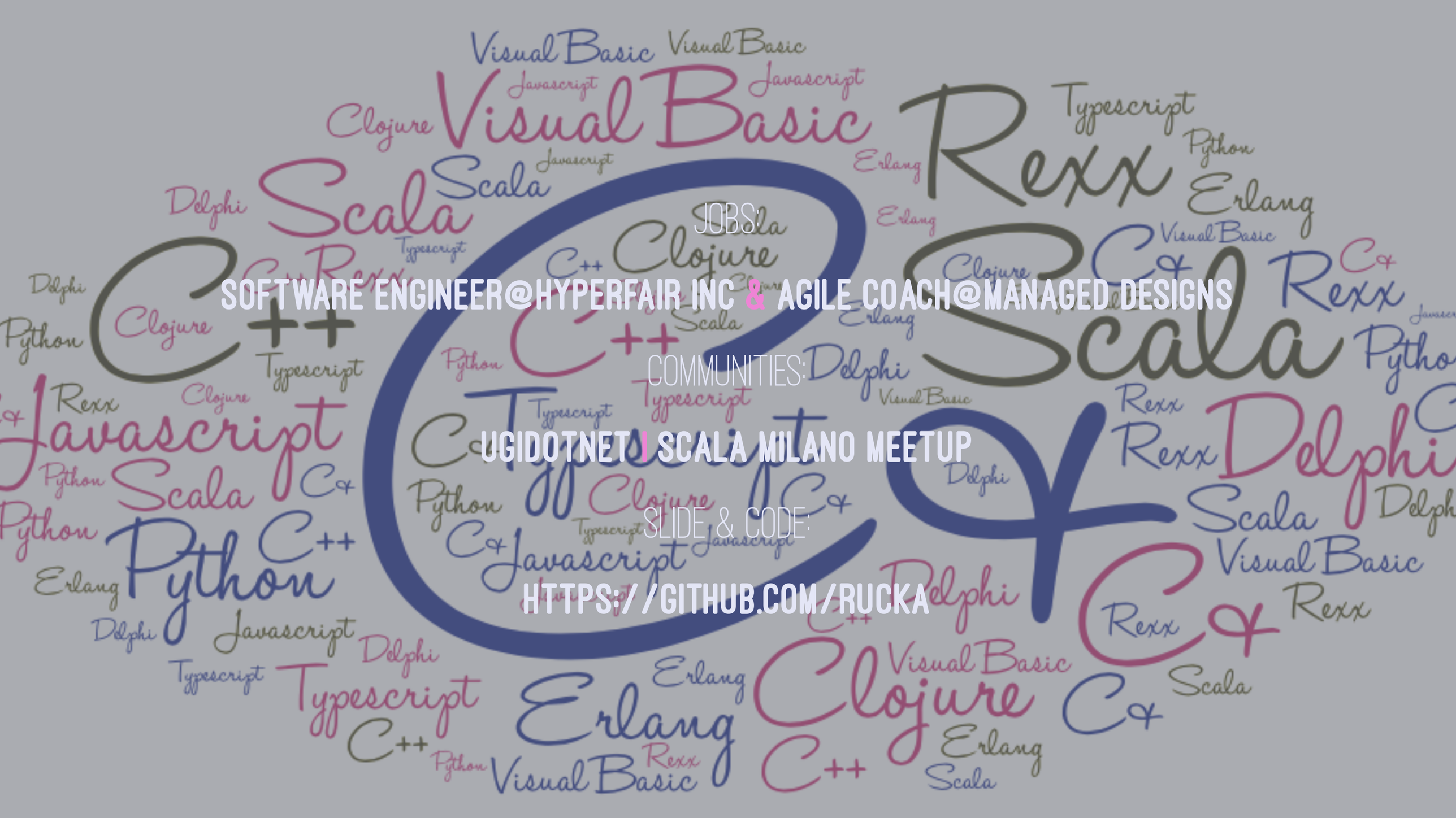
TYPESCRIPT: A STITCH IN TIME SAVES NINE

GIANLUCA CARUCCI @RUCKA

GIANLUCA.CARUCCI.ORG


JOBS:
SOFTWARE ENGINEER@HYPERFAIR INC & AGILE COACH@MANAGED DESIGNS
COMMUNITIES:
UGIDOTNET | SCALA MILANO MEETUP
SLIDE & CODE:
HTTPS://GITHUB.COM/RUCKA

Anders Hejlsberg: Introducing TypeScript

Posted: Oct 01, 2012 at 8:45AM

By: Charles

(370) | 159 comments

Average: 5

TypeScript

Starts with JavaScript

All JavaScript code is TypeScript code, simply copy and paste

All JavaScript libraries work with TypeScript

Optional Static Types, Classes, Modules

Enable scalable application development and excellent tooling

Zero cost: Static types completely disappear at run-time

Ends with JavaScript

Compiles to idiomatic JavaScript

Runs in any browser or host, on any OS

1 OCTOBER 2012


? ==
TYPESCRIPT

WEB + ? == TYPESCRIPT

WEB + C# == TYPESCRIPT ?

NON-GOAL (CUT) [1]

- EXACTLY MIMIC THE DESIGN OF EXISTING LANGUAGES. INSTEAD, USE THE BEHAVIOR OF JAVASCRIPT AND THE INTENTIONS OF PROGRAM AUTHORS AS A GUIDE FOR WHAT MAKES THE MOST SENSE IN THE LANGUAGE

[1] HTTPS://GITHUB.COM/MICROSOFT/TYPESCRIPT/WIKI/TYPESCRIPT-DESIGN-GOALS

NON-GOAL (CUT) [1]

- EXACTLY MIMIC THE DESIGN OF EXISTING LANGUAGES. INSTEAD, USE THE BEHAVIOR OF JAVASCRIPT AND THE INTENTIONS OF PROGRAM AUTHORS AS A GUIDE FOR WHAT MAKES THE MOST SENSE IN THE LANGUAGE

[1] HTTPS://GITHUB.COM/MICROSOFT/TYPESCRIPT/WIKI/TYPESCRIPT-DESIGN-GOALS

GOALS (CUT) [1]

- ALIGN WITH CURRENT AND FUTURE ECMASCRIPT PROPOSALS.
- USE A CONSISTENT. FULLY ERASABLE. STRUCTURAL TYPE SYSTEM.

[1] HTTPS://GITHUB.COM/MICROSOFT/TYPESCRIPT/WIKI/TYPESCRIPT-DESIGN-GOALS

GOALS (CUT) [1]

- ALIGN WITH CURRENT AND FUTURE ECMASCRIPT PROPOSALS.
- USE A CONSISTENT. FULLY ERASABLE. STRUCTURAL TYPE SYSTEM.

