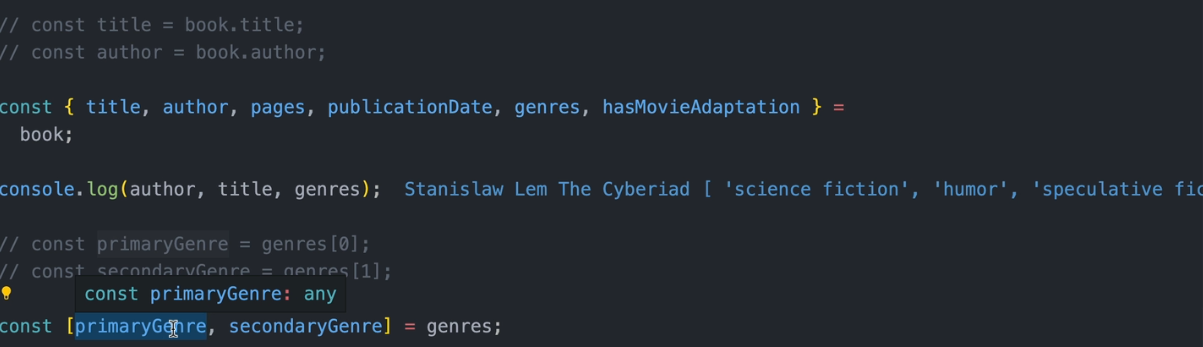
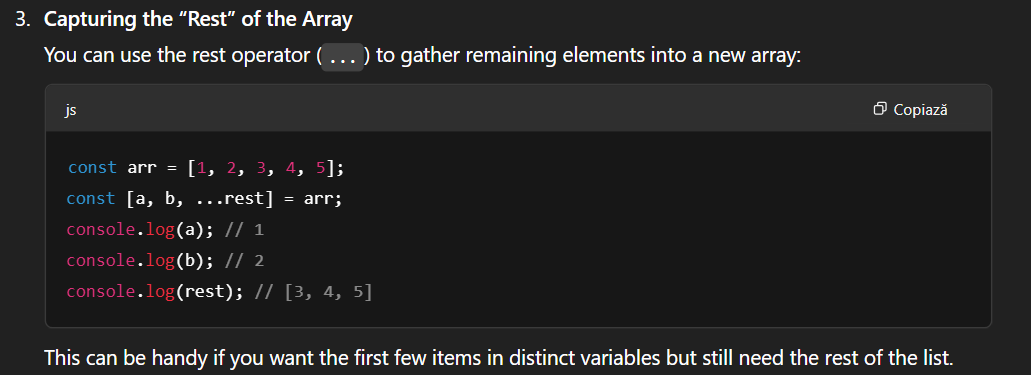
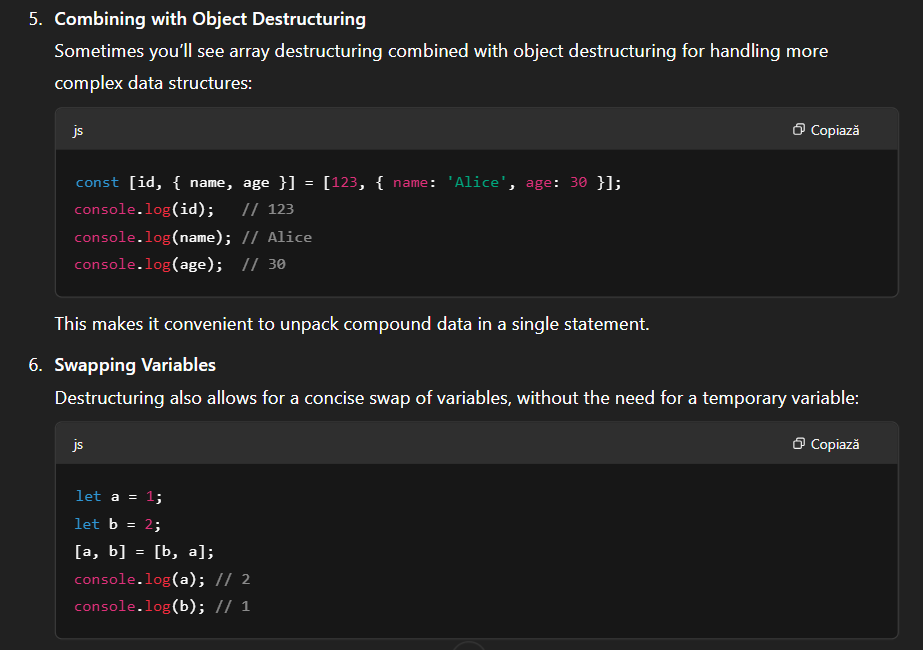
**JAVASCRIPT**

