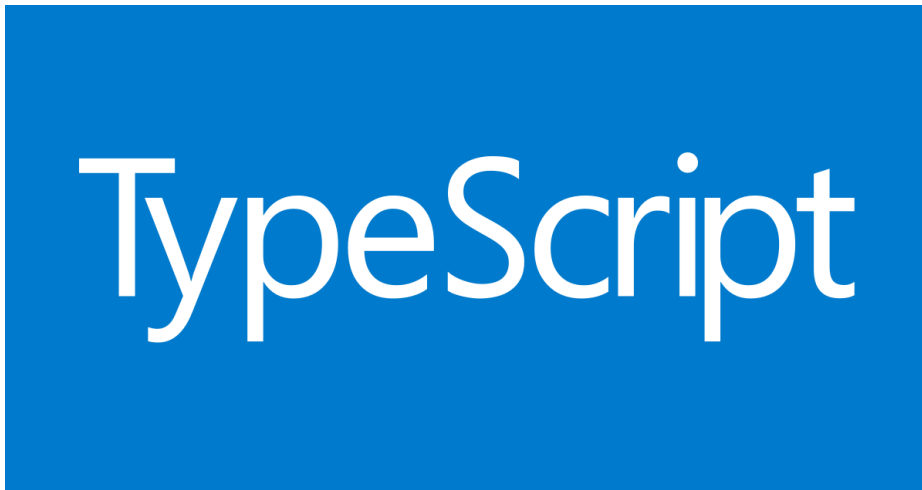


Angular 14 - 03

Typescript



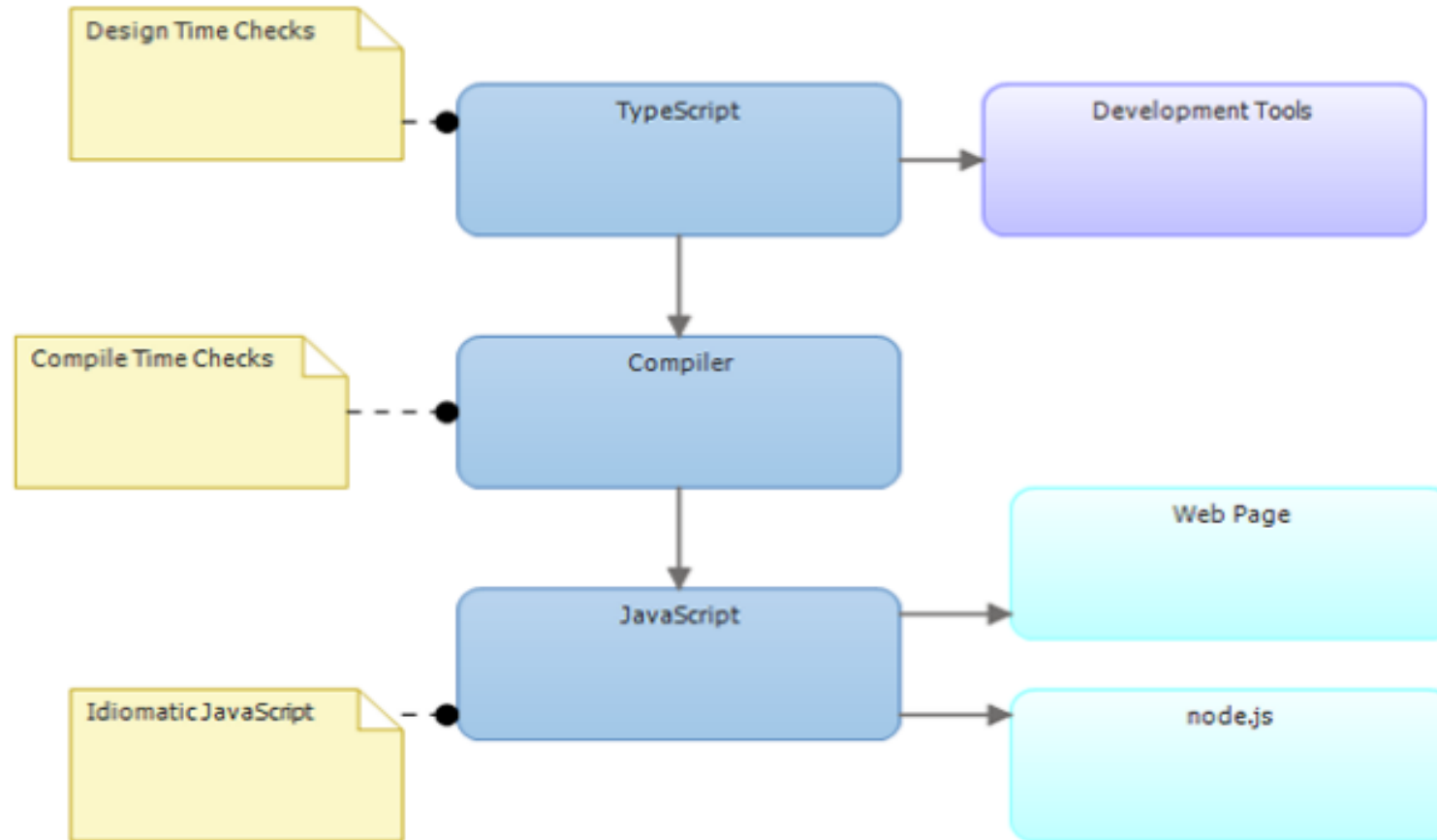
TypeScript



<https://www.typescriptlang.org/>

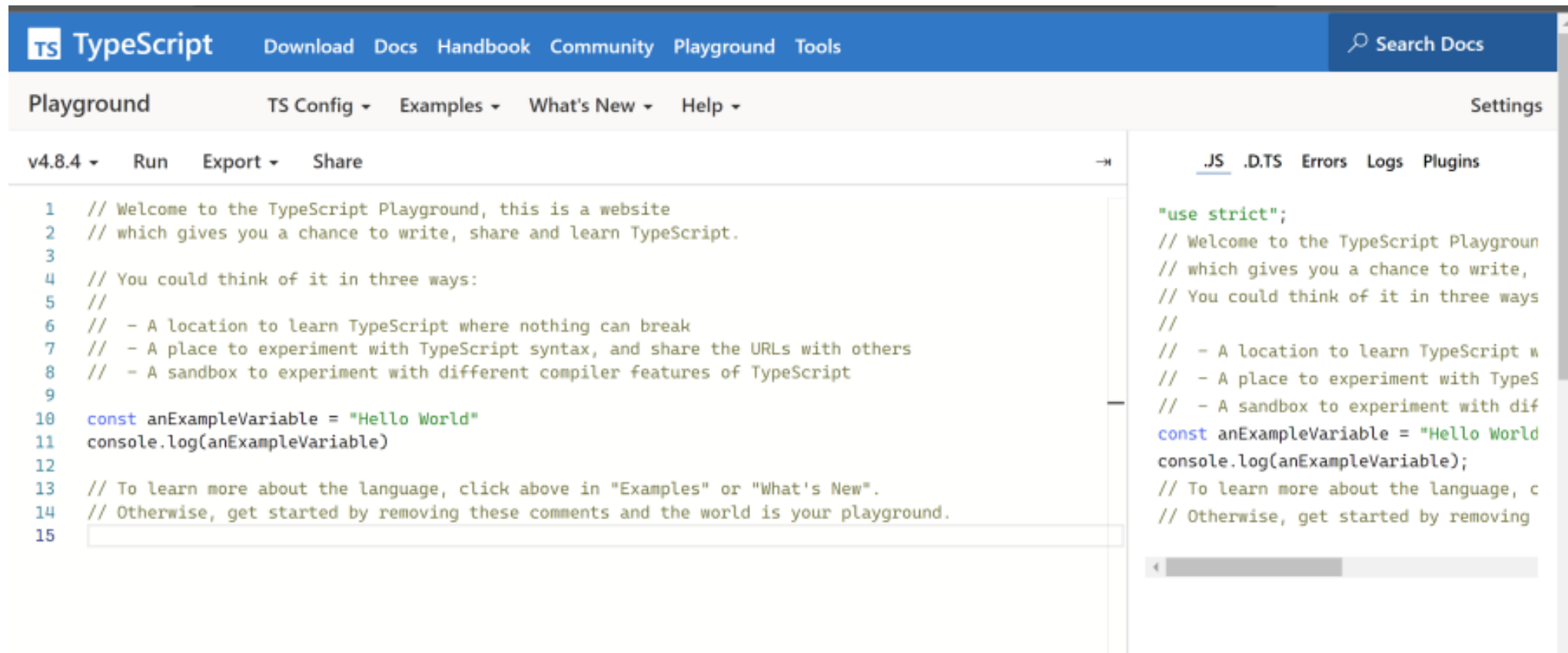
- It is an idea by **Anders Hejlsberg**, to extend the language and integrate it into IDEs.
- Totally "Open Source"
- It allows programming in a strongly typed and object-oriented syntax.
- Includes the standard itself (ECMAScript)
- "Transpiles" (generates) JavaScript manageable by all modern (and many older) browsers

