άλλες βάσεις αρίθμησης

```
let binary: number = 0b1010; // binary literal for 10
let octal: number = 0o12; // octal literal for 10
let hex: number = 0xa; // hexadecimal literal for 10
```

Τύπος δεδομένων `number`

Χρήση με τους τελεστές σύγκρισης

```
let a: number = 10;
let b: number = 5;

console.log(a > b); // Τυπώνει true
console.log(a < b); // Τυπώνει false
console.log(a >= b); // Τυπώνει true
console.log(a <= b); // Τυπώνει false
console.log(a === b); // Τυπώνει false
console.log(a !== b); // Τυπώνει true
```

Χρήση του αντικειμένου `Number`

```
let e: number = 10;

console.log(Number.MAX_VALUE);
// Τυπώνει 1.7976931348623157e+308
console.log(Number.MIN_VALUE);
// Τυπώνει 5e-324
console.log(Number.isFinite(e));
// Τυπώνει true
console.log(Number.isInteger(e));
// Τυπώνει true
```

Τύπος δεδομένων `string`

Δήλωση μεταβλητών

```
let name: string = "John Doe";
let message: string = "Hello, world!";
```

Παρεμβολή μεταβλητών

```
let age: number = 30;

console.log(`I am ${age} years old.`);
// Τυπώνει I am 30 years old.
```

Χρήση backticks για multi-line strings

```
let poem: string = `
  Roses are red,
  Violets are blue,
  Sugar is sweet,
  And so are you.
`;
```

Regular expressions

```
let text: string = "The quick brown fox";
let pattern: RegExp = /brown/;

console.log(pattern.test(text));
// Τυπώνει true
console.log(text.match(pattern));
// Τυπώνει [ 'brown', index: 10, input: 'The quick brown fox']
```

Τύπος δεδομένων `Array`

Στην Typescript οι πίνακες είναι συλλογές δεδομένων *του ίδιου τύπου*

Δήλωση πίνακα

```
let numbers: number[] = [1, 2, 3, 4, 5];
let colors: string[] = ["red", "green", "blue"];
```

Πρόσβαση στα στοιχεία του πίνακα

```
let numbers: number[] = [1, 2, 3, 4, 5];
console.log(numbers[0]); // Τυπώνει 1
console.log(numbers[2]); // Τυπώνει 3
```

Διάσχιση

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

// Πιο ευανάγνωστο, όμως χωρίς index

for (let num of numbers) {
  console.log(num);
}

// Αν χρειαζόμαστε το index της θέσης

for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
```

Τύπος δεδομένων `Array`

Προσθήκη και διαγραφή στοιχείων

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

numbers.push(4);
numbers.push(5, 6);

console.log(numbers);
// Τυπώνει [1, 2, 3, 4, 5, 6]

numbers.pop();
numbers.splice(1, 2);

console.log(numbers);
// Τυπώνει [1, 4, 5]
```

Χρήση των `filter`, `map`, `reduce`

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

let evenNumbers = numbers.filter((n) => n % 2 === 0);
let doubledNumbers = numbers.map((n) => n * 2);
let sum = numbers.reduce((a, b) => a + b, 0);

console.log(evenNumbers);
// Τυπώνει [2, 4]
console.log(doubledNumbers);
// Τυπώνει [2, 4, 6, 8, 10]
console.log(sum);
// Τυπώνει 15
```

Τύπος δεδομένων `enum`

Στην Typescript ο τύπος δεδομένων `enum` χρησιμοποιείται για τον ορισμό συνόλων με ονοματισμένες σταθερές αντί "μη φιλικών" αριθμητικών ή άλλων τιμών

Δήλωση `enum`

```
enum Color {
  Red,
  Green,
  Blue,
}

let myColor: Color = Color.Green;
console.log(myColor); // Τυπώνει 1
```

- **Στο παράδειγμα δηλώνεται ένας τύπος `enum` με όνομα `Color` και τρεις ονοματισμένες σταθερές `Red`, `Green` και `Blue`**
- **Εξ ορισμού, στις σταθερές αναθέτονται ακέραιες τιμές που ξεκινούν από το 0**
- **Η μεταβλητή `myColor` εχει την τιμή `Color.Green`, δεύτερη στη σειρά των ορισμών, που αντιστοιχεί στην τιμή `1`**

Τύπος δεδομένων `enum`

Χρήση συγκεκριμένων τιμών στις σταθερές

```
enum Color {
  Red = 1,
  Green = 2,
  Blue = 4,
}

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

Πιο εκφραστικό `switch`

```
enum Direction { Up, Down }

let direction: Direction = Direction.Up;

switch (direction) {
  case Direction.Up:
    console.log("Going up.");
    break;
  case Direction.Down:
    console.log("Going down.");
    break;
  default:
    console.log("Unknown direction.");
}
```

Συναγόμενοι (inferred) τύποι δεδομένων

Στην Typescript, αν δεν ορίζονται οι τύποι δεδομένων, τότε συνάγονται από την τιμή της αρχικοποίησής τους. Στο παράδειγμα ο συναγόμενος τύπος της `name` είναι `string`

```
let name = "John";
```

Συναγόμενοι τύποι υπάρχουν και στις τιμές που επιστρέφουν οι συναρτήσεις, αν δεν επιστρέφουν τιμή συνάγεται ο τύπος `void`

```
function add(a: number, b: number) {
  return a + b;
}
let result = add(2, 3);
```

Αφού ο τύπος των παραμέτρων είναι `number` και επιστρέφεται το αποτέλεσμα του τελεστή `+` ο τύπος της `add` συνάγεται πως είναι `number`, όμοια και του `result`

Ένωση (union) τύπων δεδομένων

Στην Typescript η ένωση τύπων επιτρέπει τη χρήση τιμών από διαφορετικούς τύπους δεδομένων. Χρησιμοποιούμε το `|` (pipe) ανάμεσα δύο ή περισσότερους τύπους

```
let myNumber: number | string = 123;
myNumber = "456";
```