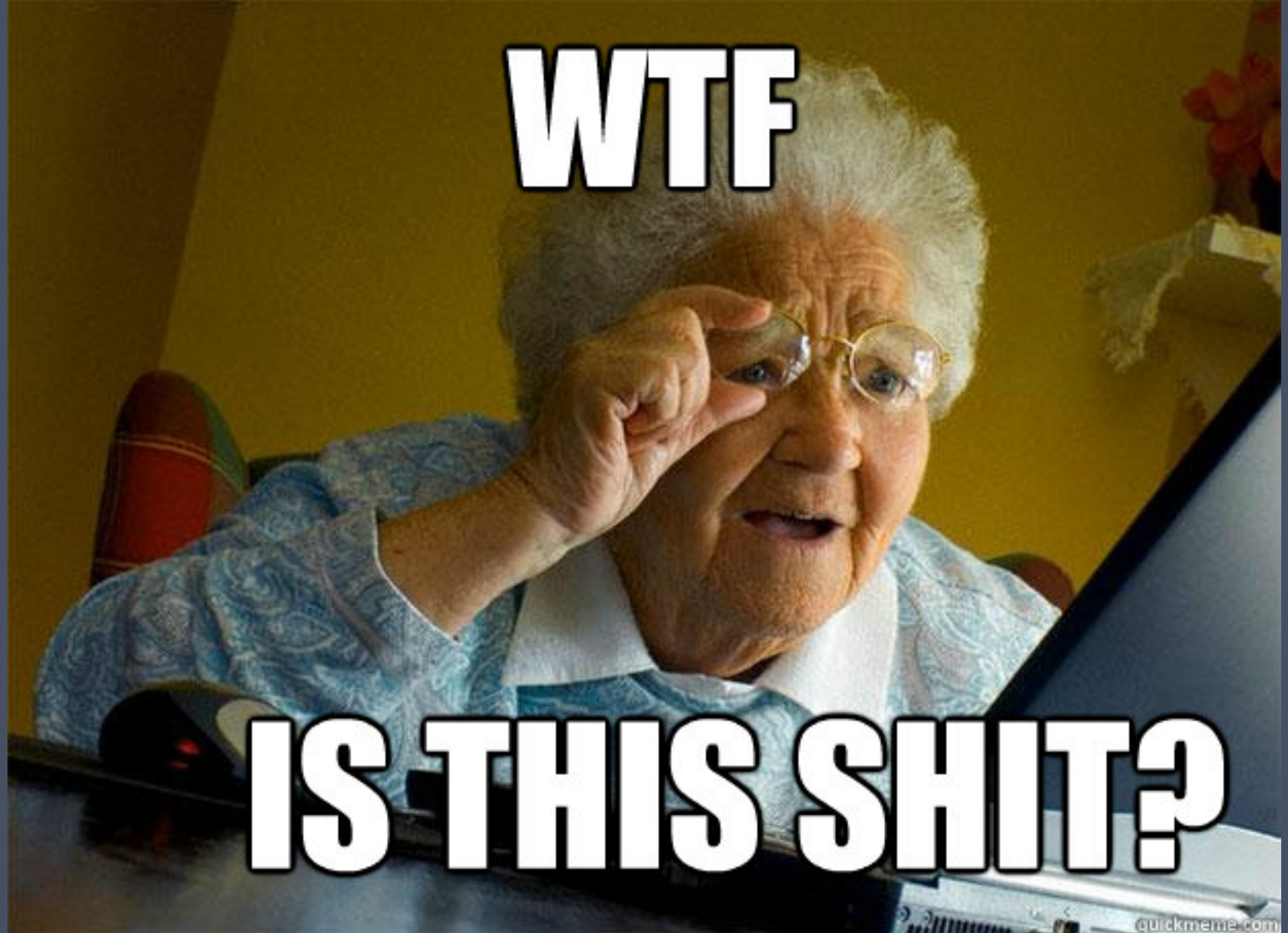
[1] HTTPS://GITHUB.COM/MICROSOFT/TYPESCRIPT/WIKI/TYPESCRIPT-DESIGN-GOALS

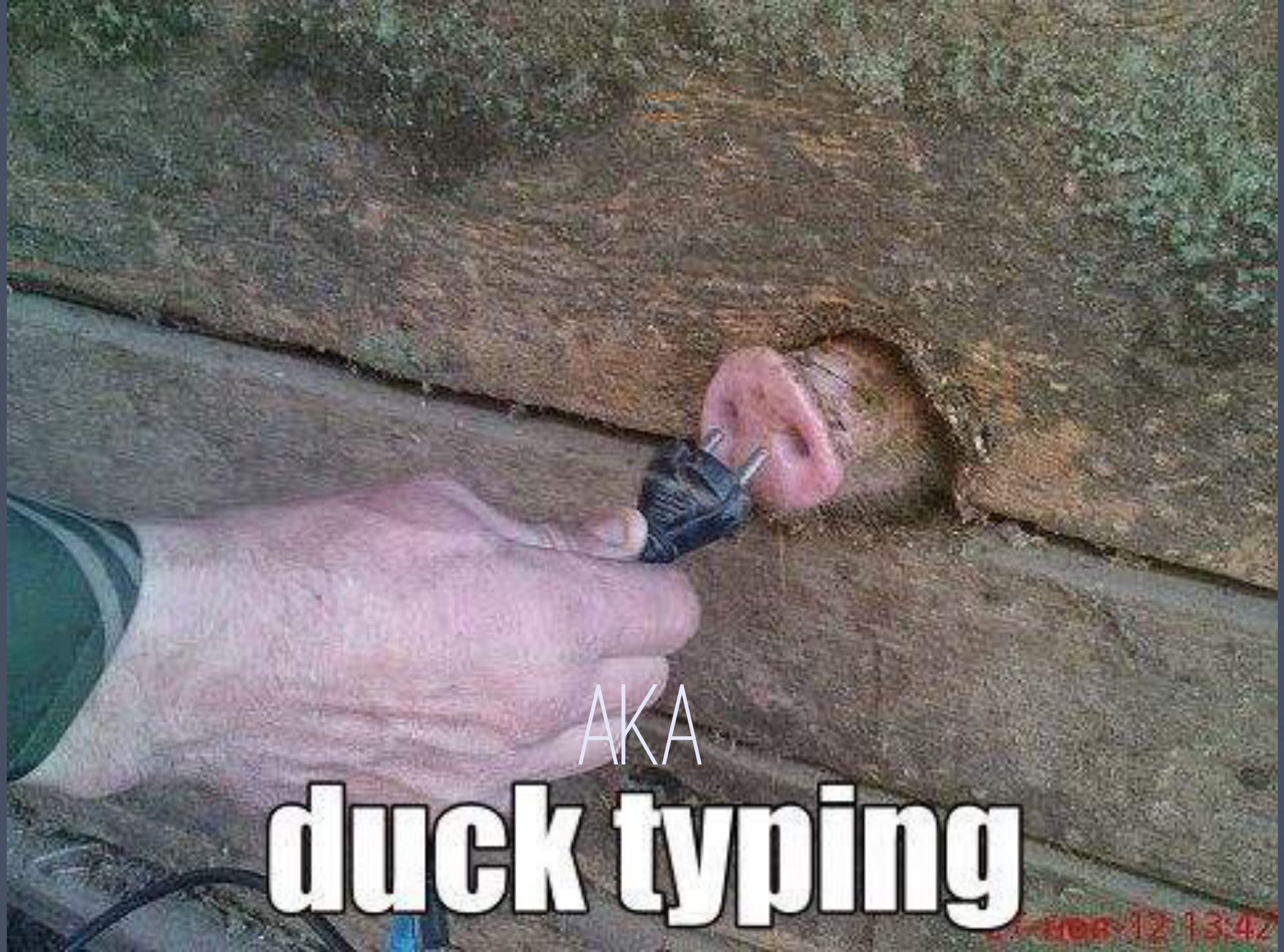
USE A CONSISTENT, FULLY ERASABLE, STRUCTURAL TYPE SYSTEM

USE A CONSISTENT, FULLY ERASABLE, STRUCTURAL TYPE SYSTEM

USE A CONSISTENT.
FULLY ERASABLE.
STRUCTURAL TYPE SYSTEM

STRUCTURAL TYPE SYSTEM


WTF
IS THIS SHIT?
quickmeme.com

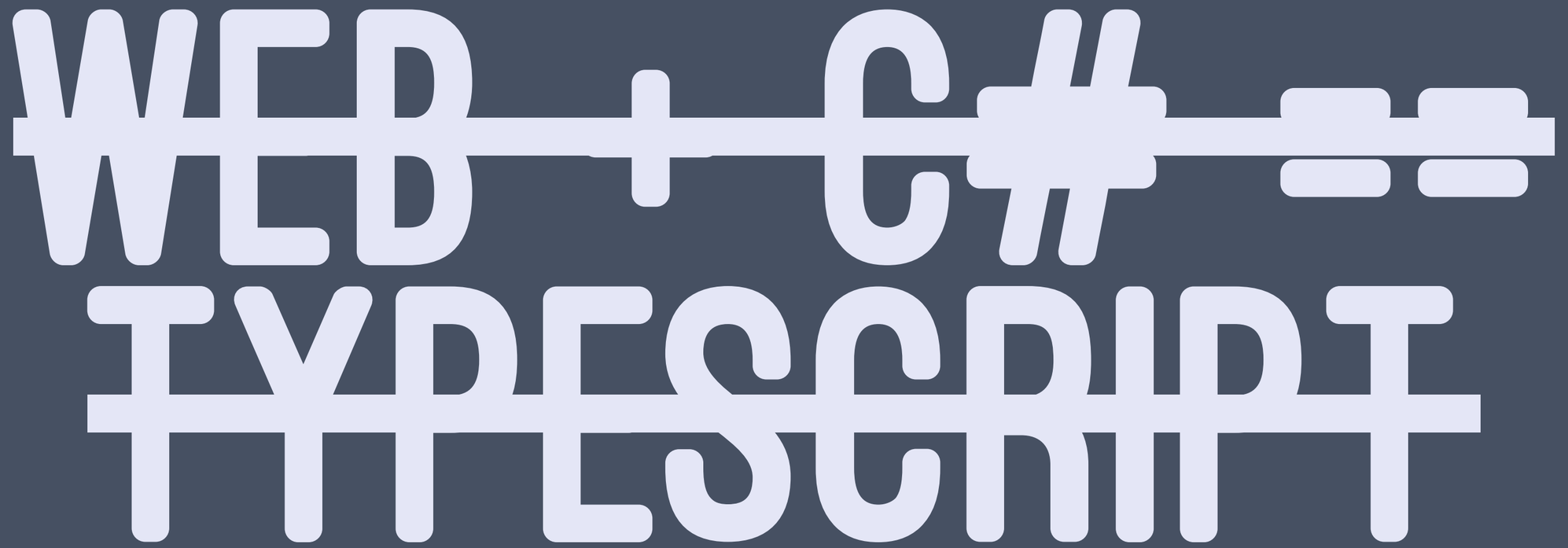

AKA
duck typing

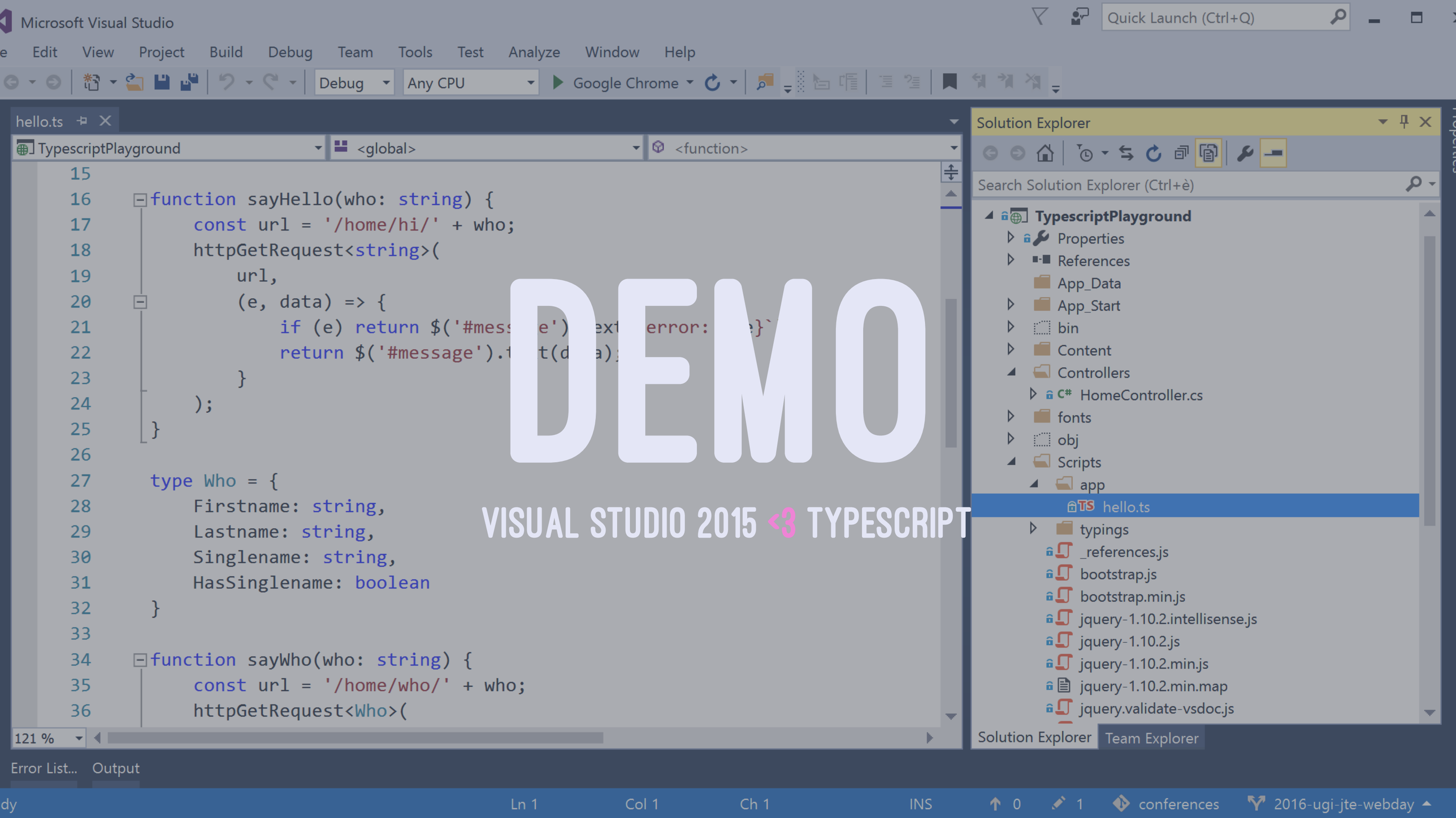
EXAMPLE

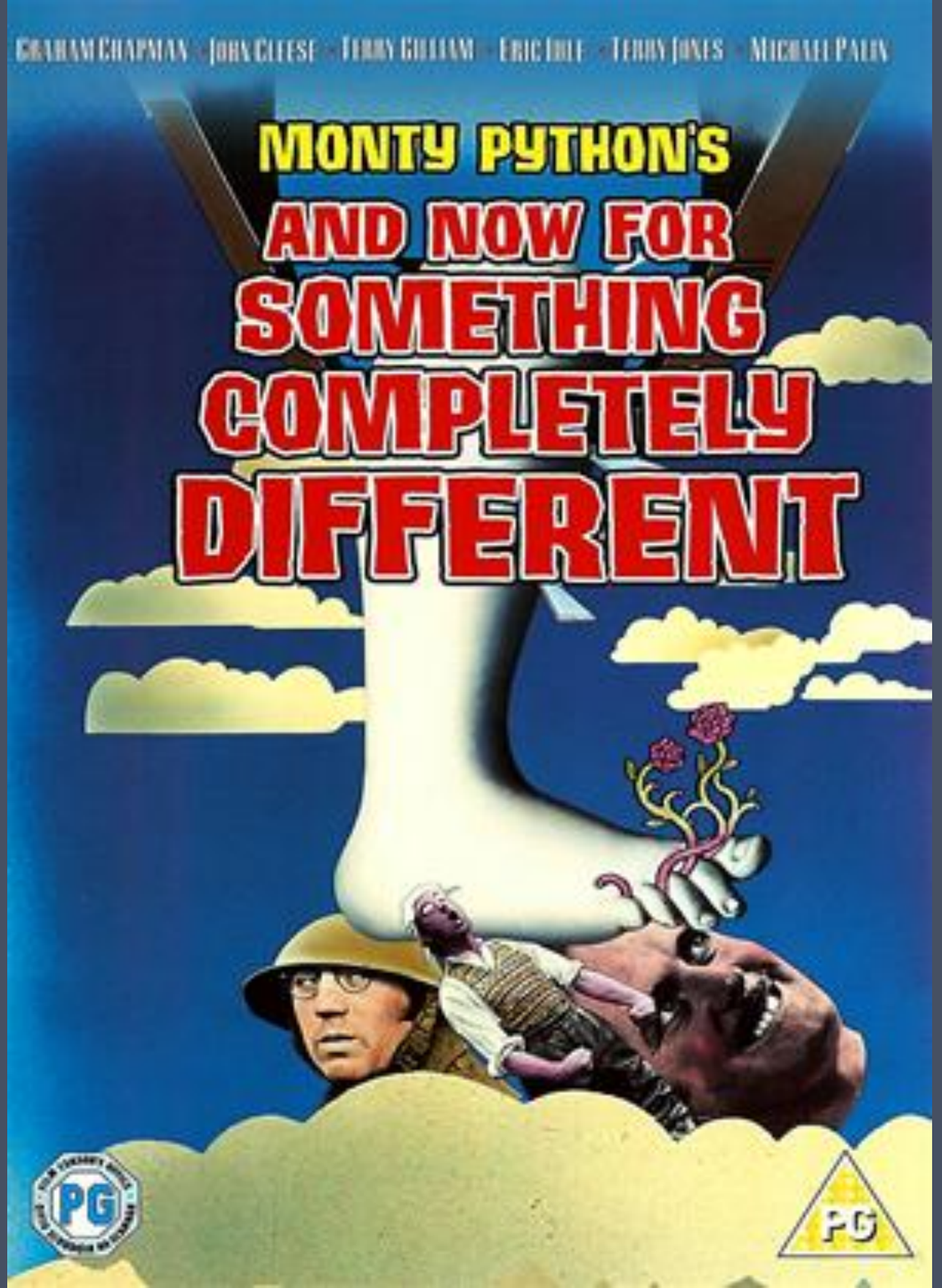
```
class Foo {public string me;}
class Bar {public string me;}

Foo foo = new Foo(){me = "foo"};
Bar bar = foo; //<--
               //Error:
               //cannot implicit convert
               //type 'Foo' to 'Bar'
System.Console.WriteLine("I am "+ bar.me);
```