Lifecycle



Typescript playground

It is a web page designed to edit and compile TypeScript code online, which allows you to share code and execute the resulting JavaScript.



Basic types

Typescript supports the following types

boolean
number
string
Arrays (Array<number>, [1,2,3], number[])
Tuples (typed array)
any
noImplicitAny
enum
void
null & undefined
union (type1 | type2 | type3 | .. | typeN)
never
Object
Assertions

<https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>

Type a variable

`<variable_name>:<type> = <value>;`

Basic types examples

```
let isDone: boolean = false;

let decimal: number = 6;

let color: string = "blue";

let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];

let x: [string, number];
x = ["hello", 10]; // OK

enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

```
let notSure: any = 4;
notSure = "maybe a string instead";

let unusable: void = undefined;

let u: undefined = undefined;
let n: null = null;

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

declare function create(o: object | null): void;

let someValue: any = "this is a string";
let strLength: number =
(<string>someValue).length;

let strLength: number = (someValue as
string).length;
```

Typed declarations (basic types)

 **TypeScript allows the declaration of types in any definition and recognizes the corresponding type from the IDE**
We say here that we are using a type annotation

```
const process = (x: string) =>{  
    const v = x + x;  
    console.log(v);  
}  
  
process('hello');
```



```
const process = (x: string) =>{  
    x.prop='john';  
    const v = x + x;  
    console.log(v);  
}
```

any

Property 'prop' does not exist on type 'string'. (2339)

[View Problem \(Alt+F8\)](#) No quick fixes available

```
const process = (x: string): number =>{  
    var v = x + x;  
    console.log(v);  
}
```




```
const process = (x: string): number =>{  
    var v = x + x;  
    console.log(v);  
    return 0;  
}
```

A function whose declared type is neither 'void' nor 'any' must return a value. (2355)

[View Problem \(Alt+F8\)](#) No quick fixes available

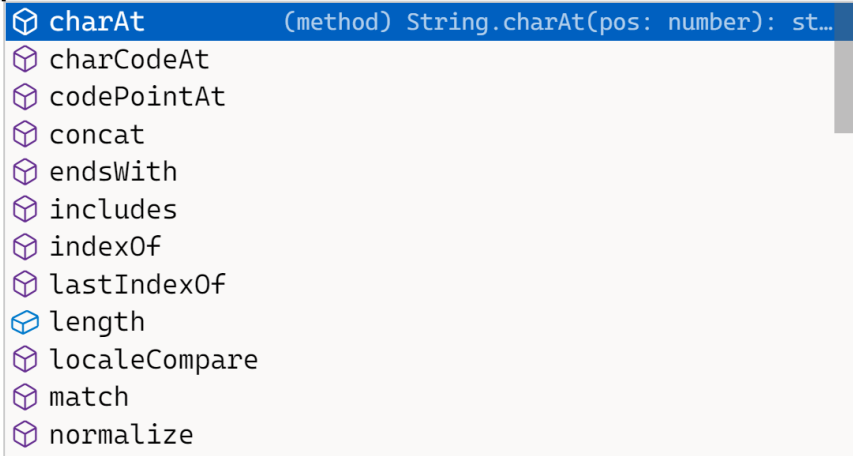
Typed declarations (basic complex types)

 Basic complex types (such as arrays) are supported in the same way: just indicate a double bracket at the end.