Είναι χρήσιμη στις περιπτώσεις που οι συναρτήσεις δέχονται παραμέτρους διαφόρων τύπων και αντίστοιχα επιστρέφουν τιμές διαφόρων τύπων

Αντίστοιχα χρησιμεύει και στις περιπτώσεις των αντικειμένων που τα χαρακτηριστικά τους ενδέχεται να είναι διάφορων τύπων

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

let person: Person = {
  name: "John",
  age: 30,
};
```

Τομή (intersection) τύπων δεδομένων

- Χρησιμοποιούμε το `&` ανάμεσα σε δύο ή περισσότερους τύπους αντικειμένων για να δημιουργήσουμε ένα τύπο με όλες τις μεθόδους και τα χαρακτηριστικά από κάθε τύπο
- Είναι χρήσιμη όταν πρέπει να συνδυαστούν πολλαπλοί τύποι, όταν μια συνάρτηση λαμβάνει αντικείμενα με χαρακτηριστικά από πολλαπλά interfaces ή όταν μια κλάση επεκτείνει (extends) πολλαπλές κλάσεις

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

interface Employee {
  company: string;
  role: string;
}

type PersonAndEmployee = Person & Employee;

let personAndEmployee: PersonAndEmployee = {
  name: "John",
  age: 30,
  company: "Acme Inc.",
  role: "Manager",
};
```

Προσαρμοσμένοι τύποι δεδομένων

Έχουμε τη δυνατότητα να ορίσουμε προσαρμοσμένους (custom) τύπους δεδομένων που είτε θα είναι ψευδώνυμα άλλων τύπων ή θα αποτελούν συνδυασμούς άλλων τύπων

```
type UserId = number | string;
let id: UserId = 123;
```

Ο τύπος `UserId` είναι ένα ψευδώνυμο της έννοιας `number` ή `string`

Ο τύπος `PersonOrEmployee` περιγράφει την έννοια της *ένωσης* των δύο τύπων, `Person` *ή* `Employee`, και περνά σαν τύπος της παραμέτου της συνάρτησης `printName`

```
type Person = {
  name: string;
  age: number;
};

type Employee = {
  name: string;
  jobTitle: string;
};

type PersonOrEmployee = Person | Employee;

function printName(personOrEmployee: PersonOrEmployee) {
  console.log(personOrEmployee.name);
}
```

Διεπαφές (interfaces) αντικειμένων

Στην Typescript μια διεπαφή είναι ένα "συμβόλαιο" που περιγράφει τα χαρακτηριστικά και τις μεθόδους ενός αντικειμένου. Ένα στιγμιότυπο αντικειμένου "υλοποιεί το συμβόλαιο" ορίζοντας ακριβώς τα χαρακτηριστικά και τις μεθόδους που περιγράφει η διεπαφή.

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

let person: Person = {
  name: "John",
  age: 30,
};
```

- **Στο παράδειγμα η διεπαφή `Person` ορίζει δύο χαρακτηριστικά `name` και `age` με τύπους `string` και `number` αντίστοιχα**
- **Η μεταβλητή `Person` *οφείλει* να υλοποιήσει ακριβώς αυτά τα χαρακτηριστικά με τα ίδια ονόματα και τους αντίστοιχους τύπους**

Διεπαφές (interfaces) αντικειμένων

- Οι διεπαφές μπορούν να ορίζουν τις μεθόδους που πρέπει να υλοποιούν οι κλάσεις που υλοποιούν τη διεπαφή
- Στο παράδειγμα η κλάση `Square` που υλοποιεί τη διεπαφή `Shape` οφείλει να υλοποιήσει μια μέθοδο `getArea` που θα επιστρέφει ένα αριθμό
- Η κλάση δέχεται την παράμετρο `sideLength` που χρησιμοποιεί η `getArea` για να επιστρέψει τον αριθμό

```
interface Shape {
  getArea(): number;
}

class Square implements Shape {
  constructor(private sideLength: number) {}

  getArea(): number {
    return this.sideLength ** 2;
  }
}

let square: Square = new Square(5);
console.log(square.getArea()); // logs 25
```

Προσαρμοσμένοι Τύποι vs Διεπαφές

Διεπαφές

- Ορίζουν ένα "συμβόλαιο" που περιγράφει τα χαρακτηριστικά και τις μεθόδους ενός αντικειμένου
- Μπορούν να επεκείνονται (extend) και να περιέχουν προαιρετικά ή μόνο για διάβασμα (readonly) χαρακτηριστικά.
- Δεν μπορούν να χρησιμοποιηθούν για να δημιουργήσουν ψευδώνυμα πρωταρχικών τύπων ή τομές ή ενώσεις τύπων

Προσαρμοσμένοι τύποι

- Μπορούν να δημιουργήσουν ψευδώνυμα πρωταρχικών τύπων ή τομές ή ενώσεις τύπων
- Οι τύποι μπορεί να είναι γενικοί (generic types)
- Μπορεί να περιγράψουν τύπους αντιστοίχισης (mapped) και τύπους υπό όρους (conditional)

Ισχυρισμός τύπου (type casting)

Με τον ισχυρισμό τύπου δηλώνουμε στον transpiler πως γνωρίζουμε περισσότερο από εκείνον για τον τύπο των δεδομένων και τον εξαναγκάζουμε να αλλάξει τον τύπο

- **Ο ισχυρισμός γίνεται είτε με τη χρήση του `as` είτε με τη χρήση του `<T>` όπου `T` είναι ο τύπος που ισχυριζόμαστε**
- **Στο παράδειγμα ο τύπος `any` (θα αναφερθούμε αργότερα) απενεργοποιεί τον έλεγχο τύπων της Typescript**
- **Άρα η μεταβλητή `value` είναι οποιουδήποτε τύπου**