**Destructuring**

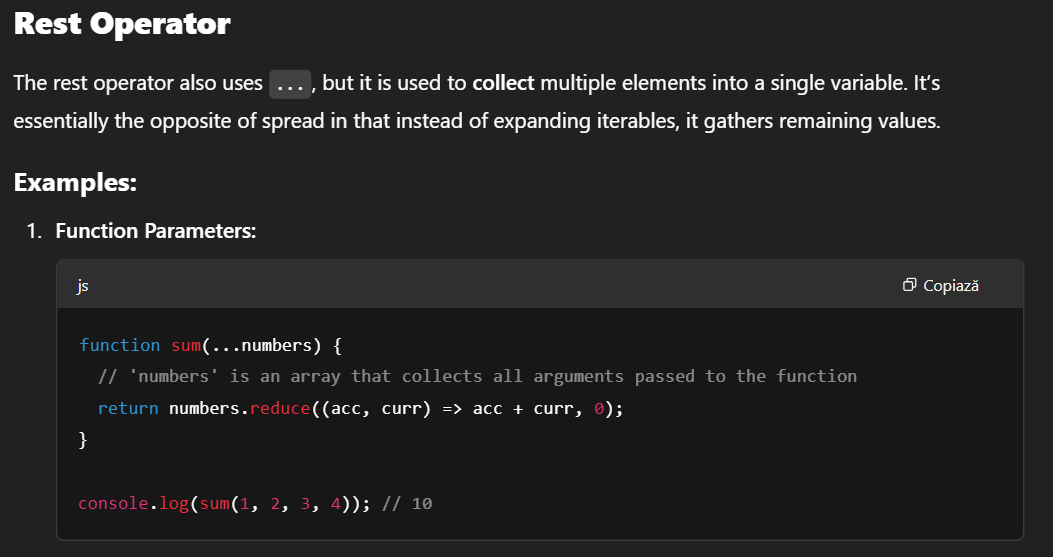
* **In the same manner we destructure the properties of an object, we can destructure the elements of an array**

****

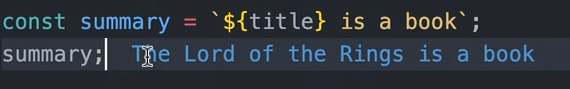
* **In the first example, we destructure the properties title, author etc… from the book object. The variables we define have to have the same names as the object’s properties**
* **In the second example, we declare new variables called primaryGenre, secondaryGenre. The first element we declare will receive the list’s first element’s value, the second will receive the second’s and so on. Here we can see that the naming does not matter, but what matters is the order in which we declare our variables.**
* **const [first, , third] = arr; // Skips the second element**
* **The rest operator can be used for array destructuring, but only at the end of the array.**
* ****

****

**Rest Operator**

****

**Template Literals**

****

**TYPESCRIPT**

**SECTION 33**

460. What & Why?

**- Typescript adds static typing to javascript**

461. Installing & Using TypeScript

npm init -y 🡺 **creates an empty package.json for our dependencies**

npm install typescript

**Typescript doesn’t run in the browser. We need to compile typescript to javascript first so:**

**npx tsc 🡺 we compile our typescript files and it generates a new .js file**

**And now it is this new javascript file that we import and run in the browser.**

462. Base Types & Primitives

**Primitives: number, string, Boolean ( just like in javascript)**

**To initialize a variable, we create it and place 2 dots after which we give its typing**

**let age: number = 24;**

**We can also set types for the parameters of a function**

**function add(a: number, b: number) { …}**

**We also have null and undefined but we don’t assign it because later on we wouldn’t be able to assign a value to the variable**

**let hobbies: null;**

**hobbies = 12; 🡺 results in error**

**We can also create a variable which has the any type so we don’t impose any conditions.**

463. Array & Object Types

**For arrays:**

**let hobbies: string[];**

**hobbies = [‘Sports’, ‘Cooking’];**

**For objects:**

**We add curly braces and impose what fields our object should have, and which types my fields should have.**

**let person: {**

**name: string;**

**age: number;**

**};**

**So this would afterwards work**

**person = {**

**name: ‘Max’,**

**age: 32**

**}**

**But we would get an error for something like this:**

**person = {**

**isEmployee: true**

**};**

**Also, if we want to have an array full of people ( objects which respect the signature from above) we would say**

**let people: {**

**name: string;**

**age: number;**

**}[];**

464. Type Inference

**SIDENOTE: Check for type inference by hovering over the variable, function, etc…**

**By default, typescript infers the type onto a variable. If we were to define a variable without a type but assign to it a string value, we wouldn’t be able afterwards to assign a value of type number to it, because typescript already inferred the type string to that variable.**

**let course = ‘Math’;**

**course = 1222;**

**🡺 results in an error**

**We could also specify the type string to the course variable, but if we assign to it a string value from the start, it would be redundant. The type would already be inferred.**

465. Working with Union Types

**We could have variables which allow multiple types to be assigned to them. For that we use union types.**

**let course: string | number = “Math”;**

**course = 1222;**

**🡺 This wouldn’t result in an error because now we allow both the string and the number types.**

466. Assigning Type Aliases

**Name aliases allow us to reuse the form of a variable or object, without repeating it in our code.**

**We could create the alias Person;**

**type Person = {**

**name: string;**

**age: number;**

**};**

**And reuse it whenever we want to create a variable of this type.**

**let person: Person;**

**let people: Person[];**

467. Diving into Functions & Function Types

**function add(a: number, b: number){**

**return a + b;**

**}**

**As previously said, we can set types for the parameters of a function. But also, in this example, typescript also infers the type of number to the function’s value.**

**but we could also set the type of a function to whatever we want ( primitive, union etc..)**

**function add(a: number, b: number) : number{**

**return a + b;**

**}**

**There also is a special return type. VOID**

**function printSmth(value: any) {**

**console.log(value);**

**}**

**The Void type is already inferred in the above shown function.**

468. Understanding Generics

**function insertAtBeggining(array: any[], value: any) {**

**const newArray = [value, …array];**

**return newArray**

**}**

**Let’s say we have the above function for adding a value at the beginning of a pre-existing list.**

**Even if we were to give our function an array full of numbers and also a number value to be added, it wouldn’t detect the fact that we have an array of numbers because we said it could be of type any.**

**So what we would want to do is use generics.**

**function insertAtBeggining<T>(array: T[], value: T) {**

**const newArray = [value, …array];**

**return newArray**

**}**

**Now the type of array returned by our function will be known.**

469. Classes & TypeScript

**Classes can be used like in java.**

**We could create a class and define its instance variables, a constructor and also methods.**

**There are also access modifiers for both the instance variables and the methods like public and private.**

**class Student {**

**firstName: string;**

**lastName: string;**

**age: number;**

**private courses: string[];**

**constructor(first : string, last : string, age: number, courses: string[])**

**{**

**this.firstName = first;**

**this.lastName = last;**

**this.age = age;**

**this.courses = courses;**

**}**

**enroll(courseName: string) {**

**this.courses.push(courseName);**

**}**

**listCourses() {**

**return this.courses.slice();**

**}**

**}**

**So you could define a new object like**

**const student = new Student(‘Max’, ‘Mosh’, 20, [‘Angular’]);**

**student.enroll(‘React’)**

**You could also not write the instance variables at the beginning, but leave them in the constructor like this(Note that we also have to provide the access modifiers now):**

**constructor(**

**public first: string,**

**public last: string,**

**public age: number,**

**private courses: string[])**

**{}**

**And everything works just as before.**

470. Working with Interfaces

**Interfaces are also very similar to those in java.**

**We could define what instance variables our objects could have, and also the**

**interface Human {**

**firstName: string;**

**age: number;**

**greet: () => void; // we only write the nr of parameters and return type**

**}**

**So we can create a new object with the Human type, as long as it respects the requirements set forth by the interface.**

**let max: Human;**

**max = {**

**firstName: ‘ MAX’,**

**age: 32,**

**greet() {**

**console.log(“Hello”};**

**},**

**};**

**This gives us the same functionality that an Alias would. But interfaces have one more extra feature. They can be implemented by classes and they force those classes to respect their structure.**

