# CSS

For every element set the distance from previous element using margin-top (not padding). In overflow: scroll; margins don’t collapse.

## Positioning

The position can take values:  
- absolute

- static (by default)

- relative (relative to default position)

- fixed

- sticky

The absolute checks the first ascendant (parent) element that have set position property and position the element relatively to this ascendant. If the ascendant with set position not found - removes the absolute element from the entire html page and positions it relatively to page sides using top, right, bottom, left properties.

fixed - same as absolute but always relative to entire html page and scrolls with the scrolling.

sticky - positioned as static but begins to scroll with page when touch the html page edge. If you set e.g. top: 50px – it will begin scrolling when the element be 50px from top edge.

## Inheritance

If you define a css rule like

.class1 .class2 {

this rule will work even if .class2 nested deeper than 1 level and the class2 is not children but e.g. grandchildren of class1.

Some properties are inherited by default no matter how deep nested elements

<div id="box">

    <section>

        <section>This is some text inside the box.<section>

    </section>

</div>

#box {

      background-color: brown;

      padding: 100px;

}

“This is some text…” will also have background color brown. The inheritance works on those properties for all selectors: tags, classes, id.

Some properties like `padding` are not inherited, to inherit them it should be set to `inherit`

#sec {

  padding: inherit;

}

But it inherits from close parent, not from grandparents:

        <div id="box">

            <section>

                <section id="sec" >This is some text inside the box.<section>

            </section>

        </div>

#box {

      background-color: brown;

      padding: 100px;

}

#sec {

  padding: inherit;

}

The padding of #sec will be 0px because the padding of parent <section> is default 0px.

## Select needed element

To select all descendants (children, grandchildren etc.) write

.class \* {

  padding: 10px;

}

To select only first-level children:

.class > \* {

  padding: 10px;

}

Select first elements B directly followed by A elements:

A + B {

  padding: 10px

}

Select all elements B followed by A element (on the same level of nesting):

A ~ B {

  padding: 10px

}

Select first of element and first children(p) of nested elements:

div p:first-child{

  padding: 10px;

}

To select only first child of element use

div > p:first-child{

  padding: 10px;

}

Select every element that is the 8th child of another element:

:nth-child(8){

  padding: 10px;

}

In SCSS to select certain descendant elements instead of

.class div {

padding: 10px;

}

you can write in this way:

.class {

div {

padding: 10px;

}

}

## FlexBox

In flex-box the `flex` property works bad:



It’s ignore 200px size.

So I prefer to use related parameters:

#div1{

  background-color: coral;

  width: 100%;

  flex-shrink: 1;

}

#div2{

  background-color: lightblue;

  width: 200px;

  flex-shrink: 0;

}

#div3{

  background-color: lightgreen;

  width: 200px;

  flex-shrink: 0;

}

The div1 take all the rest place and only div1 is shrink.

Don’t use flex-grow and width.

#div1{

  background-color: coral;

  min-width: 100px;

  width: 50%;

}

#div2{

  background-color: lightblue;

  min-width: 100px;

  width: 100%;

}

#div3{

  background-color: lightgreen;

  width: 200px;

  flex-shrink: 0;

}

div2 is twice wider than div1, but if shrink – 100px both:

## Width Height 100%

If ancestors heights are set to 100% - the elements will take the whole screen space.

But, if one of nested element are bigger than screen size, and the `overflow` of it parent is not set – it will EXPAND the screen size, and all ancestors will take the 100% of new screen size. This is because the default overflow is `visible`.

Also setting h/w in % works like shit, so try to use these parameters together:

**flex-grow: 1** – sets that the element can grow and how much relatively to other growing elements,

**flex-basis: 100px** – sets the minimal size (height if the parent is flex-direction column)

**overflow: hidden** (or another) ­– limits the size of children.

Then, when you have a growing box, if you put another element inside – you can operate on **max-width max-height** parameters.

## Pseudo classes

::after – adds element (inline) after main element – in the end:

<p class="boring-text">Here is some plain old boring text.</p>

.boring-text::after {

content: " <- BORING";

color: darkviolet;

font-weight: bolder;

}

# SCSS

All variables starts from “$” sign:

$primary-color: #3498db;

$font-size-base: 16px;

.button {

  background-color: $primary-color;

  font-size: $font-size-base;

}

# HTML

## Form input validation

There are a lot of input type like “text”, “password”, “number”, “tel”, “time”

`accept` set file filter when the file explore window is opened, but the user can select any file still.

<label> is bound to `id`.

<input> is bound to controller method parameters through `name`.

Additional fast validation in js:  
  
html and js validation is only on client side and can be avoided. To enhance security add server-side validation:



# JS

What to learn:

* Variables (let, const)
* Functions
* Arrays & Objects
* Loops, if/else
* Scopes
* Basics of this
* Promises & async/await
* ES6+ features (spread operator, destructuring, arrow functions)

## JS Front End

Get element by Id:

const elem = document.getElementById("main-text");

elem.style.color = newColor;

Instead of

<button onclick="onClickFunction()">Click me</button>

It’s recommended always to use addEventLestener() that keeps js separately from html and allows to add multiple onclick functions.

The addEventListener() method sets up a function that will be called whenever the specified event is delivered to the target.

//both versions work fine, but the script must be after pulled html elements – the DOM

// must be generated first (or use document.addEventListener('DOMContentLoaded', () => {)

<script>

document.getElementById("login-form").addEventListener("submit", e => {

    document.getElementById("login-button").disabled = true;

});

</script>

//version 2

<script>

let loginForm = document.getElementById("login-form");

let loginButton = document.getElementById("login-button");

loginForm.addEventListener("submit", e => {

    loginButton.disabled = true;

});

</script>

When DOM is loaded, the browser can safely apply js scripts on html elemetns, because “DOM loaded” means that all html elements of file are turned out into in-memory nodes that are part of DOM tree. The images, fonts, stylesheets may finish loading **after** DOM.

## Fundamentals of JS

In JS methods names start from small letter.

### Primitive DataTypes:

- Number – any number 3, 2.5, -2. Number bigger than ~1015 loose precision, than use BigInt  
- BigInt numbers (without decimals) that can be bigger than ~1015  
to declare BigInt: let y = 9999999999999999n;  
BigInt can’t be used in arithmetic with Number, use conversion:  
let x = 5n;  
let y = Number(x) / 2;  
- String, can use both “ “ and ‘ ‘.

- Boolean  
- Null  
- Undefined (similar to null)  
- Symbol - A newer feature to the language, symbols are unique identifiers  
Other data types:  
- Object - collections of related data.

- function (it is an object at the same time)

foo = function() {};

console.log(typeof foo); // function

take integer part:

 parseInt(1.9) // 1

parseInt(-3.4) // -3

Difference between var, let, const – it’s not important.

Const only can’t be reassign by “=”, but the value can be mutable.

const person = { name: "Alice" };

person.name = "Bob";           // ✅ OK

person.age = 30;               // ✅ OK — adding new property

// person = { name: "Carol" }; // ❌ Error — reassignment

Var variables have function scope. Let variables has the block scope. It can’t be accessible outside the particular code block ({block}).

**function** f() {

**if** (**true**) {

**let** b = 9

console.log(b); *// 9*

}

console.log(b); *// ReferenceError: b is not defined*

}

f();

console.log(b); *// error*

**function** f() {

**if** (**true**) {

**var** b = 9

console.log(b); *// 9*

}

console.log(b); *// 9*

}

f();

console.log(b); *// ReferenceError: b is not defined*

Best practice is not using var, but just:

**function** f() {

**let** b;

**if** (**true**) {

b = 9

console.log(b); *// 9*

}

console.log(b); *// 9*

}

This is ok:

**let** a = 10;

**function** f() {

a = 9

console.log(a) *// 9*

}

f();

**let** a = 10

*// It is not allowed (but it’s ok for var)*

**let** a = 9

*// It is allowed*

a = 10

### Hoisting in JS (only for var):

When JavaScript is executed, the interpreter moves or “hoists” all variable declarations to the top of their containing function / scope boundary, regardless of where they occur.



Transforms into this

This is ok:

a = 10;

**var** a;

console.log(a); *// 10*

a = 10; *// error*

**let** a;

console.log(a);

  
Const is the same as let but can’t be changed.  
  
Instead of

Use (but won’t work with “ “)



Difference between loose equality “==” and strict equality “===”:  
both treat NaN != NaN,  
“==” performs a type conversion when comparing two things, e.g. these consts are loose equal:  
const num = 0;

const big = 0n;

const str = "0";

const obj = new String("0");

const bool = false;  
“===” doesn’t perform a type conversion, examples above are not strict equal.  
null == undefined, but null !== undefined

null != false

new String('foo') != new String('foo')

Only Object.is(NaN, NaN) return true for comparing NaN (or use custom solutions).

let str1 = new Object("0");

let str2 = new Object("0");

let zero = 0;

console.log(str1 == zero);              // t

console.log(str2 == zero);                                  // t

console.log(str1 == str2);              // f

### Functions

function myFunction(p1, p2) {

    return p1 \* p2;

  }

Functions are hoisted to the top of the scope.

In JS, functions can be:

* Assigned to variables
* Passed as arguments
* Returned from other functions

function sayHi() {

    console.log("Hi");

}

let greeter = sayHi; // Assigning function to variable

greeter(); // Calls sayHi

let a = add(5,5);

function add(a, b) { //functions hoisted to the top of the scope

    return a + b;

}

let a1 = multiply(2, 2); //error

const multiply = function(a, b) { //function expressions don't hoisted

    return a \* b;

};

You can attach properties to functions, because functions are objects:

function counter() {}

counter.value = 0;

console.log(counter.value);

No “this” keyword.

Default parameters as in C#.

//"params" in JS

function sum(...numbers) {

    let sum = 0; //jest let sum; won't work;

    for(let i = 0;i < numbers.length;i++){

        sum += numbers[i];

    }

    return sum;

}

console.log(sum(1, 2, 3, 4)); // 10

Functions passed to other functions are common:

//setTimeout is a JS function that runs the passed function after delay (ms)

setTimeout(() => console.log("Delayed"), 1000); //doesn't block the program, just execute this later

Closure – the function that return a function and may have free variables (global variable):

function makeCounter() {

    let count = 0;

    return () => ++count;

}

const counter = makeCounter();

console.log(counter()); // 1

console.log(counter()); // 2

## Keywords

### This

In JavaScript, **this** is dynamic — its meaning depends on who called the function, not where it's defined.

Global scope:

console.log(this); // In browser: Window object

Inside object:

function regularFunction() {

    console.log(this.name);

}

let obj = { name:'Alice', func:regularFunction };

obj.func(); //Alice

Lost content:

const person = {

    name: "Bob",

    greet() {

        console.log(this.name);

    }

};

const sayHi = person.greet;

sayHi(); // ❌ undefined (or global object)

### { …item }

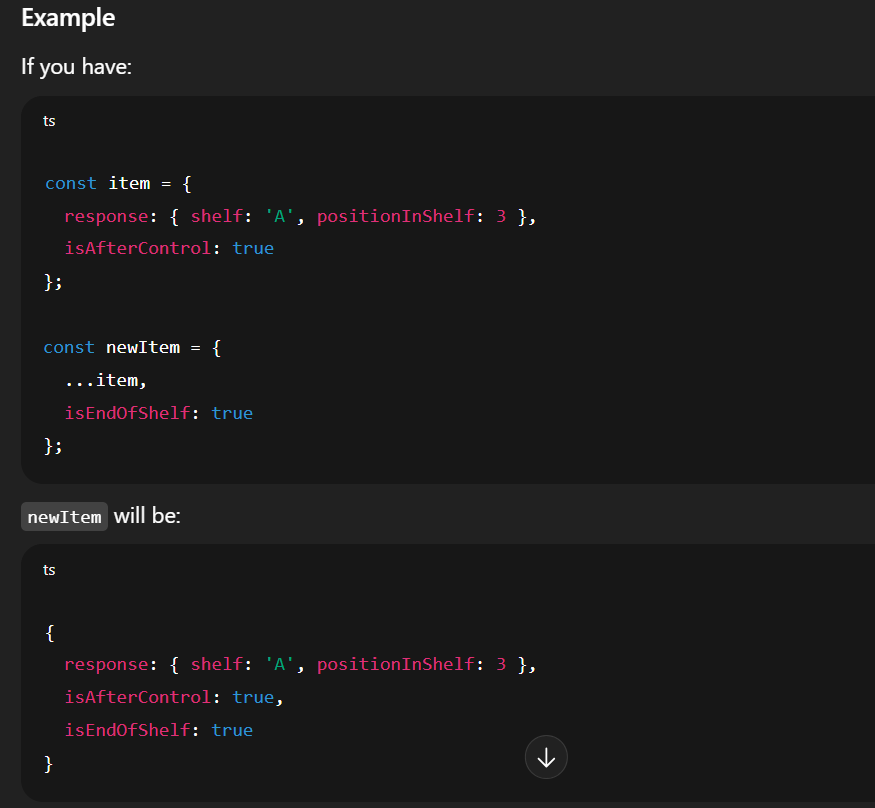
//returns and object same as item but with additional isEndOfShelf property (if exist - replaces the original value)

return {

    ...item,

    isEndOfShelf

};



It’s a good practice in Angular to use it, because it creates new object that is easier to detect to update the UI rather than mutation of existing object.

## Collections

**Arrays**

Arrays in JS:

* + Equivalent to List<T> and array T[] in C#.
  + Dynamic in size
  + Can hold mixed types



**Object**

* + Equivalent to Dictionary<string, object> in C#.
  + Keys are always strings (or symbols), can have methods.

//Object

const person = {

    name: "Alice",

    age: 30

};

**Map**

* + Similar to Dictionary<TKey, TValue> in C#, but
  + Maintains insertion order
  + **Map** is better than **Object** when keys are not strings or when key order matters.
  + Methods .set, .get, .has, .delete, .clear

const map = new Map();

map.set("key", 123);

map.set({ id: 1 }, "value");

console.log(map.get("key")); //123

**Set**

* + Equivalent to HashSet<T> in C#

//Sets

const set = new Set([1, 2, 3, 3]); //.add, .has, .delete

console.log(set); // 1, 2, 3

## Loops

for(let i in array) == for(let i = 0;i < array.length;i++)

c# foreach equivalent: for(let e **of** array)

## Promises

A **Promise** is an object that represents the eventual **completion (or failure)** of an asynchronous operation and its resulting value. Like Task<T> in C#. Promise is a class, the constructor takes a function as an parameter – this function named **executor**.

A Promise has 3 states:

* **Pending** – still in progres
* **Fulfilled** – completed successfully (then)
* **Rejected** – failed with an error (catch)

const promise = new Promise((resolve, reject) => {//timeout sarts here, not in .then line

    setTimeout(() => {

        resolve("Done!");  // marks as successful or reject("Error!") - marks as failed

    }, 1000);

});

promise

    .then(result => console.log(result)) // if resolved

    .catch(error => console.error(error)) // if rejected

    .finally(() => console.log("Always runs"));

//example

function getUser() {

    return new Promise((resolve, reject) => {

        setTimeout(() => resolve({ name: "Alice" }), 1000);

    });

}

getUser()

    .then(result => console.log(result.name)) // "Alice"

    .catch(err => console.error(err));

The executor runs synchronously when the promise is called, but the *.then* callbacks are **always asynchronous and wait for result, but also wait for current synchronous code finishes.**

**Example:**

console.log("Start");

const promise2 = new Promise((resolve, reject) => {

    for (let i = 0; i < 1e9; i++) {}  // ~1 second

    console.log("Inside executor");

    resolve("Done!");

})

.then(r => console.log(r)); //then schedules microtask

// microtasks are run always after the current synchronous code

for (let i = 0; i < 1e9; i++) {}  // ~1 second

console.log("End");

//Output

// Start

// Inside executor   <-- after 1 s

// End               <-- after 1 s

// Done!

Promise.all([promise1, promise2]);

try{

    await Promise.all([

        resolvePromise.then((result) => console.log(result)).catch((error) => {throw new Error(error);}),

        rejectPromise.then((result) => console.log(result)).catch((error) => {throw new Error(error);})

    ]);

} catch (e){

    console.log(`Caught error with message: ${e.message}`);

} finally {

    console.log("End of try-catch");

}

//Output if timeout resolvePromise is longer than rejectPromise:

// Caught error with message: bad request

// End of try-catch

// ok

**Current stack, microtask queue, macrotask queue**

*.then* or *.catch* are scheduled to microtask queue that runs after current stack (synchronous code finishes). All *setTimeout()* are scheduled to macrotask queue that runs after the whole microtask queue run.

console.log("Start");

setTimeout(() => Promise.resolve().then(() => console.log("setTimeout Promise.then")), 0);

setTimeout(() => console.log("setTimeout"), 0); // macrotask queue

Promise.resolve().then(() => console.log("Promise.then")); // microtask queue

console.log("End");

//Output

// Start

// End

// Promise.then

// setTimeout(Promise.then)

// setTimeout

## Async await

Async/await is syntax sugar over Promises. Async await allows to replace this:

promise.then(result => {

    // ...

  }).catch(err => {

    // ...

  });

To this:

try {

    const result = await promise;

// ...

  } catch (err) {

    // ...

  }

*async* function always return a promise (even if you return a simple value – implicit conversion).

*await* pauses execution inside an async function until a Promise settles:

* + If resolved → returns the result.
  + If rejected → throws the error.

*await* only works inside async functions.

async function getData() {

    return 42;

  }

  getData().then(console.log); // logs 42

  //Behind the scenes: return 42; is turned into return Promise.resolve(42);

To handle errors use try-catch instead of .catch()

Don’t mix await and .then,:

//instead of this

const data = await fetch(url).then(res => res.json());//note that await is applied for whole .then chain - awaits for fetch and then awaits for res.json()

//write this

const res = await fetch(url);

const data = await res.json();

//Full flow example TS

async function getUser(id: number) {

    const res = await fetch(`/api/users/${id}`);

    if (!res.ok) throw new Error("User not found");

    return res.json();

}

async function main() {

    try {

        const user = await getUser(1);

        console.log("User:", user);

    } catch (e) {

        console.error("Error:", e);

    }

}

In modern environments (ES modules, Node 14+, browsers), **top-level await** is allowed

const res = await fetch("...");

But in older environments or files, it must be **inside an async function**.

## C# and JavaScript/TypeScript

Nice article about C# and JS <https://mauricebutler.wordpress.com/2011/11/07/getting-started-with-javascript-with-a-c-background/> (but it’s from 2011)

- Same { } ;  
- Same if, else, switch



- Class and Interface keywords same in C# and TS

- same accessors (public, protected …) in C# and TS

- arrow => same only for lightweight functions

- async await. C# returns Task, JS and TS return Promise,

- garbage collection,

- C# have NuGet package manager, for JS similar role plays npm,

- TS have union types,

- TS have decorators, e.g. @sealed. Some of them are implemented in C# as keywords or attributes,

- In JS, the value `NaN` (Not a Number) occurs when the value is returned as a number type, but the value is not parseable as a number. The value `infinity` occurs when a number exceeds the upper limit 1.7976931348623157E+10308,

- The type of `Null` in JS is `object`,

- In JS anything that exists and has a value will evaluate as true unless the value is false, null, undefined, 0, NaN or an empty string,

- Instead of `someObject?.prop` in JS write ` if (someObject) `

- You can add properties to objects (in this case it’s a function object):

- C# foreach loop in JS: **in -> of**

- JS **in** iterates over properties:



x: fname, lname, age

Similar try-catch, *error* instead of *exception*:

try {

    throw new Error("Error message");

  } catch (err) {

    console.error("Error caught:", err.message);

  } finally {

    console.log("Try-catch run");

  }

Sa in C#, try-catch works only with sync code or async with await:

const resolvePromise1 = new Promise((resolve, reject) => {

    setTimeout(() => resolve("ok"), 1000 );

});

const rejectPromise1 = new Promise((resolve, reject) => {

    setTimeout(() => reject("bad request"), 1000);

});

try{

    await resolvePromise1.then((result) => console.log(result))

        .catch((error) => { throw new Error(error); });

    await rejectPromise1.then((result) => console.log(result))

        .catch((error) => {throw new Error(error)});

} catch (e) {

    console.log(`Registered error: ${e.message}`);

} finally {

    console.log("End of try catch");

}

//the output appears in 1 second:

// ok

// Caught error with message: bad request

// End of try-catch

Instead of “… ?? False” use double negation “!! ...”

!!this.listItemData().group?.some((item) => item.isEndOfShelf)

If group is null or undefined, the !! converts it undefined -> true -> false

## Display pdf on desktop and mobile

1. Download and configure pdfjs library.
2. Use code:

**Code**

@model Invent.Repository.Model.View.InfoKierowcaViewModel

@using System;

@{

    Layout = null;

    var base64 = @Model.TemplatedPDFBase64;

}

<style>

    .iframe-container {

        overflow: hidden;

        position: relative;

    }

        .iframe-container iframe {

            border: 0;

            height: 100%;

            left: 0;

            position: absolute;

            top: 0;

            width: 100%;

        }

        .iframe-container object {

            border: 0;

            height: 100%;

            left: 0;

            position: absolute;

            top: 0;

            width: 100%;

        }

    .btn-space {

        margin-right: 3px;

    }

</style>

<input type="hidden" id="pdfFile" value="@base64" />

<div class="row">

    <div class="row text-center">

        <div class="btn-group" role="group" style="margin-bottom:4px;">

            <button class="btn btn-sm btn-space" id="prev">Poprzednia strona</button>

            <button class="btn btn-sm btn-space" id="next">Następna strona</button>

        </div>

        <div class="col-sm-12">

            <span>Strona: <span id="page\_num"></span> / <span id="page\_count"></span></span>

        </div>

    </div>

</div>

<div class="iframe-container text-center">

    <canvas style="width:100%; max-width:800px" id="the-canvas"></canvas>

</div>

<script>

    (function ()

    {

        let BASE64\_MARKER = ';base64,';

        let pdfjsLib = window['pdfjs-dist/build/pdf'];

        pdfjsLib.GlobalWorkerOptions.workerSrc = window['pdfjs-dist/build/pdf.worker/src/pdf.worker.js'];

        document.getElementById('prev').addEventListener('click', onPrevPage);

        document.getElementById('next').addEventListener('click', onNextPage);

        let pdfAsDataUri = $('#pdfFile').val();

        let pdfAsArray = convertDataURIToBinary(pdfAsDataUri);

        let pdfDoc = null;

        let pageNum = 1;

        let pageRendering = false;

        let pageNumPending = null;

        let scale = 3;

        let pdfCanvas = document.getElementById('the-canvas');

        let ctx = pdfCanvas.getContext('2d');

        pdfjsLib.getDocument(pdfAsArray).promise.then(function (pdfDoc\_)

        {

            pdfDoc = pdfDoc\_;

            document.getElementById('page\_count').textContent = pdfDoc.numPages;

            renderPage(pageNum);

        });

        function convertDataURIToBinary(dataURI)

        {

            let base64Index = dataURI.indexOf(BASE64\_MARKER) + BASE64\_MARKER.length;

            let base64 = dataURI.substring(base64Index);

            let raw = window.atob(base64);

            let rawLength = raw.length;

            let array = new Uint8Array(new ArrayBuffer(rawLength));

            for (let i = 0; i < rawLength; i++)

            {

                array[i] = raw.charCodeAt(i);

            }

            return array;

        };

        function renderPage(num)

        {

            pageRendering = true;

            // Using promise to fetch the page

            pdfDoc.getPage(num).then(function (page)

            {

                let viewport = page.getViewport({ scale: scale });

                pdfCanvas.height = viewport.height;

                pdfCanvas.width = viewport.width;

                // Render PDF page into pdfCanvas context

                let renderContext = {

                    canvasContext: ctx,

                    viewport: viewport

                };

                let renderTask = page.render(renderContext);

                // Wait for rendering to finish

                renderTask.promise.then(function ()

                {

                    pageRendering = false;

                    if (pageNumPending !== null)

                    {

                        // New page rendering is pending

                        renderPage(pageNumPending);

                        pageNumPending = null;

                    }

                });

            });

            // Update page counters

            document.getElementById('page\_num').textContent = num;

        }

        function queueRenderPage(num)

        {

            if (pageRendering)

            {

                pageNumPending = num;

            } else

            {

                renderPage(num);

            }

        }

        function onPrevPage()

        {

            if (pageNum <= 1)

            {

                return;

            }

            pageNum--;

            queueRenderPage(pageNum);

        }

        function onNextPage()

        {

            if (pageNum >= pdfDoc.numPages)

            {

                return;

            }

            pageNum++;

            queueRenderPage(pageNum);

        }

    })();

</script>

Where TemplatedFile is

 $@"data:application/pdf;base64," + Convert.ToBase64String(bytes);

bytes is byte[]

Result:



# TypeScript

Nice article about C# and TypeScript: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html>

Types are just sets:



In TypeScript, this becomes very natural once you realize that every type is just a set. How do you describe a value that either belongs in the string set or the number set? It simply belongs to the union of those sets: string | number.

- Object doesn’t need to have a relation to interface if I want to use this object in place where suppose to be interface-implemented-object.

When you write:

setTimeout(() => {

if (this.el.nativeElement && this.el.nativeElement.setFocus) {

this.el.nativeElement.setFocus();

}

}, 0);

The setTimeout(..., 0) delays the execution until after the current call stack is cleared. This is often used to ensure that any view updates or DOM manipulations have been completed before executing the focus method. This technique ensures that any pending UI or event processing is completed before the callback runs.

The condition if (this.el.nativeElement && this.el.nativeElement.setFocus) checks that the element exists and **that it has a setFocus method (it’s not calling method).**

Method:

getGlobalStyle(type: 'none' | 'red' | 'black' | 'hero' | 'no-network'): string {

returns string

requires an argument “type” which must be one of: 'none' | 'red' | 'black' | 'hero' | 'no-network'

# Angular

**Signal**

signal - reactive getter function. It’s like a reactive property, but it’s a function – so you have to call it using ().

const count = signal(0);

console.log('The count is: ' + count()); //The count is: 0

**computed(() => signal);**

hasInnerEndOfShelf = computed(() =>

  !!this.listItemData().group?.some(item => item.isEndOfShelf)

);

*listItemData()* is signal - reactive getter function.

Whenever *listItemData()* changes:

* + Angular re-runs the arrow function inside computed()
  + Updates the *hasInnerEndOfShelf* signal’s value

**effect(() => { signals & code });**

effect(

  () => {

    const items = this.filteredDataWithSeparators();

    if (items?.length && !this.isLoading() && this.scrollFacade.currentIndex() < 0) {

      this.scrollFacade.setHighlightingStatusToFirst();

      this.scrollFacade.scrollToHighlighted();

    }

  },

  { allowSignalWrites: true }

);

*filteredDataWithSeparators*, *this.isLoading*, *this.scrollFacade.currentIndex()* are signals, if any of them change – the code within effect will be executed.

Normally, effect() is **read-only** for signals — it shouldn’t modify them, because that could cause infinite loops.

Here, one of methods within effect also updates the signal, the { allowSignalWrites: true } tells Angular: "Yes, I know I'm writing to signals inside this effect — it's intentional."

Because effect() automatically runs whenever dependencies change — no manual subscription or ngOnChanges() needed.

## RxJS

**RxJS** - is a library for reactive programming using Observables, often used in Angular projects.

### Subscribe on observable and effect()

subscribe() in RxJS is like an effect() in Angular Signals, but only on one variable and have to be unsubscribed manually.

this.activatedRouter.queryParams.subscribe((params) => {

  // this function runs whenever the queryParams observable \*emits\* a new value

});

It uses *observable<T>* instead of *signal<T>*

Example:

this.activatedRouter.queryParams.subscribe((params) => {

    const eanFromNav = params['ean'];

    const disableEan = params['disableEan'] === 'true';

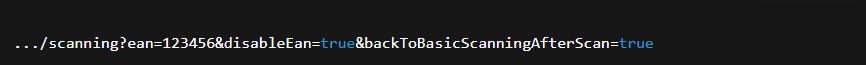
    const backToBasicScanningAfterScan = params['backToBasicScanningAfterScan'] === 'true';

});

*activatedRouter* is an instance of Angular’s *ActivatedRoute*.

*.queryParams* is an observable that emits any time the URL query parameters change.

The observable “emits” mean it produces a new value and sends it to anyone who's subscribed.

*subscribe(...)* lets you react to these changes — e.g. when someone visits:

Angular will emit:

{

  "ean": "123456",

  "disableEan": "true",

  "backToBasicScanningAfterScan": "true"

}

---------------------

    this.settingsMonitorService

      .watch$<UnknownArticleSettings>(this.UNKNOWN\_ARTICLE\_SETTINGS\_KEY)

      .pipe(skip(1), takeUntilDestroyed(this.destroyRef))

      .subscribe((settings) => {

        this.cachedSettings = settings ?? DEFAULT\_UNKNOWN\_ARTICLE\_MODAL\_SETTINGS;

      });

.watch$ is a method of settingsMonitorService ($ points that it returns observable):

    watch$<T = any>(key: string): Observable<T> {

        return new Observable<T>((subscriber) => {

*.pipe(skip(1))*

* + Skips the first emission.
  + Why? Possibly because the initial value is already loaded elsewhere (e.g., via this.loadInitialSettings()).

*takeUntilDestroyed(this.destroyRef)*

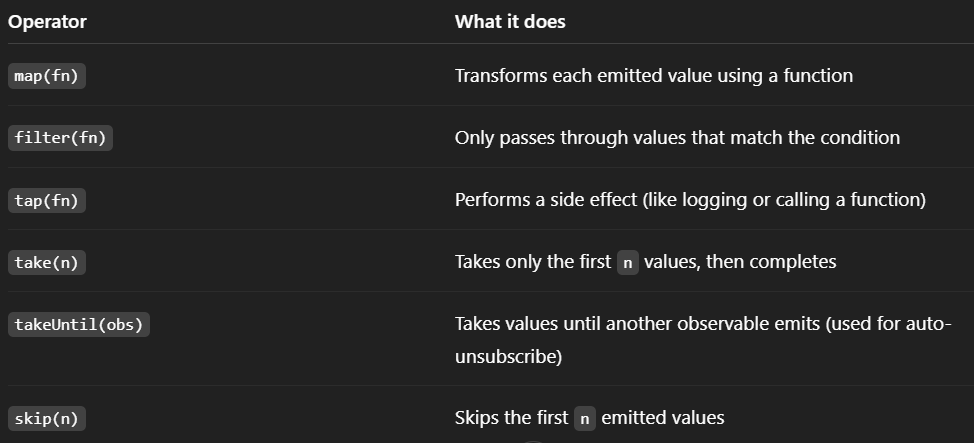
* + Automatically unsubscribes from the observable when the component is destroyed.
  + Cleaner than manually tracking a destroy$ subject.

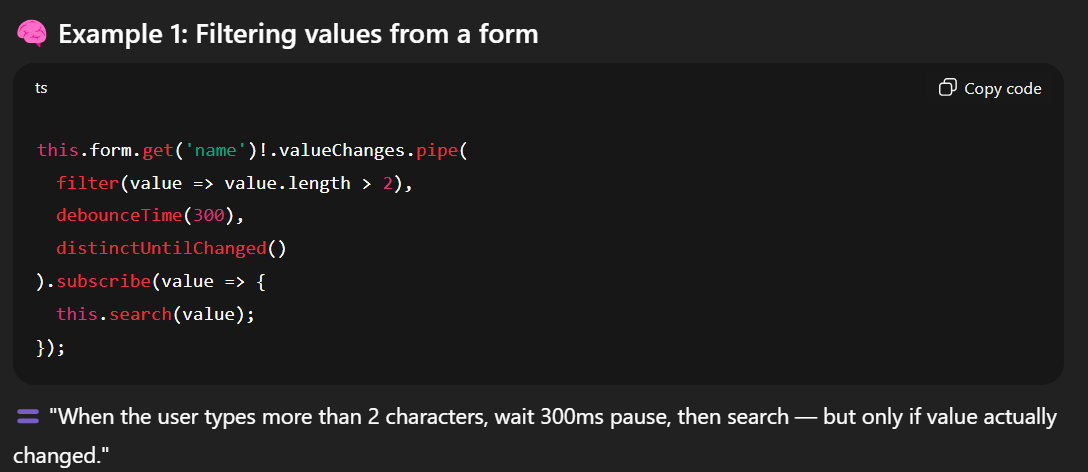
*.subscribe(...)*

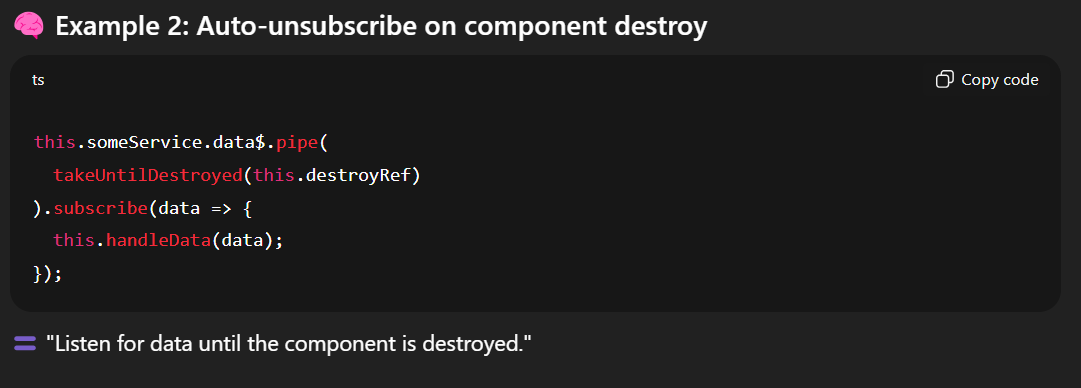
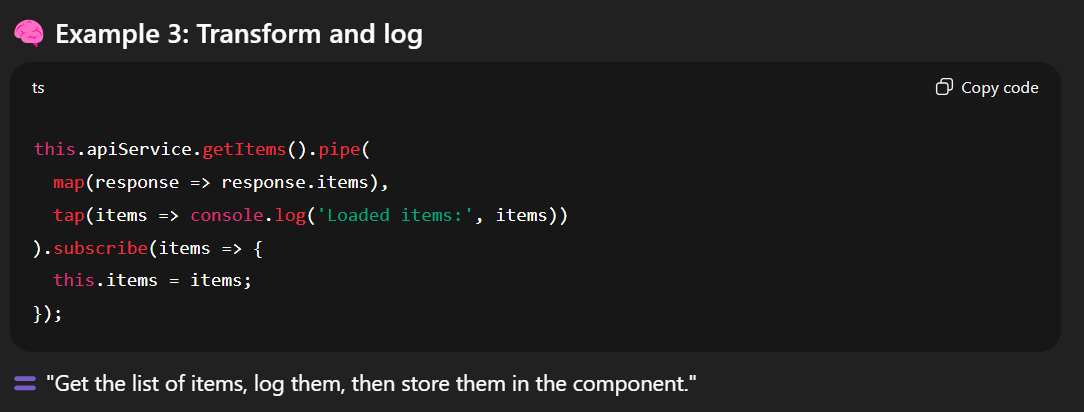
* + Reacts to each emitted value (after skipping the first).
  + Updates the local this.cachedSettings

### .Pipe

In RxJS, *.pipe()* is used to compose a chain of operators that transform, filter, or otherwise handle values emitted by an Observable.

*.pipe* works with operator to define a chain, the examples of operators:

Examples:



Why use .pipe() instead of chaining?

In RxJS v6+, the old way of chaining like observable.map(...).filter(...) is deprecated.

Now, all operators are used inside .pipe(), making the code clearer and more modular (trust me – it’s just better).