```
let value: any = "hello";
let strLength1 = (value as string).length;
let strLength2 = (<string>value).length;
```

- **Η μεταβλητή `strLength1` ισχυρίζεται τον τύπο της `value` με τη χρήση του `as`**
- **Η μεταβλητή `strLength2` ισχυρίζεται τον τύπο της `value` με χρήση του `<string>`**

Ισχυρισμός τύπου (type casting)

- Ο ισχυρισμός τύπου χρησιμοποιείται και στις διεπαφές ή τις κλάσεις
- Στο παράδειγμα ορίζουμε τη διεπαφή `Person` και στη συνέχεια τη μεταβλητή `person` που είναι οποιουδήποτε τύπου
- Στη συνέχεια ορίζουμε τη μεταβλητή `personInfo` να έχει την τιμή της `person` με τον τον ισχυρισμό πως ο τύπος της είναι αυτός της διεπαφής `Person`

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

let person: any = {
  name: "Alice",
  age: 30,
};

let personInfo = person as Person;
console.log(personInfo.name); // prints "Alice"
console.log(personInfo.age); // prints 30
```

Γενικοί τύποι (generics)

- Οι γενικοί τύποι στην Typescript είναι ένας τρόπος να παραχθεί επαναχρησιμοποιήσιμος κώδικας που μπορεί να λειτουργεί με οποιοδήποτε τύπο δεδομένων.
- Στο παράδειγμα η διεπαφή `List` λαμβάνει σαν παράμετρο τον τύπο `T` και καθορίζει τις μεθόδους `add` και `get` που εξαρτώνται από τον τύπο `T`
- Η κλάση `stringList` εξειδικεύει την υλοποίηση με τον τύπο `string`

```
interface List<T> {
  add(item: T): void;
  get(index: number): T;
}
class StringList implements List<string> {
  private items: string[] = [];
  add(item: string) {
    this.items.push(item);
  }
  get(index: number): string {
    return this.items[index];
  }
}
let list = new StringList();
list.add("hello");
list.add("world");
console.log(list.get(0)); // logs "hello"
console.log(list.get(1)); // logs "world"
```

Γενικοί τύποι (generics)

- Στο παράδειγμα έχουμε μια γενική συνάρτηση που παίρνει σαν όρισμα ένα πίνακα τιμών οποιουδήποτε τύπου και επιστρέφει ένα πίνακα με τις τιμές συγκεκριμένου τύπου (`string`)
- Η συνάρτηση χρησιμοποιεί τη μέθοδο `filter` για να ξεχωρίσει μόνο τις τιμές που είναι του δωσμένου τύπου `type`
- Χρησιμοποιεί τον τελεστή `typeof`

```
function filterByType<T>(arr: T[], type: string): T[] {
  return arr.filter((value) => typeof value === type);
}

let arr = [1, "hello", true, 2, "world", false];
let filteredArr = filterByType(arr, "string");
// επιστρέφει ["hello", "world"]
```

- Η μεταβλητή `arr` περιέχει ένα πίνακα με τιμές διαφορετικών τύπων
- Χρησιμοποιούμε την `filterByType` για να ξεχωρίσουμε τις τιμές τύπου `string`

Τύποι αντιστοίχισης

- Οι τύποι αντιστοίχισης μετατρέπουν τον τύπο ενός αντικειμένου σε άλλο τύπο με τα ίδια ονόματα χαρακτηριστικών με ενδεχομένως διαφορετικούς τύπους τιμών
- Χρησιμοποιούμε το `keyof` για να διατρέξουμε τα ονόματα των χαρακτηριστικών του αρχικού αντικειμένου και στη συνέχεια μετασχηματίζουμε την τιμή

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

type ReadonlyPerson = {
  readonly [K in keyof Person]: Person[K];
};

let person: ReadonlyPerson = { name: "John", age: 30 };

person.name = "Jane"; // error στο transpile
```

Στο παράδειγμα προστίθεται το `readonly` σε όλα τα χαρακτηριστικά της διεπαφής `Person`

Τύποι αντιστοίχισης

- Στο παράδειγμα ορίζουμε μια διεπαφή `Person` με τρια χαρακτηριστικά `name`, `age` και `address`
- Ο τύπος `UnionOfProperties` διασχίζει τα ονόματα των χαρακτηριστικών (κλειδιά) του τύπου `T` και δημιουργεί ένα νέο τύπο με την ένωση των κλειδιών του `T`.
- Ο τύπος `PersonProps` είναι είτε `string` είτε `number` είτε `{street: string; city: string; country: string;}`

```
interface Person {
  name: string;
  age: number;
  address: {
    street: string;
    city: string;
    country: string;
  };
}
type UnionOfProperties<T> = T[keyof T];
type PersonProps = UnionOfProperties<Person>;

const personName: PersonProps = "John";
const personAge: PersonProps = 30;
const personAddress: PersonProps = {
  street: "street",
  city: "city",
  country: "country",
};
```

Τύποι υπό όρους

- Ορίζονται με τη χρήση του `extends` για τον έλεγχο αν ένας τύπος επεκτείνει άλλο τύπο και μετά μετασχηματίζουν τον τύπο
- Στο πρώτο παράδειγμα ο τύπος `IsString` είναι ο τύπος `true` αν ο γενικός τύπος `T` επεκτείνει τον τύπο `string`
- Στο δεύτερο παράδειγμα αν ο γενικός τύπος `T` είναι ο τύπος `true` τότε ο τύπος `IfElse` είναι ο γενικός τύπος `U` αλλιώς είναι ο γενικός τύπος `V`

```
type IsString<T> = T extends string ? true : false;

let a: IsString<string> = true;
// a is of type true
let b: IsString<number> = false;
// b is of type false
```

```
type IfElse<T, U, V> = T extends true ? U : V;

let c: IfElse<true, string, number> = "hello";
// c is of type string
let d: IfElse<false, string, number> = 42;
// d is of type number
```

Τύπος δεδομένων `any`

Ο τύπος `any` απενεργοποιεί τους ελέγχους τύπων της Typescript και η μεταβλητή μπορεί να λάβει οποιαδήποτε τιμή. *Συνιστάται έντονα να αποφεύγουμε τη χρήση του*, εκτός αν

1. Μεταφέρουμε κώδικα από τη Javascript στην Typescript και υπάρχει πολυπλοκότητα στο να ορίσουμε τύπους στα δεδομένα και ζητάμε μια *προσωρινή λύση*
2. Διάφορες βιβλιοθήκες της Javascript δεν ορίζουν τύπους και για να τις χρησιμοποιήσουμε ευέλικτα χρησιμοποιούμε το `any` για τα αντικείμενα της βιβλιοθήκης
3. Τα δεδομένα μας είναι δυναμικά, π.χ. δεδομένα που εισάγει ο χρήστης ή μια κλήση σε ένα API που επιστρέφει ποικιλία τύπων, χρησιμοποιούμε το `any` λόγω της αβεβαιότητας
4. Θέλουμε να βελτιώσουμε κώδικα Javascript που δεν έχει τύπους και χρειαζόμαστε ευελιξία ώστε να μην κάνουμε μεγάλες επεμβάσεις στον παλαιό κώδικα μέχρι να ολοκληρώσουμετη βελτίωση

Τύπος δεδομένων `any`

Χρήση του `any`

```
let data: any = "hello";
data = 42;
data = true;
```

Χρήση στους πίνακες

```
let arr: any[] = [1, "hello", true];
```

Χρήση στα αντικείμενα

```
let obj: any = { name: "John", age: 30, isMarried: true };
```

Χρήση στις συναρτήσεις

```
function foo(arg: any): any {
  return arg;
}
```

Σε όλα τα παραδείγματα η χρήση του `any` απενεργοποιεί τον έλεγχο των τύπων και ο transpiler παράγει κώδικα Javascript επιδεκτικό σε λογικά λάθη εκτέλεσης που σχετίζονται με λάθος χρήση των τύπων δεδομένων

Τύπος δεδομένων `void`

Στην Typescript ο τύπος `void` χρησιμοποιείται όταν μια συνάρτηση δεν επιστρέφει τιμή ή αν η τιμή που επιστρέφει δεν χρησιμοποιείται.

```
class Logger {
  log(message: string): void {
    console.log(message);
    return true;
  }
} // error TS2322: Type 'true' is not   assignable to type 'void'.
```

```
function logError(error: string): void {
  console.error(error);
}

let result: void = logError("Something went wrong");
console.log(result); // Τυπώνει undefined
```

```
function fn(x: () => void) {
  x();
}
```

```
interface Logger {
  log(message: string): void;
  warn(message: string): void;
  error(message: string): void;
}
```

Τύπος δεδομένων `unknown`

- Ο τύπος `unknown` διαφέρει από τον τύπο `any` καθώς ενώ αναπαριστά ένα άγνωστο τύπο δεδομένων μας υποχρεώνει να υλοποιήσουμε τους κατάλληλους ελέγχους πριν χρησιμοποιήσουμε τα δεδομένα
- Με άλλα λόγια χρησιμοποιούμε το `unknown` για οτιδήποτε άγνωστο που πρέπει να υλοποιηθεί γι' αυτό προσεκτικός έλεγχος τύπων

```
function getLength(value: unknown): number | undefined {
  if (typeof value === "string") {
    return value.length;
  } else if (Array.isArray(value)) {
    return value.length;
  } else {
    return undefined;
  }
}

let strLength = getLength("Hello, world!"); // 13
let arrLength = getLength([1, 2, 3, 4, 5]); // 5
let numLength = getLength(123); // undefined
```

Στο παράδειγμα η συνάρτηση επιστρέφει μήκος συμβολοσειρών και πινάκων, όχι όμως αριθμών γιατί δεν έχει υλοποιηθεί αυτή η περίπτωση

Τύπος δεδομένων `never`

- Ο τύπος δεδομένων `never` χρησιμοποιείται στις περιπτώσεις που δεν είναι δυνατόν να συμβούν, όπως για παράδειγμα στις συναρτήσεις που πάντα προκαλούν εξαίρεση στην εκτέλεση ή δεν επιστρέφουν ποτέ τίποτα
- Η συνάρτηση `throwError` προκαλεί πάντα εξαίρεση στην εκτέλεση και δεν επιστρέφει κάποιο τύπο

```
function throwError(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

- Η συνάρτηση `infiniteLoop` υλοποιεί ένα άπειρο βρόγχο και δεν επιστρέφει ποτέ κάποια τιμή

Τύπος `Record`

- Ορίζει τύπο που περιγράφει αντικείμενα με ονόματα χαρακτηριστικών και τιμές συγκεκριμένου τύπου

```
type MyRecord = Record<string, ValueType>;
```

- `ValueType` είναι οποιοσδήποτε τύπος της Typescript

```
class Person {
  data: Record<string, unknown>;

  constructor(data: Record<string, unknown>) {
    this.data = data;
  }

  getData() {
    return this.data;
  }
}

const data = {
  name: "Alice", age: 30,  height: 170,
  weight: 65,  gender: "female",
};
const person = new Person(data);
console.log(person.getData());
```

Συναρτήσεις στην Typescript

Οι συναρτήσεις επιστρέφουν τύπους και οι παράμετροί τους έχουν τύπους, μπορεί να είναι προαιρετικές και να έχουν εξ ορισμού τιμές

```
function greet(name?: string, greeting = "Hello"): string {
  if (!name) {
    name = "Anonymous";
  }
  return `${greeting}, ${name}!`;
}

console.log(greet()); // Τυπώνει "Hello, Anonymous!"
console.log(greet("John")); // Τυπώνει "Hello, John!"
console.log(greet("Jane", "Hi")); // Τυπώνει "Hi, Jane!"
```

Το `?` δηλώνει παράμετρο που μπορεί είναι `undefined` αντί του τύπου που ακολουθεί

Επιτρέπεται η υπερφόρτωση των συναρτήσεων

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
  return a + b;
}

console.log(add(1, 2)); // Τυπώνει 3
console.log(add("Hello", "World")); // Τυπώνει "HelloWorld"
```

Με την υπερφόρτωση μπορούμε να έχουμε το ίδιο σώμα συνάρτησης που επεξεργάζεται διαφορετικές παραμέτρους εισόδου

Συναρτήσεις στην Typescript

Function types