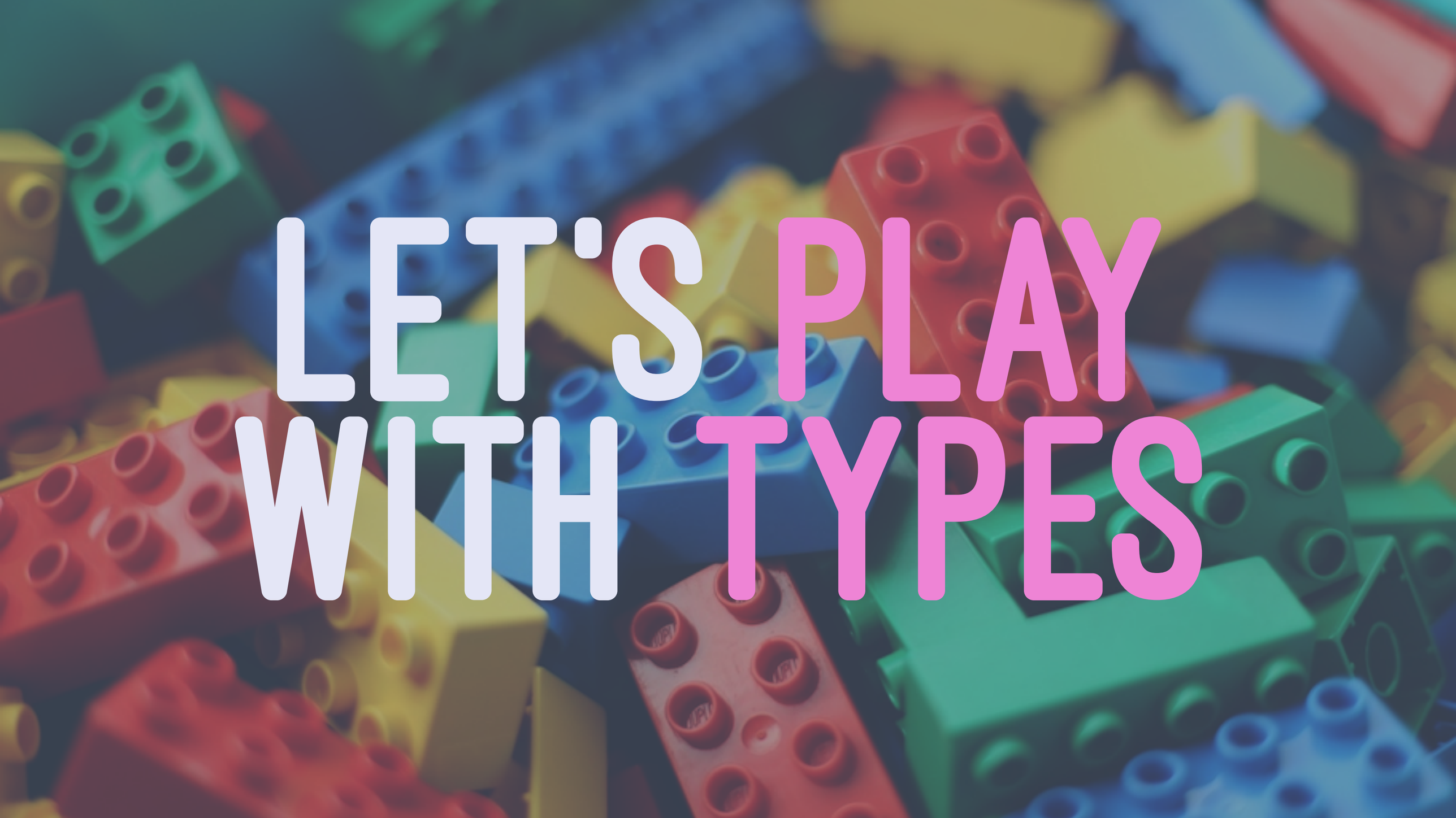
EXAMPLE

```ts
interface Foo { me: string }
interface Bar { me: string }

const foo: Foo = { me: 'foo' }
const bar: Bar = foo

console.log(`I am ${bar.me}`); //I am foo
```


WEB + C#
TYPESCRIPT


Microsoft Visual Studio
Quick Launch (Ctrl+Q)
Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
Any CPU
Google Chrome
hello.ts
TypescriptPlayground
<global>
<function>
15
16 function sayHello(who: string) {
17 const url = '/home/hi/' + who;
18 httpGetRequest<string>(
19 url,
20 (e, data) => {
21 if (e) return $('#mess e') ext error: e}`
22 return $('#message').t t(d a);
23 }
24);
25 }
26
27 type Who = {
28 Firstname: string,
29 Lastname: string,
30 Singlename: string,
31 HasSinglename: boolean
32 }
33
34 function sayWho(who: string) {
35 const url = '/home/who/' + who;
36 httpGetRequest<Who>(
DEMO
VISUAL STUDIO 2015 <3 TYPESCRIPT
121 %
Solution Explorer
Search Solution Explorer (Ctrl+è)
TypescriptPlayground
Properties
References
App_Data
App_Start
bin
Content
Controllers
HomeController.cs
fonts
obj
Scripts
app
hello.ts
typings
_references.js
bootstrap.js
bootstrap.min.js
jquery-1.10.2.intellisense.js
jquery-1.10.2.js
jquery-1.10.2.min.js
jquery-1.10.2.min.map
jquery.validate-vsdoc.js
Solution Explorer
Team Explorer
Error List...
Output
dy
Ln 1
Col 1
Ch 1
INS
0
1
conferences
2016-ugi-jte-webday


GRAHAM CHAPMAN · JOHN CLEESE · TERRY GILLIAM · ERIC IDLE · TERRY JONES · MICHAEL PALIN
MONTY PYTHON'S
AND NOW FOR
SOMETHING
COMPLETELY
DIFFERENT
PG
PG


LET'S PLAY
WITH TYPES

GOALS (CUT) [1]

> STATICALLY IDENTIFY CONSTRUCTS THAT ARE LIKELY TO BE ERRORS.