**class Instructor implements Human {**

**firstName: String;**

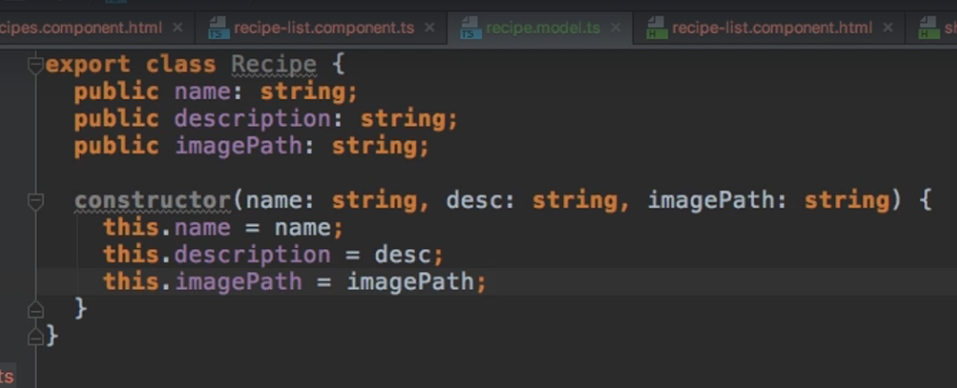
**age: number;**

**greet() {**

**console.log (“Hello”}**

**}**

**For our Course’s application:**

****

**We create a model class which will later be used.**

**In another class, we create a new recipes variable which will be of type Array of Recipe ( the model class we created beforehand) .**

**Text

Description automatically generated**

Access Modifiers

**Public:**

* **If no modifier is provided, then the method or property is assumed to be public which means it can be accessed internally or externally.**

**Private:**

* **The method or property is only accessible internally within the class.**

**Protected:**

* **The method or property is accessible only internally within the class or any class that extends it but not externally.**

**Readonly:**

* **Will cause the TypeScript compiler to throw an error if the value of the property is changed after its initial assignment in the class constructor.**

[**https://stackoverflow.com/questions/37233735/interfaces-vs-types-in-typescript**](https://stackoverflow.com/questions/37233735/interfaces-vs-types-in-typescript)

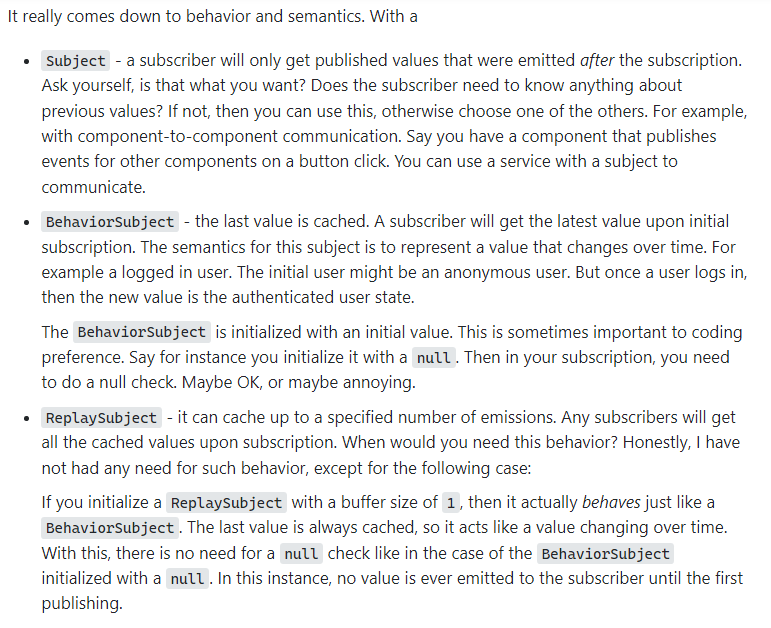
**CDH: related-inspirations.component pentru @OnChange**

**BehaviorSubject vs Subject**

[**https://stackoverflow.com/questions/43348463/what-is-the-difference-between-subject-and-behaviorsubject**](https://stackoverflow.com/questions/43348463/what-is-the-difference-between-subject-and-behaviorsubject)

**Text

Description automatically generated with low confidence**

**Text

Description automatically generated**

[**https://stackoverflow.com/questions/39885264/what-are-rxjs-subjects-and-the-benefits-of-using-them/39885817#39885817**](https://stackoverflow.com/questions/39885264/what-are-rxjs-subjects-and-the-benefits-of-using-them/39885817#39885817)