```
const process = (x: string[]) => {  
  x[0]. -> has "Intellisense"  
}
```




```
const process = (x: string[]) => {  
  x[0].  
}
```



The dropdown menu shows the following methods:

- charAt (method) String.charAt(pos: number): st...
- charCodeAt
- codePointAt
- concat
- endsWith
- includes
- indexOf
- lastIndexOf
- length
- localeCompare
- match
- normalize

Typed declarations (complex types)

 Interfaces are ideal for these types of declarations:

```
interface Monument {  
  name: string;  
  height: number;  
}  
  
const monuments: Monument[] = [];  
  
monuments.push({  
  name: 'Liberty',  
  height: 46,  
});
```



```
monuments.push({  
  name: 'Liberty',  
  height: 46,  
  location: 'USA'  
});
```

(property) location: `string`

Argument of type '{ name: string; height: number; location: string; }' is not assignable to parameter of type 'Monument'.

Object literal may only specify known properties, and 'location' does not exist in type 'Monument'. (2345)

[View Problem \(Alt+F8\)](#) No quick fixes available

Typed declarations (complex types)

Comparator

```
monuments.push({
  name: 'Liberty',
  height: 46,
});

monuments.push({
  name: 'North Angel',
  height: 20
});

const comparator=(a: Monument, b: Monument):number => {
  if (a.height> b.height) return -1;

  if (a.height < b.height) return 1;

  return 0;
}

var orderedMonuments= monuments.sort(comparator);

var tallerMonument= orderedMonuments[0];
console.log(tallerMonument.name);
```

Typed declarations (Lambda expressions)

 **Function declarations passed as arguments are done using lambda expressions.**

Lambda expressions can be thought of as defining contracts.

In this case, the contract indicates that process5 receives an argument of type function, which takes no parameters, and returns a string.

```
const process =(x:() => string) => {  
    x();  
}
```

Typed declarations (Objects)

 **We can define a function that receives any object**

Here, process receives as an argument an object whose double structure consists of a number and a string.

```
const process =(x: {a: number; b: string}) => {  
    return x.a;  
}
```

Interfaces

🔗 **As in OOP languages (Java, C#), we can declare interfaces:** custom types whose structure is defined by these "contracts", which are checked by the context at compile time. Those interfaces allow an external declaration of contracts, reusable throughout the application.


```
interface Thing {  
  a: number;  
  b: string;  
  c?: boolean; //optional  
}  
  
const process=(x:Thing)=>{  
  x.b  
}  
  
const result = process({a:10, b:'hello'});
```

Interfaces

 Interfaces can declare methods, and use the usual overloading techniques:


```
interface Thing {  
  a: number;  
  b: string;  
  c?: boolean; //optional  
  
  doSomething(s: string, n?: number): string;  
  doSomething(n: number): number;  
  
}  
  
const process=(x:Thing)=>{  
  x.b  
  x.doSomething(2)  
}
```

Classes

 TypeScript allows us to work with one of the most important new features of ES6: classes.