[1] HTTPS://GITHUB.COM/MICROSOFT/TYPESCRIPT/WIKI/TYPESCRIPT-DESIGN-GOALS

1. PARAMETRIC POLIMORPHISM[2]

```
interface Valid<A> { a: A }
interface Invalid<E> { errors: E[] }

function save<A>(data: Valid<A>): void {
    alert(`${JSON.stringfy(data.a)} saved!`);
}

function showErrors<E>(data: Invalid<E>): void {
    alert(`${JSON.stringfy(data.errors)} :(`);
}
```

[2] HTTPS://EN.WIKIPEDIA.ORG/WIKI/PARAMETRIC_POLYMORPHISM

2. UNION TYPES

```
type Validated<E, A> = Valid<A> | Invalid<E>
```

3.1 DISCRIMINATED (TAGGED) UNION

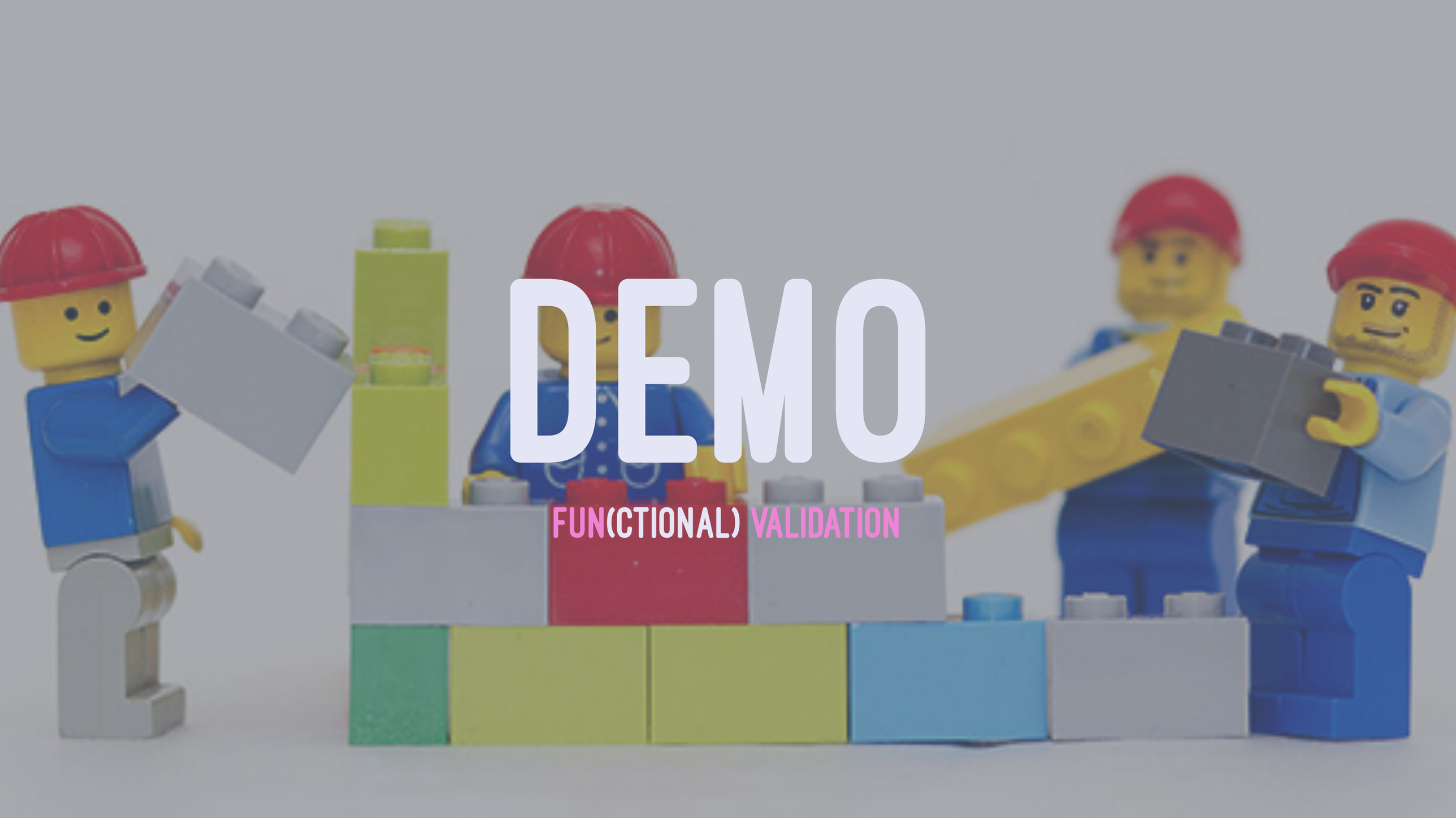
```
function isValid<A>(arg: any): arg is Valid<A> {
    return arg.a !== undefined;
}

// here validatedCustomer is Validated<E, Customer>
if (isValid(validatedCustomer)) {
    // here validatedCustomer is Valid<Customer>
    return save(validatedCustomer);
} else {
    // here validatedCustomer is Invalid<E>
    return showErrors(validatedCustomer);
}
```

3.2 DISCRIMINATED (TAGGED) UNION

```
interface NameIsTooShort { kind: 'NameIsTooShort', name: string }
interface VatCodeNotValid { kind: 'VatCodeNotValid', vatcode: string }
interface EmailIsNotValid { kind: 'EmailIsNotValid', email: string }
interface WebsiteUrlNotValid { kind: 'WebsiteUrlNotValid', url: string }

type CustomerError = NameIsTooShort | VatCodeNotValid | EmailIsNotValid | WebsiteUrlNotValid

function errorMessage(e: CustomerError): string {
    switch (e.kind) {
        case 'NameIsTooShort':
            return `Name must be at least 3 chars long, actual is ${e.name.length}`;
        case 'EmailIsNotValid':
            return `Email address '${e.email}' is not valid`;
        case 'VatCodeNotValid':
            return `VAT code '${e.vatcode}' is not valid`;
        case 'WebsiteUrlNotValid':
            return `Url '${e.url}' is not valid`;
    }
}
```

4. ITERSECTION TYPES

```
type A = {a: number}
type B = {b: number}
type I = A & B
let x: I = {a: 1, b: 2}; // OK
x = {a: 1, c: "three"}; //  Error:
// Object literal may only specify known properties,
// and 'c' does not exist in type 'I'.
```



DEMO

FUN(CTIONAL) VALIDATION

USING FUNCTIONAL PROGRAMMING

YOU ARE STILL ALIVE

(MAYBE)


UNDER THE HOOD...

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Algebraic data type

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(October 2015)* *(Learn how and when to remove this template message)*

ALGEBRAIC DATA TYPE[3]

In computer programming, more specifically functional programming and type theory, an **algebraic data type** is a kind of composite type, i.e., a type formed by combining other types.