```
type MathOperation = (a: number, b: number) => number;

function calculate(a: number, b: number, op: MathOperation): number {
  return op(a, b);
}

function add(a: number, b: number): number {
  return a + b;
}

function subtract(a: number, b: number): number {
  return a - b;
}

console.log(calculate(5, 3, add)); // Τυπώνει 8
console.log(calculate(5, 3, subtract)); // Τυπώνει 2
```

- Ένα function type καθορίζει ένα "αποτύπωμα" συνάρτησης, στο παράδειγμα μιας συνάρτησης που απαραίτητα λαμβάνει δύο αριθμούς σαν παραμέτρους και επιστρέφει ένα αριθμό
- Στη συνέχεια οι συναρτήσεις `add` και `subtract` υλοποιούν το function type
- Ο τύπος `MathOperation` μπορεί να περνά σαν τύπος παραμέτρου όπως στην συνάρτηση `calculate`

Ασκήσεις

1. Μια συνάρτηση δέχεται ένα αριθμό και επιστρέφει το τετράγωνο του αριθμού
2. Μια συνάρτηση δέχεται μια συμβολοσειρά και την επιστρέφει αντεστραμμένη
3. Μια συνάρτηση δέχεται ένα πίνακα αριθμών και επιστρέφει το άθροισμά τους
4. Μια συνάρτηση δέχεται ένα αριθμό και επιστρέφει true αν είναι ζυγός αλλιώς false
5. Μια συνάρτηση δέχεται δύο αριθμούς και επιστρέφει το άθροισμα όλων των μεταξύ τους αριθμών που περιλαμβάνει και τα δύο άκρα
6. Μια συνάρτηση δέχεται ένα πίνακα συμβολοσειρών και επιστρέφει το μέγεθος της μεγαλύτερης συμβολοσειράς
7. Μια συνάρτηση δέχεται ένα αντικείμενο και επιστρέφει ένα πίνακα με τα χαρακτηριστικά του
8. Μια συνάρτηση δέχεται μια συμβολοσειρά και επιστρέφει true αν είναι παλίνδρομη

Ασκήσεις

9. Μια συνάρτηση δέχεται ένα πίνακα αριθμών και επιστρέφει τον μεγαλύτερο και τον μικρότερο
10. Μια συνάρτηση δέχεται ένα αριθμό και επιστρέφει true αν ο αριθμός είναι πρώτος
11. Μια συνάρτηση δέχεται ένα πίνακα συμβολοσειρών και επιστρέφει τον πίνακα με τις συμβολοσειρές στα κεφαλαία
12. Μια συνάρτηση δέχεται ένα πίνακα αριθμών και επιστρέφει τον πίνακα χωρίς τους ζυγούς αριθμούς
13. Μια συνάρτηση δέχεται ένα αριθμό και επιστρέφει το παραγοντικό του
14. Μια συνάρτηση δέχεται ένα πίνακα αντικειμένων με χαρακτηριστικό `name` και επιστρέφει μια συμβολοσειρά με όλα τα ονόματα χωρισμένα μεταξύ τους με κενό
15. Μια συνάρτηση δέχεται μια συμβολοσειρά και επιστρέφει τους μοναδικούς χαρακτήρες της

Κλάσεις στην Typescript

Βασίζονται στις κλάσεις του προτύπου ECMAScript 6 και εμπλουτίζονται με έλεγχο τύπων

Typescript

```
class Person {
  name: string;
  age: number;

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}

let person: Person = new Person("John", 30);
person.greet();
console.log(person.name);
console.log(person.age);
```

ECMAScript 6

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}

let person = new Person("John", 30);
person.greet();
console.log(person.name);
console.log(person.age);
```

Στο παράδειγμα τα χαρακτηριστικά της κλάσης δηλώνονται εκτός του constructor.

Κληρονομικότητα στις κλάσεις

- Χρησιμοποιούμε το `extends` για να δηλώσουμε πως μια κλάση κληρονομεί μια άλλη
- Η κλάση `Dog` κληρονομεί την `Animal` και προσθέτει το χαρακτηριστικό `breed` και τη μέθοδο `bark`
- Ο constructor της `Dog` ***οφείλει*** να αρχικοποιήσει πρώτα την `Animal` με τη χρήση της `super`

```
class Animal {
  constructor(public name: string, public age: number) {}
  speak() {
    console.log(`${this.name} says hello!`);
  }
}

class Dog extends Animal {
  breed: string;

  constructor(name: string, age: number, breed: string) {
    super(name, age);
    this.breed = breed;
  }

  bark() {
    console.log(`${this.name} barks!`);
  }
}

const dog = new Dog("Suky", 5, "Golden Retriever");
dog.speak(); // Τυπώνει Suky says hello!
dog.bark(); // Τυπώνει Suky barks!
```

Τροποποιητές πρόσβασης (access modifiers)

Στην Typescript οι τροποποιητές πρόσβασης είναι λέξεις κλειδιά που ελέγχουν την πρόσβαση στα μέλη (χαρακτηριστικά και μεθόδους) μιας κλάσης από έξω από την κλάση

1. `public` Τα μέλη που σημειώνονται σαν `public` είναι προσβάσιμα τόσο εντός όσο και εκτός της κλάσης. Είναι η ***εξ ορισμού*** ρύθμιση πρόσβασης αν δεν χρησιμοποιηθεί καμία
2. `private` Τα μέλη που σημειώνονται σαν `private` είναι προσβάσιμα μόνο εντός της κλάσης. Δεν είναι προσβάσιμα και δεν μπορούν να τροποποιηθούν εκτός της κλάσης
3. `protected` Τα μέλη που σημειώνονται σαν `protected` είναι προσβάσιμα μόνο εντός της κλάσης και εντός των παραγόμενων από την αρχική κλάση κλάσεων

Παράδειγμα χρήσης των access modifiers

```
class Person {
  public name: string; // default access modifier is public
  private age: number;
  protected gender: string;

  constructor(name: string, age: number, gender: string) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }

  public getAge(): number {
    return this.age;
  }

  private setAge(age: number): void {
    this.age = age;
  }

  protected getGender(): string {
    return this.gender;
  }
}
```

```
class Student extends Person {
  public getGender(): string {
    return "unknown";
  }
}

const p = new Person("Alice", 25, "female");
console.log(p.name); // τυπώνει "Alice"
console.log(p.age);
// error: Property 'age' is private and only
// accessible within class 'Person'
console.log(p.gender);
// error: Property 'gender' is protected and only
// accessible within class 'Person'
p.setAge(26);
// error: Property 'setAge' is private and only
// accessible within class 'Person'

const s = new Student("Bob", 20, "male");
console.log(s.getGender()); // τυπώνει "unknown"
```

Συντόμευση δήλωσης χαρακτηριστικών

Μπορούμε να δηλώσουμε τα χαρακτηριστικά μέσω του constructor αν χρησιμοποιήσουμε *access modifiers* (`public` , `private` , `protected`) πριν το όνομα του χαρακτηριστικού. Στην περίπτωση αυτή τα χαρακτηριστικά και δηλώνονται και αρχικοποιούνται με το `new`

```
class Person {
  constructor(public name: string, public age: number) {}

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
const john = new Person("John", 30);
john.greet();
// Hello, my name is John and I am 30 years old.
```

Εξ ορισμού τιμές και αρχικοποίηση

Δηλώνουμε τα χαρακτηριστικά της κλάσης έξω από τον constructor όπου δίνουμε τις εξ ορισμού τιμές, αν αυτές απαιτούνται. Τα χαρακτηριστικά αρχικοποιούνται στον constructor ανάλογα με τις παραμέτρους του

```
class Person {
  name: string = "Anonymous";
  age: number = 0;

  constructor(name?: string, age?: number) {
    if (name) {
      this.name = name;
    }
    if (age) {
      this.age = age;
    }
  }

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
```

- **Τα χαρακτηριστικά `name` και `age` δηλώνονται εκτός του constructor και λαμβάνουν αρχικές τιμές**
- **`name?` και `age?` σημαίνει πως επιτρέπουμε να είναι `undefined`**

Υπολογιζόμενα χαρακτηριστικά κλάσης

Μια κλάση στην Typescript μπορεί να έχει χαρακτηριστικά που η τιμή τους υπολογίζεται κατά την εκτέλεση και εξαρτάται από τις τιμές άλλων χαρακτηριστικών. Δηλώνονται με τη χρήση της λέξης κλειδί `get` πριν τη μέθοδο που υπολογίζει την τιμή τους

- Στο παράδειγμα η κλάση `Rectangle` έχει το υπολογιζόμενο χαρακτηριστικό `area`
- Η μέθοδος `area` είναι ***getter*** για το υπολογιζόμενο χαρακτηριστικό
- Το υπολογιζόμενο χαρακτηριστικό είναι προσβάσιμο όπως και όλα τα άλλα χαρακτηριστικά (`rectangle.area`)

```
class Rectangle {
  constructor(private width: number, private height: number) {}

  get area() {
    return this.width * this.height;
  }
}

const rectangle = new Rectangle(10, 5);
console.log(rectangle.area); // τυπώνει 50
```

Getters και Setters για private χαρακτηριστικά

- Χρησιμοποιούμε τις λέξεις κλειδιά `get` και `set` πριν τον ορισμό των μεθόδων που καθορίζουν την ανάκτηση και την ανάθεση τιμής στο private χαρακτηριστικό
- Η μέθοδος getter επιστρέφει την τιμή του private χαρακτηριστικού
- Η μέθοδος setter καθορίζει τον τρόπο ανάθεσης τιμής στο private χαρακτηριστικό
- Κατά σύμβαση, συσχετίζουμε το private χαρακτηριστικό με τους getter και setter με τη χρήση του `_` πριν το όνομά του

```
class User {
  private _name: string = "";

  get name(): string {
    return this._name.toUpperCase();
  }

  set name(newName: string) {
    this._name = newName;
  }
}

const user = new User();
user.name = "John";
console.log(user.name); // τυπώνει JOHN
```

Χαρακτηριστικά μόνο για ανάγνωση

Ένα συνηθισμένο σχήμα για χαρακτηριστικά που είναι διαθέσιμα μόνο για ανάγνωση είναι με τη χρήση του `private` και ένα getter. Η υλοποίηση γίνεται πιο ευσύνοπτη με τη χρήση του `readonly`

```
class Widget {
  private _id: string;
  get id(): string {
    return this._id;
  }
  constructor(id: string) {
    this._id = id;
  }
}
let widget = new Widget("textBox");
console.log(`Widget id: ${widget.id}`);
// Widget id: textBox
```

```
class Widget {
  readonly id: string;
  constructor(id: string) {
    this.id = id;
  }
}
let widget = new Widget("text");
widget.id = "newId";
// error TS2540: Cannot assign to 'id'
// because it is a read-only property.
```

Αφηρημένες κλάσεις

- Είναι κλάσεις που δεν μπορούν να αποκτήσουν απ' ευθείας στιγμιότυπα
- Ορίζουν ένα κοινό σύνολο από χαρακτηριστικά και μεθόδους που οι παραγόμενες κλάσεις κληρονομούν και μπορούν να υλοποιήσουν
- Στο παράδειγμα μια αφηρημένη κλάση έχει μια αφηρημένη μέθοδο που υλοποιείται στην παραγόμενη κλάση

```
abstract class Animal {
  abstract makeSound(): void;
}

class Dog extends Animal {
  makeSound() {
    console.log("Woof!");
  }
}

const dog = new Dog();
dog.makeSound(); // τυπώνει Woof!
```

Ασκήσεις στις κλάσεις της Typescript

1. Γράψτε μια κλάση `Person` με ιδιότητες `firstName`, `lastName` και `age` και μια μέθοδο `getFullName()` που επιστρέφει το πλήρες όνομα του ατόμου.
2. Η κλάση `Student` κληρονομεί την `Person` και έχει την ιδιότητα `courses` (πίνακας συμβολοσειρών με ονόματα μαθημάτων). Προσθέστε μια μέθοδο `enroll(course:string)` που προσθέτει ένα μάθημα στον πίνακα `courses`.
3. Γράψτε μια κλάση `Employee` με ιδιότητες `firstName`, `lastName`, `jobTitle` και `salary`. Προσθέστε μια μέθοδο `getAnnualSalary()` που επιστρέφει τον ετήσιο μισθό με βάση την ιδιότητα `salary`.
4. Υλοποιήστε μια απλή ιεραρχία κληρονομικότητας από μια αφηρημένη κλάση `Shape` στις κλάσεις `Circle` και `Rectangle`. Η κλάση `Shape` να έχει την αφηρημένη μέθοδο `area()` που υπολογίζει το εμβαδό του σχήματος και υλοποιείται στις `Circle` και `Rectangle`.

Ασκήσεις στις κλάσεις της Typescript

5. Γράψτε μια κλάση `BankAccount` με ιδιότητες `accountNumber`, `accountHolder` και `balance`. Υλοποιήστε τις μεθόδους `deposit(amount:number)` και `withdraw(amount:number)` που μεταβάλουν τη ιδιότητα `balance`. Η τιμή της `balance` δεν μπορεί να είναι μικρότερη από το μηδέν.
6. Γράψτε μια κλάση `Counter` με μια μη δημόσια ιδιότητα `_count` (αρχικοποιημένη στο 0) και μεθόδους `increment()` και `decrement()` που αυξάνουν ή μειώνουν τον μετρητή κατά 1 αντίστοιχα. Υλοποιήστε ένα getter `count` που επιστρέφει την τρέχουσα τιμή του μετρητή.
7. Γράψτε μια κλάση `Vehicle` με ιδιότητες `make`, `model` και `year`. Υλοποιήστε μια στατική μέθοδο `compare(v1: Vehicle, v2: Vehicle)` που συγκρίνει δύο αντικείμενα της `Vehicle` βάση της ιδιότητας `year` και επιστρέφει το νεότερο όχημα.

Ασκήσεις στις κλάσεις της Typescript

8. Γράψτε μια κλάση `Calculator` με μεθόδους `add(x: number, y: number)`, `subtract(x: number, y: number)`, `multiply(x: number, y: number)`, και `divide(x: number, y: number)`. Υλοποιήστε τον χειρισμό της διαίρεσης με το μηδέν στη μέθοδο `divide()`.
9. Γράψτε μια κλάση `Library` με μια μη δημόσια ιδιότητα `_books` (πίνακας συμβολοσειρών με ονόματα βιβλίων). Προσθέστε τη μέθοδο `addBook(book: string)` που προσθέτει ένα βιβλίο στη βιβλιοθήκη και μια μέθοδο `findBook(title: string)` που αναζητά ένα βιβλίο με τον τίτλο του.
10. Γράψτε μια κλάση `Author` με ιδιότητες `firstName`, `lastName` και `books` (πίνακας συμβολοσειρών με τίτλους βιβλίων). Προσθέστε τη μέθοδο `addBook(title: string)` που προσθέτει ένα νέο βιβλίο στη βιβλιογραφία του συγγραφέα. Υλοποιήστε μια στατική μέθοδο `compareBooks(a1: Author, a2: Author)` που συγκρίνει δύο στιγμιότυπα της `Author` με βάση τον αριθμό των βιβλίων των συγγραφέων και επιστρέφει το συγγραφέα με τα περισσότερα βιβλία.

Διακοσμητές (Decorators)

- Οι *διασκομητές* στην Typescript είναι *συναρτήσεις* που επιτρέπουν την τροποποίηση της συμπεριφοράς μιας κλάσης ή των μελών της (χαρακτηριστικά, μέθοδοι, κτλ) κατά τη διάρκεια του ορισμού της, πριν από την εκτέλεση του κώδικα.
- Οι διασκομητές εφαρμόζονται με τη σύνταξη `@decoratorName` ακριβώς πριν την κλάση ή το μέλος της κλάσης που σχεδιάστηκαν να τροποποιήσουν
- Οι διακοσμητές χρησιμοποιούνται σε ένα ευρύ πεδίο σκοπών που περιλαμβάνει τη προσθήκη μεταδεδομένων σε μια κλάση, την επικύρωση της εισόδου στις μεθόδους και τη δημιουργία προσωρινής αποθήκευσης για μια μέθοδο

Οι διακοσμητές είναι βασικό δομικό στοιχείο στο Angular Framework

Πέρασμα παραμέτρων προς στους διακοσμητές ανάλογα με το στοιχείο που διακοσμείται

- Όταν διακοσμείται μια *κλάση*, τότε στο διακοσμητή περνά αυτόματα η *συνάρτηση του constructor* που διακοσμείται.
- Όταν διακοσμείται μια *ιδιότητα της κλάσης*, τότε περνά αυτόματα το *στιγμιότυπο* της κλάσης και το *όνομα της ιδιότητας* που διακοσμείται.
- Όταν διακοσμείται μια *μέθοδος της κλάσης*, τότε περνά αυτόματα το *στιγμιότυπο* της κλάσης, το *όνομα της μεθόδου* και η περιγραφή της μεθόδου που διακοσμείται.
- Όταν διακοσμείται μια *παράμετρος μεθόδου* της κλάσης, τότε περνά αυτόματα το *στιγμιότυπο* της κλάσης, το *όνομα της μεθόδου* και ο αύξοντας αριθμός της σειράς της παραμέτρου στα ορίσματα της μεθόδου που διακοσμείται.

Πέρασμα παραμέτρων προς στους διακοσμητές ανάλογα με το στοιχείο που διακοσμείται

```
function ClassDecorator(constructor: Function) {
  // ...
}

@ClassDecorator
class ExampleClass {
  // ...
}
```

```
function PropertyDecorator(target: Object, propertyKey: string | symbol) {
  // ...
}

class ExampleClass {
  @PropertyDecorator
  public exampleProperty: string;
}
```

```
function MethodDecorator(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<any>
) {
  // ...
}

class ExampleClass {
  @MethodDecorator
  public exampleMethod(): void {
    // ...
  }
}
```

```
function ParameterDecorator(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number
) {
  // ...
}

class ExampleClass {
  public exampleMethod(@ParameterDecorator param: string): void {
    // ...
  }
}
```

Παράδειγμα διακόσμησης κλάσης

- Η συνάρτηση `logClass`, όταν εφαρμοστεί σαν διακοσμητής, θα δεχτεί σαν όρισμα τον constructor της κλάσης.
- Στο πρότυπο ECMAScript 6, ο constructor της κλάσης δημιουργεί αυτόματα την ιδιότητα `name` με το όνομα της κλάσης.
- Η εφαρμογή του διακοσμητή στην κλάση *εκτελεί άμεσα* τη συνάρτηση διακόσμησης.

```
function logClass(target: Function) {
  console.log(`Decorating ${target.name}`);
}

@logClass
class Person {
  constructor(public name: string, public age: number) {}

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
```

```
Decorating Person
```

`@decoratorName` είναι "syntactic sugar"

Η έννοια της *"συντακτικής ζάχαρης (syntactic sugar)"* σε μια γλώσσα προγραμματισμού αναφέρεται σε μια δομή σύνταξης που παρέχεται στη γλώσσα χωρίς να προσθέτει νέα λειτουργικότητα αλλά καθιστά τη γραφή του κώδικα πιο ευανάγνωστη ή ευκολότερη στη χρήση. Αντί `@decoratorName` μπορούμε να γράψουμε `decoratorName(class ...)`

```ts
function logClass(target: Function) {
  console.log(`Decorating ${target.name}`);
}

@logClass
class Person {
  constructor(public name: string, public age: number) {}

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
```

```ts
function logClass(target: Function) {
  console.log(`Decorating ${target.name}`);
}

logClass(
  class Person {
    constructor(public name: string, public age: number) {}

    greet(): void {
      console.log(
        `Hello, my name is ${this.name} and I am ${this.age} years old.`
      );
    }
  }
);
```

Το αποτέλεσμα είναι ίδιο και στις δύο περιπτώσεις: `Decorating Person`

Διακοσμητές με παραμέτρους

- Αν *μια συνάρτηση επιστρέφει άλλη συνάρτηση* και εφαρμοστεί σαν διακοσμητής σε μια κλάση, τότε η κλάση διακοσμείται με τη συνάρτηση που επιστρέφεται από την αρχική συνάρτηση.
- Η συνάρτηση που επιστρέφεται μπορεί να *λάβει παραμέτρους* από την αρχική συνάρτηση και τελικά να διακοσμήσει την κλάση.

```
function logClass(prefix: string, suffix: string) {
  return (target: Function) => {
    console.log(prefix);
    console.log(`Decorating ${target.name}`);
    console.log(suffix);
  };
}

@logClass("Begin", "End")
class Person {
  constructor(public name: string, public age: number) {}

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
```

```
Begin
Decorating Person
End
```

Πολλαπλή διακόσμηση

- Μπορούμε να εφαρμόσουμε σε μια κλάση όσους διασκομητές θέλουμε
- Οι διακοσμητές εφαρμόζονται από τα δεξιά προς τα αριστερά (ή αλλιώς από κάτω προς τα πάνω στο παράδειγμα)
- Η έξοδος είναι

```
Begin
Decorating Person
End
[
Decorating Person
]
Decorating Person
```

```typescript
function logClass(target: Function) {
  console.log(`Decorating ${target.name}`);
}

function logClassWithParams(prefix: string, suffix: string) {
  return (target: Function) => {
    console.log(prefix);
    console.log(`Decorating ${target.name}`);
    console.log(suffix);
  };
}

@logClass
@logClassWithParams("[", "]")
@logClassWithParams("Begin", "End")
class Person {
  constructor(public name: string, public age: number) {}

  greet(): void {
    console.log(
      `Hello, my name is ${this.name} and I am ${this.age} years old.`
    );
  }
}
```

Διακόσμηση ιδιότητας

- Δημιουργούμε ένα διακοσμητή ιδιότητας με όνομα `upperCase` που θα μετατρέπει σε κεφαλαία την ιδιότητα που εφαρμόζεται.
- Ο διακοσμητής ορίζει getter και setter που επιδρούν άμεσα στην τιμή της ιδιότητας που διακοσμείται.
- Με το `new MyClass("hello")` ενεργοποιείται άμεσα ο setter που μετατρέπει την τιμή σε κεφαλαία.
- Με το `console.log(example.myString)` ενεργοποιείται ο getter.

```
function upperCase(target: any, key: string) {
  let value = target[key];
  const getter = function () {
    return value;
  };
  const setter = function (newVal: string) {
    value = newVal.toUpperCase();
  };
  Reflect.defineProperty(target, key, {
    get: getter,
    set: setter,
    configurable: true,
  });
}

class MyClass {
  @upperCase myString: string;
  constructor(str: string) {
    this.myString = str;
  }
}

const example = new MyClass("hello");
console.log(example.myString); // Τυπώνει "HELLO"
example.myString = "world";
console.log(example.myString); // Τυπώνει "WORLD"
```

Διακόσμηση μεθόδου

- Δημιουργούμε διακοσμητή μεθόδου που μετρά το χρόνο εκτέλεσης της μεθόδου.
- Στη μεταβλητή `originalMethod` αναθέτουμε την αναφορά στη μέθοδο πριν τη διακόσμηση.
- Μεταβάλουμε τη μέθοδο σημειώνοντας πρώτα το χρόνο της αρχής της εκτέλεσης, καλούμε την αρχική μέθοδο και σημειώνουμε το χρόνο λήξης της εκτέλεσης.

```
function MeasureExecutionTime(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<any>
) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    const startTime = performance.now();
    const result = originalMethod.apply(this, args);
    const endTime = performance.now();
    const executionTime = endTime - startTime;

    console.log(
      `Χρόνος εκτέλεσης της μεθόδου '${String( propertyKey )}': ${executionTime} ms`
    );

    return result;
  };
}

class MyClass {
  @MeasureExecutionTime
  public exampleMethod(): void {
    for (let i = 0; i < 1000000; i++) {
      // Κάποιοι υπολογισμοί
    }
  }
}

const myInstance = new MyClass();
myInstance.exampleMethod();
```