```
class Greeter {  
  greeting: string;  
  
  constructor(message: string) {  
    this.greeting = message;  
  }  
  
  greet() {  
    return "Hello, " + this.greeting;  
  }  
}  
  
const greeter = new Greeter("world");  
console.log(greeter.greet());
```

Classes

 TypeScript allows us to work with one of the most important new features of ES6: classes.

- The implementation of classes allows to include data members (**state**), as well as a functional part (**methods**)
- Within the available set of possible user methods, there is one whose name is assigned in advance: **constructor**.
- It will be the method called with the call syntax **new Class()**...
- The declaration of a constructor method must follow all the usual **OOP restrictions** in order to enhance the consistency of the class.
- The **this** keyword always refers to the active class.
- The methods are defined like those of any other function:
 - They can receive parameters of any class
 - Parameters that are not passed can be defined if we use the optional declarations.
- Additionally, many theorists claim that the **S.O.L.I.D. should apply** equally to the implementation of a class, and especially the first (**Separation of Concerns**).
- A **class may inherit** from another, **override** methods, implement **predefined interfaces**, and serve as a **base for other classes**, in order to implement a **hierarchy**.

Classes: public, private, protected and static

 **Getters/setters are a way to intercept how a member of an object is accessed.**

- **public** is the default

```
class Person {  
    public name: string;  
}
```

- **private** elements cannot be accessed from outside the class. By convention private elements are usually marked with "_" at the beginning of the name

```
class Person {  
    private _name: string;  
}
```

- **protected** elements can only be accessed by **derivatives** of a class.

```
class Person {  
    protected name: string;  
}
```

- **static** elements belong to the class instead of the instances.

```
class Grid {  
    static origin = {x: 0, y: 0};  
}
```


Classes: getters and setters

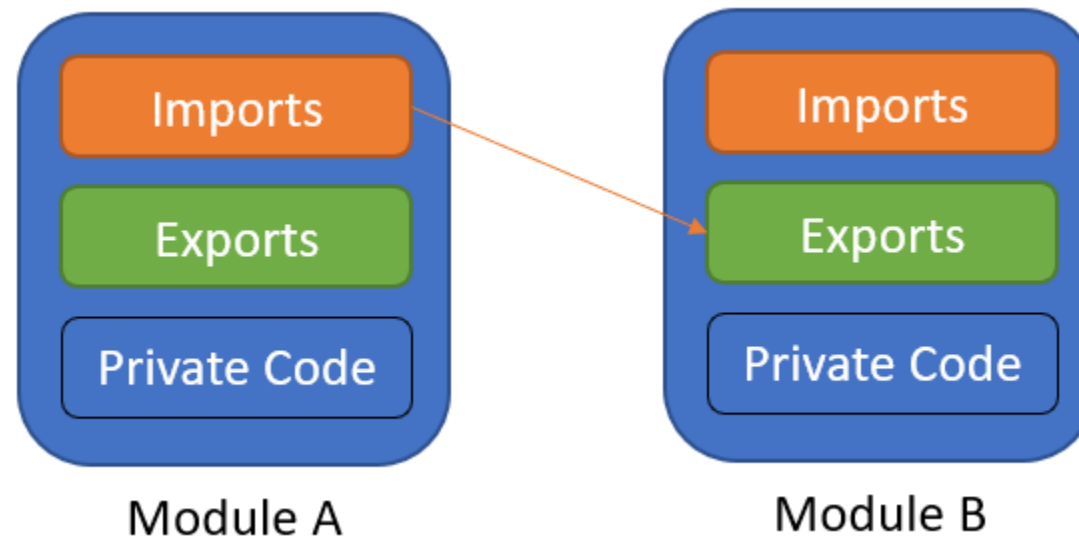
 **Getters/setters are a way to intercept how a member of an object is accessed.**

This allows finer control over how a member of an object is accessed.