(AKA ADT)

Two common classes of algebraic types are product types (i.e., tuples and records) and sum types, also called tagged or disjoint unions or *variant types*.[1]

The values of a product type typically contain several values, called *fields*. All values of that type have the same combination of field types. The set of all possible values of a product type is the set-theoretic product, i.e., the Cartesian product, of the sets of all possible values of its field types.

The values of a sum type are typically grouped into several classes, called *variants*. A value of a variant type is usually created with a quasi-functional entity called a *constructor*. Each variant has its own constructor, which takes a specified number of arguments with specified types. The set of all possible values of a sum type is the set-theoretic sum, i.e., the disjoint union, of the sets of all possible values of its variants. Enumerated types are a special case of sum types in which the constructors take no arguments, as exactly one value is defined for each constructor.

Values of algebraic types are analyzed with pattern matching, which identifies a value by its constructor or field names and extracts the data it contains.

Algebraic data types were introduced in Hope, a small functional programming language developed in the 1970s at Edinburgh University.[2]

Contents [hide]

[3] HTTPS://EN.WIKIPEDIA.ORG/WIKI/ALGEBRAICDATATYPE

Functor

From Wikipedia, the free encyclopedia

This article is about the mathematical concept. For other uses, see Functor (disambiguation).

In mathematics, a **functor** is a type of mapping between categories which is applied in category theory. Functors can be thought of as homomorphisms between categories. In the category of small categories, functors can be thought of more generally as morphisms.

Functors were first considered in al[illegible] are associated to topological spaces, and algebraic homomorphisms are associated to [illegible] ma[illegible]s. [illegible] functors a[illegible] use[illegible] throughout modern mathematics to relate various categories. Thus, functors are generally applicable in areas within [illegible]ics that [illegible]ory can make abstract.

The word *functor* was borrowed by [illegible] fr[illegible] philosopher Rudolf C[illegible][3] who used the term in a linguistic context:[2] see function word.

FUNCTOR^4

Contents [hide]

^{4} HTTPS://EN.WIKIPEDIA.ORG/WIKI/FUNCTOR

Definition [edit]

(A SORT OF) MONAD[5]

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Monad (functional programming)

From Wikipedia, the free encyclopedia

For the object in category theory, see Monad (category theory).

In functional programming, **monads** are a way to build computer programs by joining simple components in robust ways. A monad encapsulates values of a particular data type, creating a new type associated with a specific computation; this computation follows a set of predicates called *monad laws*. The monad represents computations with a sequential structure: a monad defines what it means to chain operations together. This allows the programmer to build pipelines that process data in a series of steps (i.e. a series of *actions* applied to the data), in which each action is decorate[illegible] with the additional processing rules provided by the monad.[1] A monad is defined by a *return* operator that creates va[illegible]es, [illegible] pipeline.

Monads allow a programming style where [illegible] omposable parts, combining in flexible ways the possible actions that can work on a particular type of data. As such, mona[illegible] icolons"; a semicolon is the operator used to chain together individual statements in many imperative programmin[illegible] extra code will be executed between the actions in the pipeline. Monads have also been explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time.[2] They can also be seen as a functional design pattern to build generic types.[3]

Purely functional programs can use monads to structure procedures that include sequenced operations like those found in structured programming.[4][5] Many common programming concepts can be described in terms of a monad structure without losing the beneficial property of referential transparency, including side effects such as input/output, variable assignment, exception handling, parsing, nondeterminism, concurrency, continuations, or domain-specific languages. This allows these concepts to be defined in a purely functional manner, without major extensions to the language's semantics. Languages like Haskell provide monads in the standard core, allowing programmers to reuse large parts of their formal definition and apply in many different libraries the same interfaces for combining functions.[6]

The name and concept comes from category theory, where monads are one particular kind of functor, a mapping between categories; however, in functional programming contexts the term *monad* is usually used with a meaning corresponding to that of the term *strong monad* in category theory.[7]

Contents [hide]

5 HTTPS://EN.WIKIPEDIA.ORG/WIKI/MONAD(FUNCTIONALPROGRAMMING)

Applicative functor

From Wikipedia, the free encyclopedia

In functional programming, specifically Haskell, an **applicative functor** is a structure that is like a monad (`return`, `fmap`, `join`) without `join`, or like a functor with `return`. Applicative functors are the programming equivalent of *strong lax monoidal functors* in category theory. In Haskell, applicative functors are implemented in the `Applicative` type class. Applicative functors were introduced in 2007 by Conor McBride and Ross Paterson in their paper *Functional Pearl: applicative programming with effects*.[1]

(A SORT OF)

APPLICATIVE^6

Relationship to monad [edit]

Due to historical accident, applicative functors were not implemented as a subclass of `Functor` nor as a superclass of `Monad`, but as a separate type class with no overlap. It turned out that in practice, there was very little overlap and after such a separation, also in GHC 7.4, it was proposed to make Applicative retroactively a subclass of Functor.[2]

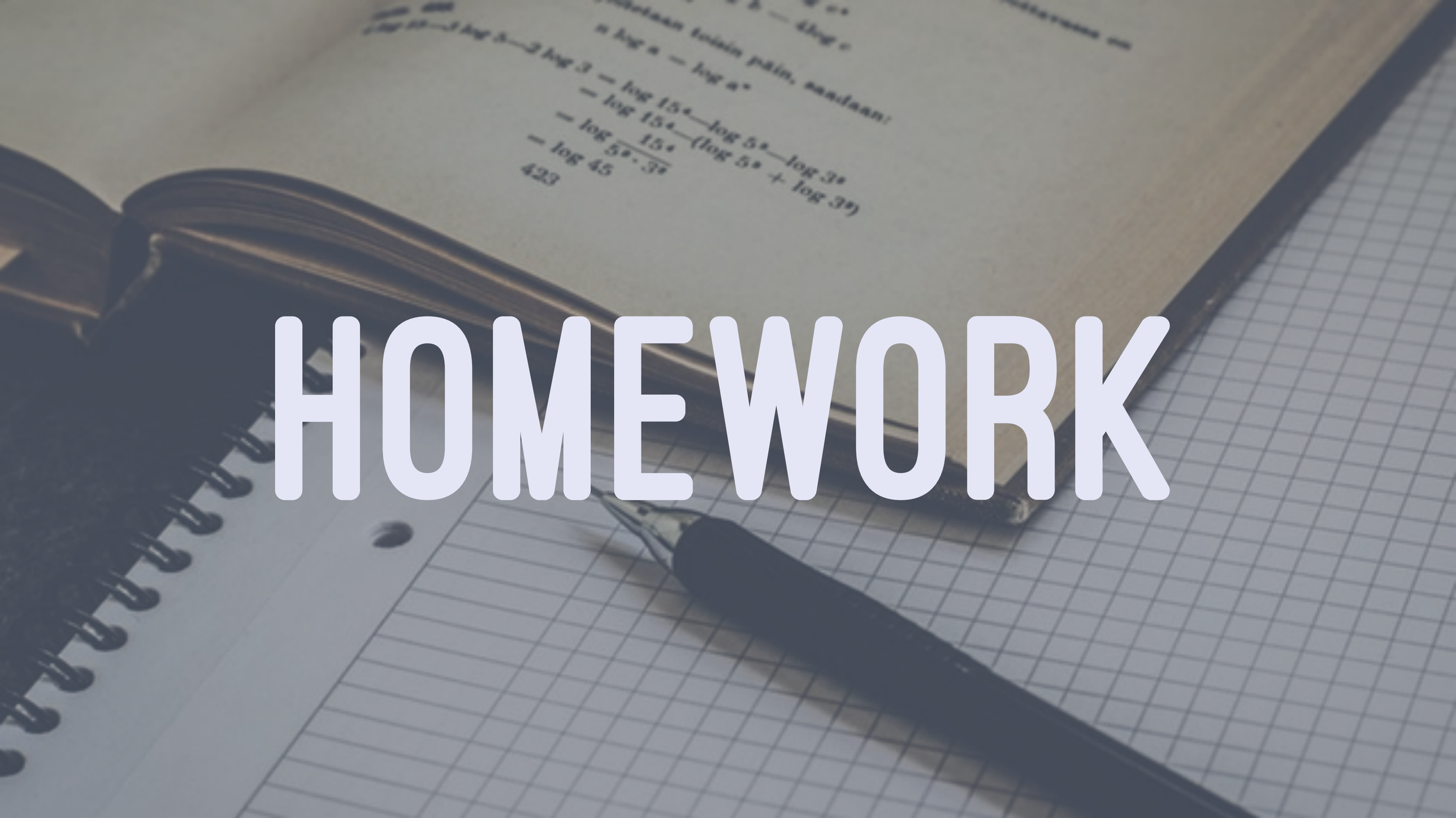
See also [edit]

- Current definition of the Applicative class in Haskell

References [edit]

1. ^ Mcbride, Conor; Paterson, Ross (2008-01-01). "Applicative programming with effects". *Journal of Functional Programming*. **18** (01): 1–13. doi:10.1017/S0956796807006326. ISSN 1469-7653.
2. ^ "Functor-Applicative-Monad Proposal - HaskellWiki". *wiki.haskell.org*. Retrieved 2016-04-06.

6 THAT'S THE ONLY WAY TO MANAGE TYPESCRIPT DEPENDECIES WITH ASP.NET CORE: HTTP://WWW.TYPESCRIPTLANG.ORG/DOCS/HANDBOOK/ASP-NET-CORE.HTML


HOMEWORK

TypeScript

TypeScript 2.0 is now available!

September 22, 2016 by Daniel Rosenwasser // 91 Comments

Today we're excited to announce the final release of TypeScript 2.0!

TypeScript 2.0 has been a joint effort from the team, with several contributions from the community, bringing several new features that enhance developer productivity, advance TypeScript's alignment with ECMAScript's evolution, provide wider support for JavaScript libraries and tools, and augment the language service that powers a first-class editing experience across platforms.

UPDATE TO TYPESCRIPT 2.0

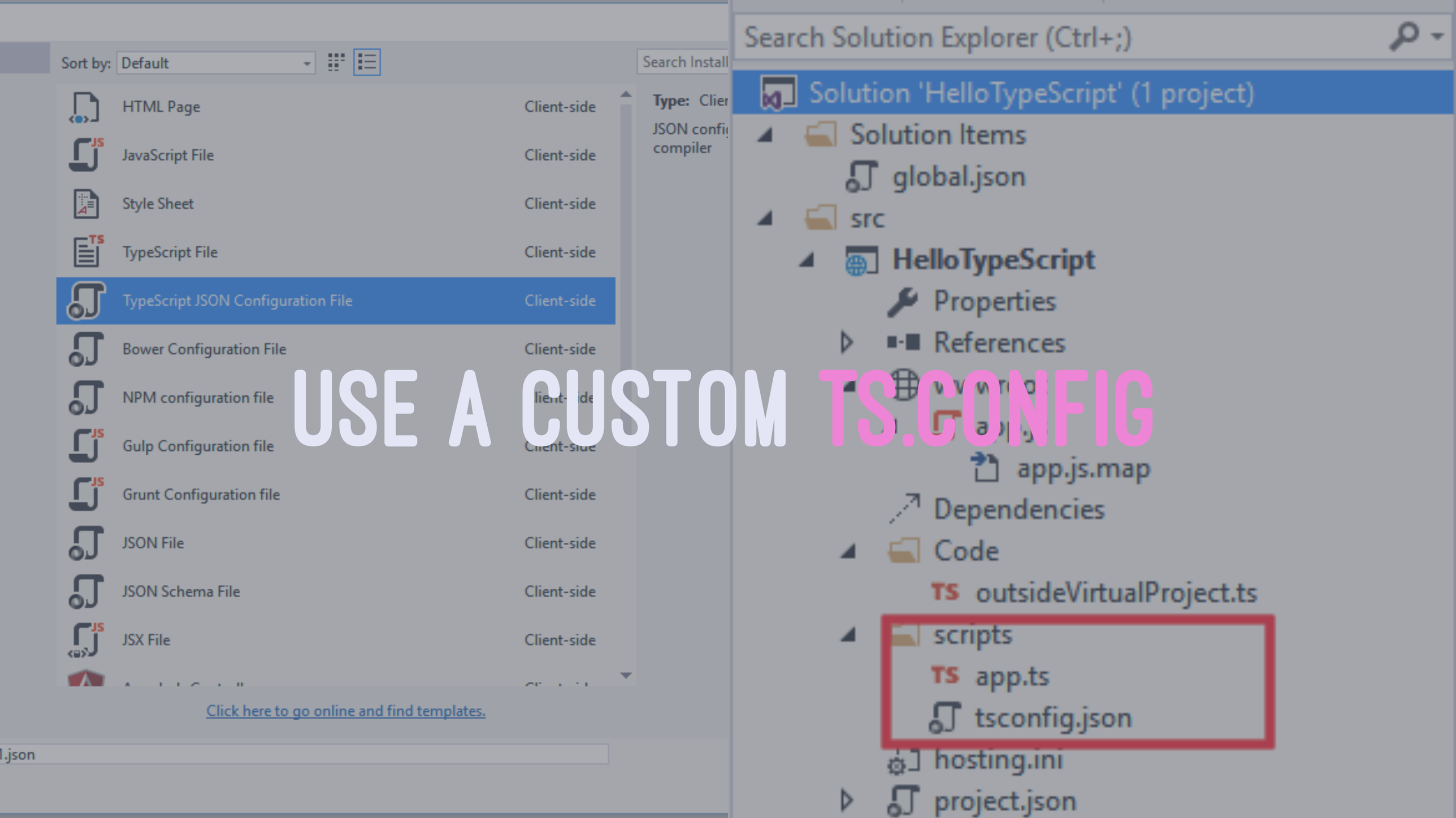
To get started, you can download TypeScript 2.0 for Visual Studio 2015 (which needs Update 3), grab it with NuGet, start using TypeScript 2.0 in Visual Studio Code, or install it with npm:

```
npm install -g typescript@2.0
```

For Visual Studio "15" Preview users, TypeScript 2.0 will be included in the next Preview release.

The 2.0 Journey

A couple of years ago we set out on this journey to version 2.0. TypeScript 1.0 had successfully shown developers the potential of JavaScript when combined with static types. Compile-time error checking saved countless hours of bug hunting, and TypeScript's editor tools gave developers a huge productivity boost as they began building larger and larger JavaScript apps. However, to be a full superset of the most popular and widespread language in the world, TypeScript still had some growing to do.



USE A CUSTOM TS.CONFIG

TypeScript 2.0

Null- and undefined-aware types

TypeScript has two special types, Null and Undefined, that have the values `null` and `undefined` respectively. Previously it was not possible to explicitly name these types, but `null` and `undefined` may now be used as type names regardless of type checking mode.

The type checker previously considered `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` were valid values of *every* type and it wasn't possible to specifically exclude them (and therefore not possible to detect erroneous use of them).

`--strictNullChecks`

`--strictNullChecks` switches to a new strict null checking mode.

In strict null checking mode, the `null` and `undefined` values are *not* in the domain of every type and are only assignable to themselves and `any` (the one exception being that `undefined` is also assignable to `void`). So, whereas `T` and `T | undefined` are considered synonymous in regular type checking mode (because `undefined` is considered a subtype of any `T`), they are different types in strict type checking mode, and only `T | undefined` permits `undefined` values. The same is true for the relationship of `T` to `T | null`.

Example

ENABLE

--strictNullChecks

FLAG

TSLint

An extensible linter for the TypeScript language.

View on GitHub | Learn More | Download .tar.gz

USE TSLINT

TSLint checks your TypeScript code for readability, maintainability, and functionality errors.

Getting Started

Local installation

```
$ npm install tslint typescript --save-dev
```

Global installation

Modules

> **A note about terminology:** It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).

PREFER MODULES OVER NAMESPACES

Introduction

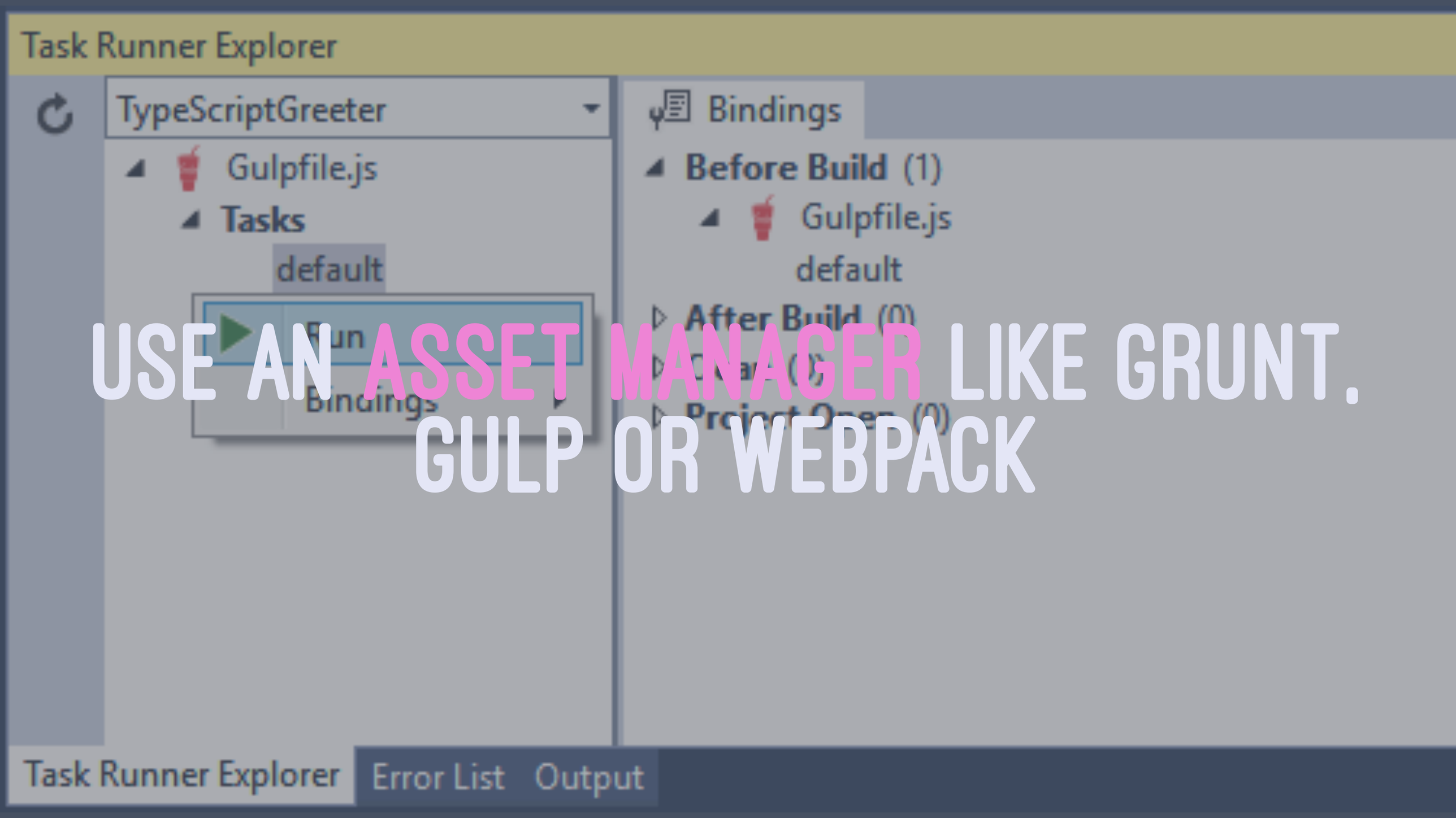
Starting with the ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

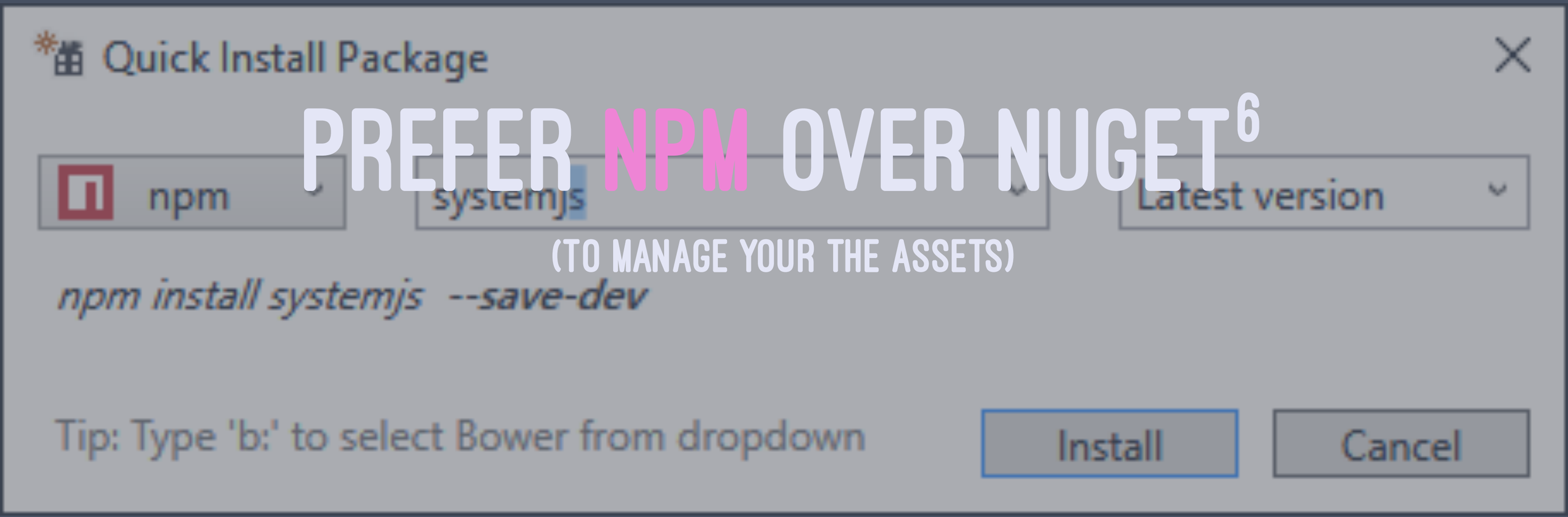
Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module.



USE AN ASSET MANAGER LIKE GRUNT. GULP OR WEBPACK



PREFER NPM OVER NUGET[6]

(TO MANAGE YOUR THE ASSETS)

[6] THAT'S THE ONLY WAY TO MANAGE TYPESCRIPT DEPENDECIES WITH ASP.NET CORE: HTTP://WWW.TYPESCRIPTLANG.ORG/DOCS/HANDBOOK/ASP-NET-CORE.HTML

THE RIGHT TOOL FOR THE RIGHT JOB


QUESTIONS?


Thank You!