```
class Employee {  
    private _name:string='';  
  
    get name(): string {  
        return this._name;  
    }  
  
    set name(newName: string) {  
        if (newName.length>3) {  
            this._name= newName;  
        } else {  
            console.log("Error: ¡Actualización no autorizada!");  
        }  
    }  
}
```

Modules

- One of the main drawbacks to using JavaScript in the construction of complex applications is the impossibility of declaring separate areas by means of what in other languages is understood as “**namespaces**”.
- Therefore, the main goal of creating modules in TypeScript is to organize the code into **separate blocks**.
- The language allows us to work with two types of modules: **internal** (called namespaces) and **external** (simply modules)
 - This was done starting with TypeScript 1.5, to align the nomenclature with the terminology used by the ECMAScript 2015 standard.



Modules: ES Module Syntax

 A file can declare a main export via export default:

```
// @filename: hello.ts
export default function helloWorld() {
  console.log("Hello, world!");
}
```

This is then **imported** via:

```
import helloWorld from "./hello.js";
helloWorld();
```

Modules: ES Module Syntax

 In addition to the default export, you can have more than one export of variables and functions via the export by omitting default:

```
// @filename: maths.ts
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;

export class RandomNumberGenerator {}

export function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}
```

These can be used in another file via the **import** syntax:

```
import { pi, phi, absolute } from "./maths.js";

console.log(pi);
const absPhi = absolute(phi);
```

An import can be renamed using a format like import {old as new}:

```
import { pi as  $\pi$  } from "./maths.js";
```

Modules: ES Module Syntax

 You can take all of the exported objects and put them into a single namespace using `*` as name:

```
import * as math from "./maths.js";  
console.log(math.pi);
```

You can import a file and not include any variables into your current module via **import `./file`**:

```
// @filename: app.ts  
import "./maths.js";  
console.log("3.14");
```

An import can be renamed using a format like `import {old as new}`:

```
import { pi as  $\pi$  } from "./maths.js";
```

Modules: TS ES Module Syntax

 Types can be exported and imported using the same syntax as JavaScript values:

```
// @filename: animal.ts
export type Cat = { breed: string; yearOfBirth: number };

export interface Dog {
  breeds: string[];
  yearOfBirth: number;
}

// @filename: app.ts
import { Cat, Dog } from "./animal.js";
type Animals = Cat | Dog;
```

Modules: TS ES Module Syntax

 TypeScript has extended the import syntax with two concepts for declaring an import of a type:

```
// @filename: animal.ts
export type Cat = { breed: string; yearOfBirth: number };

export type Dog = { breeds: string[]; yearOfBirth: number };
export const createCatName = () => "fluffy";

// @filename: valid.ts
import type { Cat, Dog } from "./animal.js";
export type Animals = Cat | Dog;

// @filename: app.ts
import type { createCatName } from "./animal.js";
const name = createCatName();
```


Modules: TS ES Module Syntax

 TypeScript 4.5 also allows for individual imports to be prefixed with `type` to indicate that the imported reference is a type:

```
// @filename: app.ts
import { createCatName, type Cat, type Dog } from "./animal.js";

export type Animals = Cat | Dog;
const name = createCatName();
```


Modules: CommonJS Syntax

 **CommonJS** is the format which most modules on npm are delivered in. Even if you are writing using the ES Modules syntax, having a brief understanding of how CommonJS syntax works will help you debug easier.

Identifiers are exported via setting the **exports** property on a global called **module**.

```
function absolute(num: number) {  
  if (num < 0) return num * -1;  
  return num;  
}
```

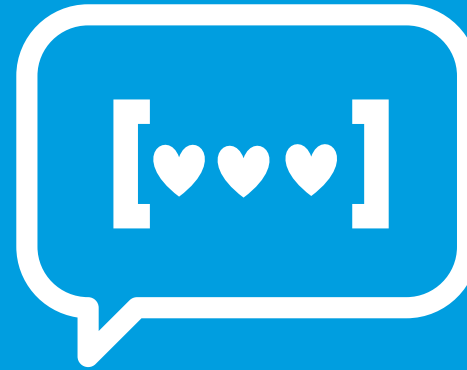
```
module.exports = {  
  pi: 3.14,  
  squareTwo: 1.41,  
  phi: 1.61,  
  absolute,  
};
```

Then these files can be **imported** via a require statement:

```
const maths = require("maths");  
maths.pi;
```



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time
and effort you have taking in answering all of that is
important in order to improve our training plans so that you
will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us:

