UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 16

TYPESCRIPT: JAVASCRIPT WITH SYNTAX FOR TYPES

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

JAVASCRIPT AND TYPES

JavaScript features a peculiar type system we learned to ~~love~~ accept

- It is **dynamically** typed

```
let x;
console.log(typeof(x)); //undefined
x = 1.25e7;
console.log(typeof(x)); //number
x = "Hello Web Technologies!";
console.log(typeof(x)); //string
```

- It is also **weakly** typed and loves performing implicit casts

```
let y = "100" - 1 + "42"; //string - number -> number, number + string -> string
console.log(`${y}: ${typeof(y)}`);//9942: string
```

JAVASCRIPT AND TYPES

- These characteristics make the language **practical** for small web page manipulation tasks
  - It just works, no need to be verbose, declaring types or explicit casts
- However, as the complexity of the programs grows:
  - It’s easier to introduce tricky bugs
  - Not the best developer experience (little IDE support, very few errors caught in the IDE and not at runtime)
  - Code becomes less maintainable and hard to understand

JAVASCRIPT: DAILY DEVELOPMENT TALES

```
let message = "Hello Web Technologies!";
console.log(message.toLowerCase()); //hello web technologies!
message(); //TypeError: message is not a function (at runtime)
```

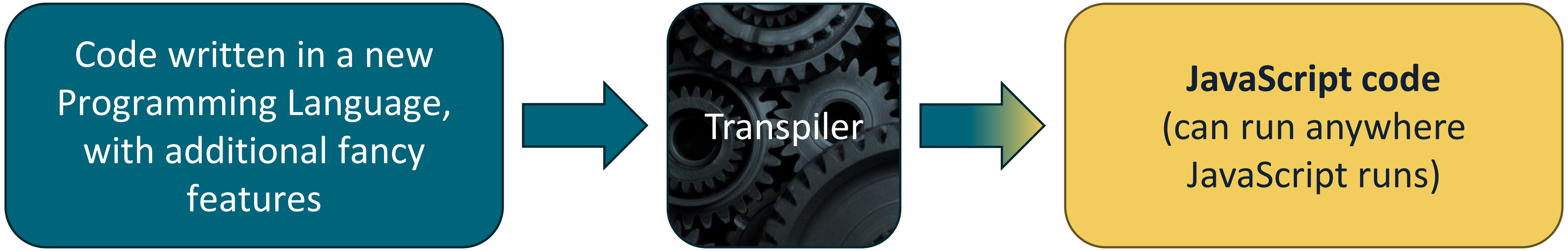
```
let employee = {name: "Jordan Belfort", role: "Stockbroker"};
employee.nome = "The Wolf of Wall Street"; //no error at all
console.log(employee.name); //Jordan Belfort
```

```
let team = ["Jordan Belfort", "Donnie Azoff", "Chester Ming"];
team.add("Nicky 'Rugrat' Koskoff"); //TypeError: team.add is not a function
console.log(team);
```

It would be nice if we could catch these bugs **before** runtime!

TRANSPILERS

- JS started being used also for increasingly complex programs
- Developers felt the need to address these limitations
- JS was already widely supported, a new language was not feasible
- Lots of new languages that compiled to standard JS popped up!
  - CoffeeScript, TypeScript, Dart, Elm, … (~350 more are listed on this page)



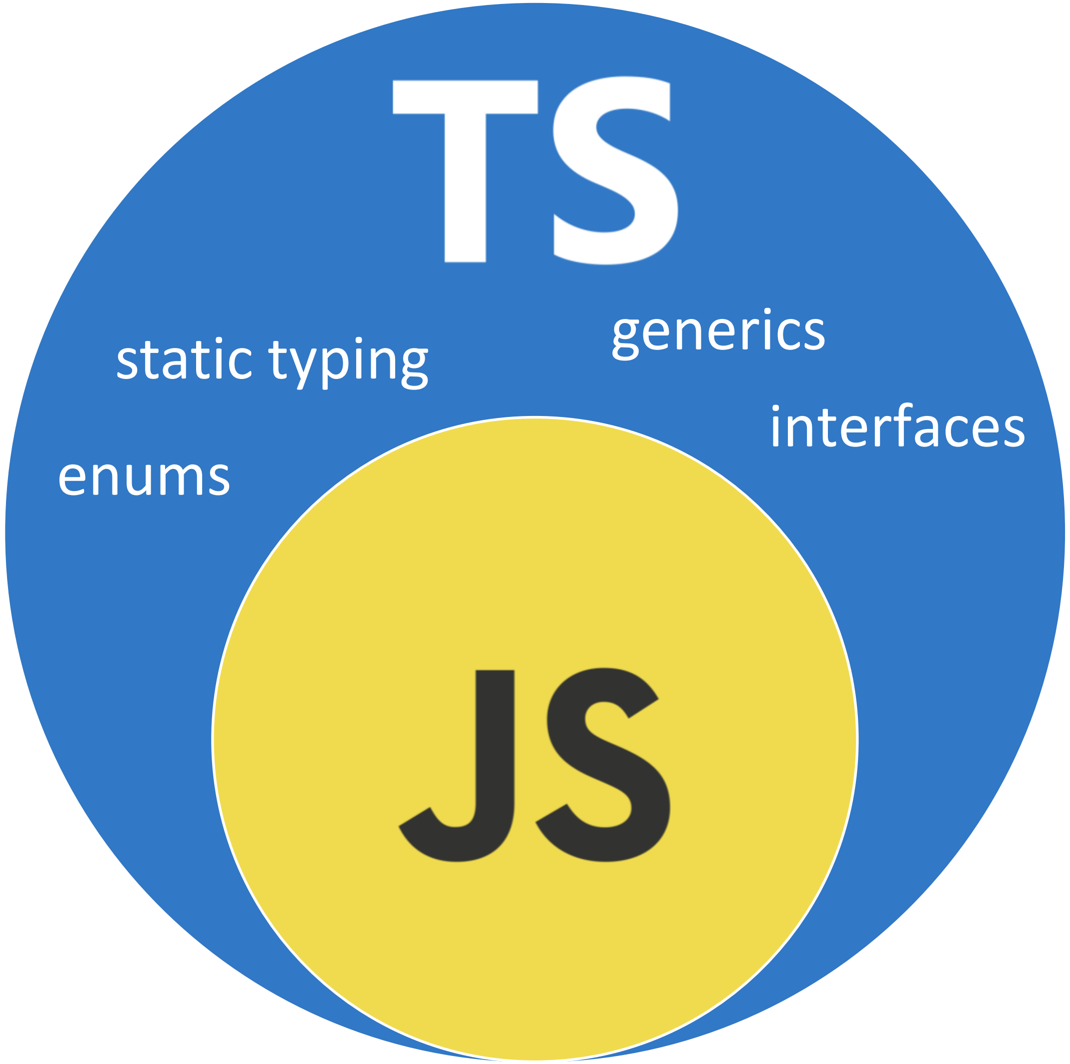
Code written in a new Programming Language, with additional fancy features

Transpiler

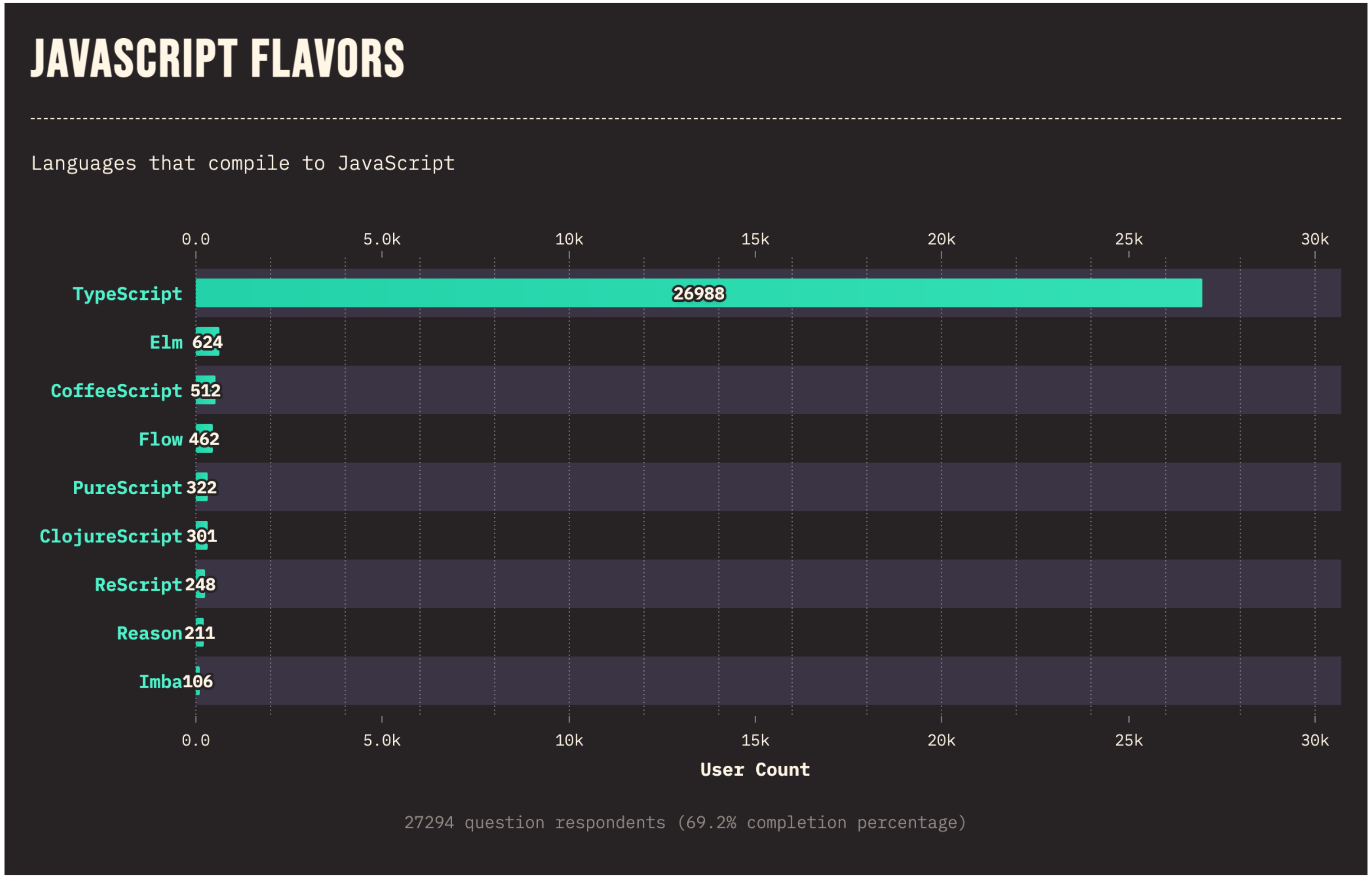
JavaScript code
(can run anywhere JavaScript runs)

TYPESCRIPT

- Born in 2012, designed and developed by Microsoft
- Core dev: Anders Hejlsberg
- Superset of JavaScript
- … with static typing and more!
- Compiles to JavaScript
- By far, the **most popular** JS flavor



TYPESCRIPT: ADOPTION



From the State of JavaScript 2022 survey

INSTALLING TYPESCRIPT

TypeScript can be installed as an **npm** package

```
@luigi → D/O/T/W/2/e/TypeScript $ npm install -g typescript

added 1 package in 9s
```

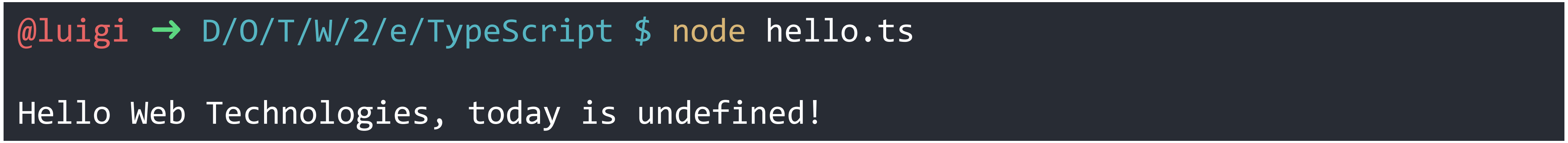
- The above command makes the **tsc** compiler globally available
- You can use **npx** or similar tools to run tsc from a local `node_modules` package
- TypeScript files typically have a **.ts** extension

HELLO TYPESCRIPT

```
// hello.ts file
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
  console.log(`Hello ${entity}, today is ${date}!`);
}

greet("Web Technologies");
```

- Standard JavaScript code is also valid TypeScript code
- We can run the above script as a plain JavaScript file with Node.js



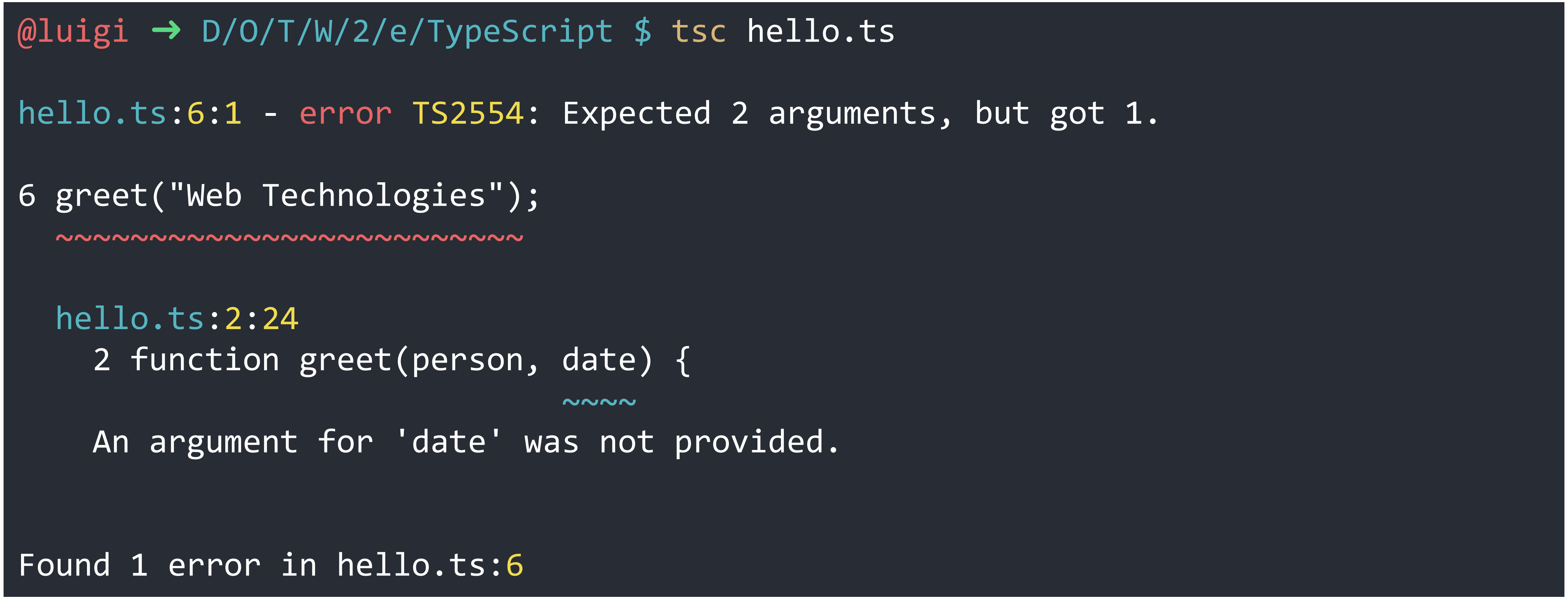
```
@luigi → D/O/T/W/2/e/TypeScript $ node hello.ts

Hello Web Technologies, today is undefined!
```

- Looks like we forgot the second parameter!

HELLO TYPESCRIPT

What if we try to compile **hello.ts** with **tsc**?



```
@luigi → D/O/T/W/2/e/TypeScript $ tsc hello.ts

hello.ts:6:1 - error TS2554: Expected 2 arguments, but got 1.

6 greet("Web Technologies");
  ~~~~~~~~~~~~~~~~~~~~~~~~~

  hello.ts:2:24
    2 function greet(person, date) {
                             ~~~~
    An argument for 'date' was not provided.


Found 1 error in hello.ts:6
```

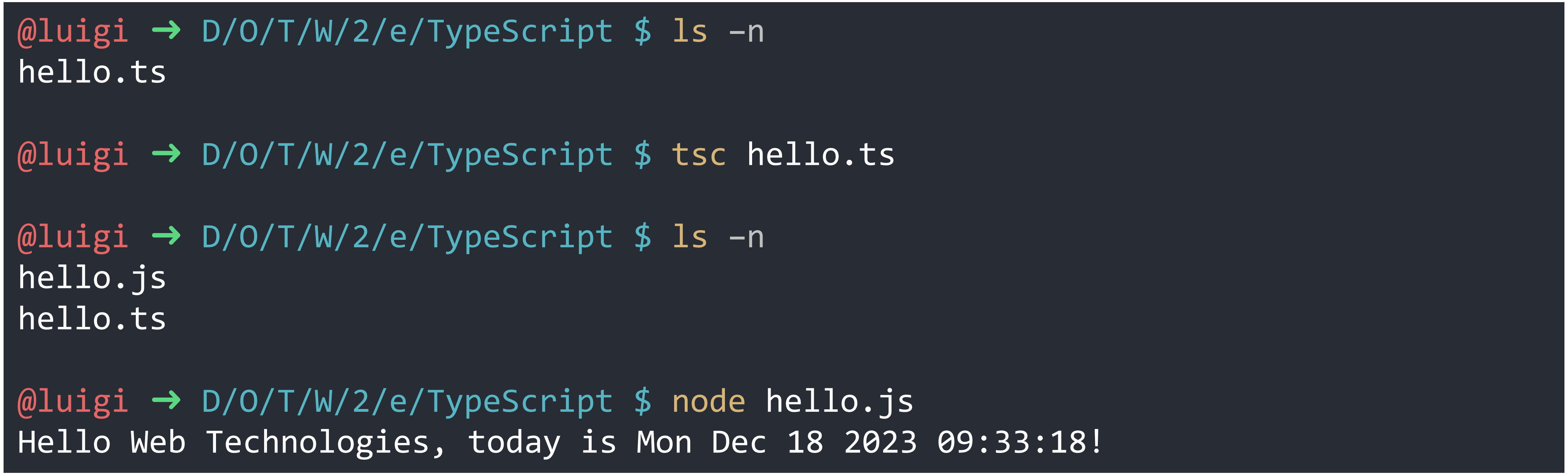
HELLO TYPESCRIPT

- The TypeScript compiler detected the bug
- ..even though we’ve only written standard JavaScript code so far!
- Let’s fix the bug

```
// hello.ts file
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
  console.log(`Hello ${entity}, today is ${date}!`);
}

greet("Web Technologies", Date());
```

HELLO TYPESCRIPT



```
@luigi → D/O/T/W/2/e/TypeScript $ ls -n
hello.ts

@luigi → D/O/T/W/2/e/TypeScript $ tsc hello.ts

@luigi → D/O/T/W/2/e/TypeScript $ ls -n
hello.js
hello.ts

@luigi → D/O/T/W/2/e/TypeScript $ node hello.js
Hello Web Technologies, today is Mon Dec 18 2023 09:33:18!
```

The **tsc** compiler transpiled **hello.ts** to the newly-created **hello.js**, which can be executed with Node as usual.

HELLO TYPESCRIPT: GENERATED CODE

hello.ts

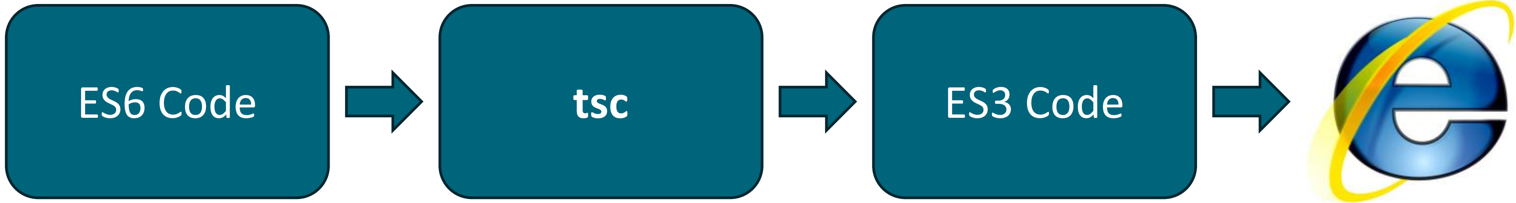
```
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
  console.log(`Hello ${entity}, today is ${date}!`);
}
greet("Web Technologies", Date());
```

hello.js

```
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
    console.log("Hello ".concat(entity, ", today is ").concat(date, "!"));
}
greet("Web Technologies", Date());
```

TYPESCRIPT: DOWNLEVELING

- **tsc** preserved comments and indentations. Transpiled code looks like it has been written by a human.
- The template string was changed to multiple **.concat()** invocations
- Why is that?
  - Template strings are an ES6 feature, and, by default, TS targets ES3 (1999)
  - TS can rewrite code from newer versions to older ones (**downleveling**)
  - We can still use the nice new features, but our code can run also on old browsers supporting only ES3!



ES6 Code → **tsc** → ES3 Code →

TYPESCRIPT: DOWNLEVELING

- When compiling with tsc, we can specify the **target** version to use

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc —target es6 hello.ts
```

- The above command compiles the TS file to ES6 JavaScript

hello.js

```
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
    console.log(`Hello ${entity}, today is ${date}!`);
}
greet("Web Technologies", Date());
```

TYPESCRIPT: DECLARING TYPES

- So far, we’ve only written plain old JavaScript code
- TS allows us to declare **types** for variables, params and return values
- The most common primitive types in TS are **string**, **number**, **boolean**

primitive_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

console.log(`
  Welcome to ${courseName}!
  Credits: ${credits} ETCS
  Status: ${isGreat ? "Great" : "Could be better"}
`);
```

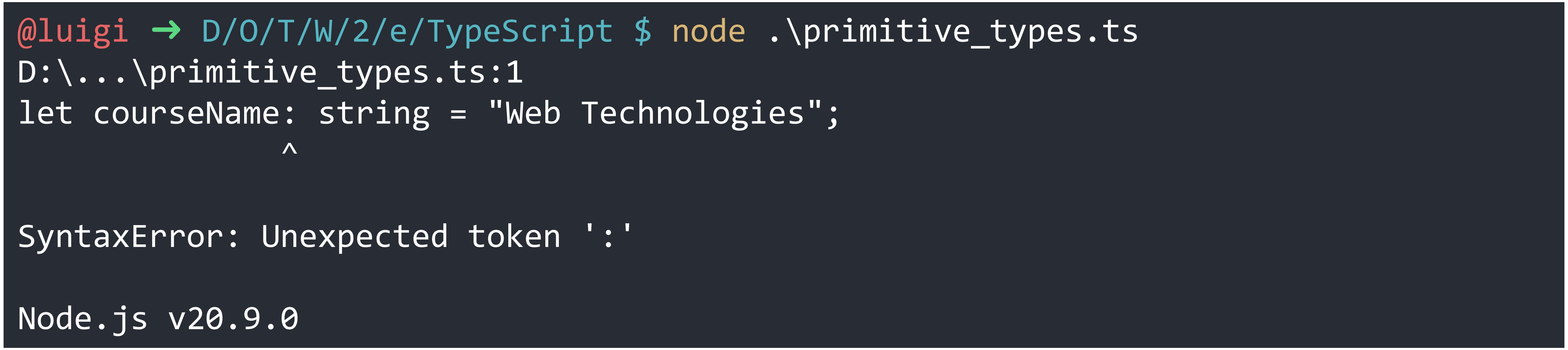
TYPESCRIPT: DECLARING TYPES

primitive_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

console.log(/* Omitted for the sake of brevity */);
```

- The above example is **not** valid JavaScript code anymore!



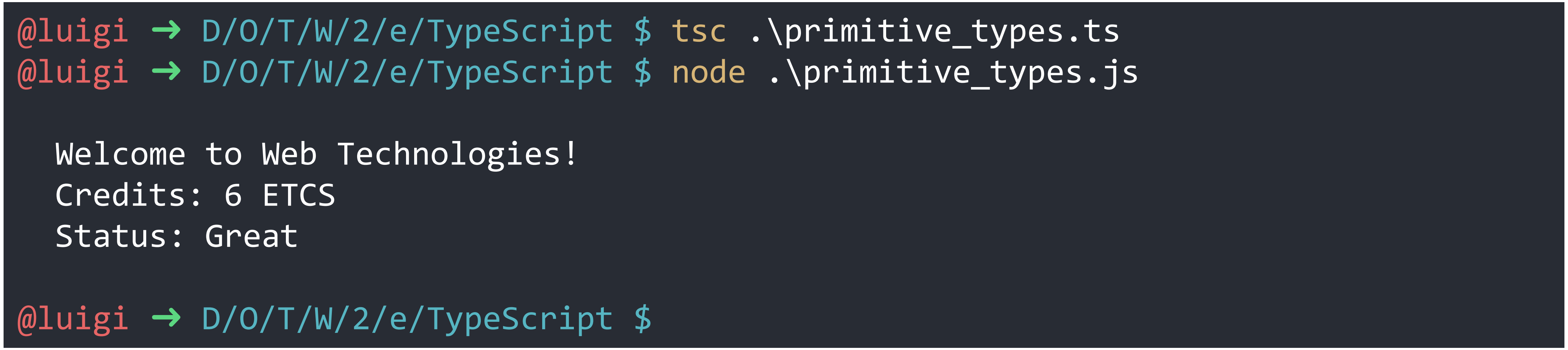
```
@luigi → D/O/T/W/2/e/TypeScript $ node .\primitive_types.ts
D:\...\primitive_types.ts:1
let courseName: string = "Web Technologies";
              ^


SyntaxError: Unexpected token ':'

Node.js v20.9.0
```

TYPESCRIPT: DECLARING TYPES

- To execute our example, we must compile the TS file beforehand



```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\primitive_types.ts
@luigi → D/O/T/W/2/e/TypeScript $ node .\primitive_types.js

  Welcome to Web Technologies!
  Credits: 6 ETCS
  Status: Great

@luigi → D/O/T/W/2/e/TypeScript $
```

TYPESCRIPT: ENJOYING TYPE CHECKING!

primitive_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

credits = "9 CFU"; // This would be fine in plain JavaScript!
```

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\primitive_types.ts
primitive_types.ts:5:1 - error TS2322: Type 'string' is not assignable to type
'number'.

5 credits = "9 CFU";
  ~~~~~~~


Found 1 error in primitive_types.ts:5
```

TYPESCRIPT: ARRAY TYPES

To specify the type of an array, we can use the syntax **type[]**

array.ts

```typescript
let langs: string[] = ["TypeScript", "JavaScript", "PHP", "Java"];
let primes: number[] = [2, 5, 7, 97, 829, 839];

langs.push({name: "C#"}); //would be fine in plain JavaScript
```

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\array.ts
array.ts:4:12 -  error TS2345: Argument of type '{ name: string; }' is not
assignable to parameter of type 'string'.

4 langs.push({name: "C#"}); //would be fine in plain JavaScript
             ~~~~~~~~~~~~~


Found 1 error in array.ts:5
```

TYPESCRIPT: THE ANY TYPE

- TypeScript includes a special type: **any**
- When a value is of type any, we can do anything with it and the compiler won’t complain!
- The code below compiles fine
  - …and throws a **TypeError: x.foo is not a function**

```
let x: any;

x = {name: "Angular", lang: "TypeScript"};
x.foo();
x = "Hello!" ;
let n:number = x;
```

TYPESCRIPT: WORKING WITH FUNCTIONS

- TypeScript allows us to specify the types of both the inputs and output values of functions

```
function greet(name: string): string {
  return `Hello ${name.toUpperCase()}`;
}

console.log(greet(42));
//Err. TS2345: Arg. of type 'number' is not assignable to param. of type 'string'
```

TYPESCRIPT: AUTOMATIC TYPE INFERENCE

- When no explicit type annotation is provided, TypeScript tries to **automatically infer types** from the **context**

```
function greet(name){ //function greet(name: any): string
  return `Hello ${name}`;
}

let arr = [1,2,3,4,5]; //let arr: number[]

let course = "Web Technologies"; //let course: string

greet(course);

course = {name: "Web Technologies", credits: 6};
//TS2322: Type '{name: string; credits: number;}' not assignable to type 'string'
```

TYPESCRIPT: IMPLICIT ANY

- When you don’t specify a type, and TypeScript can’t infer it from context, the compiler will typically default to **any**.
- We might want to avoid this, because **any** isn’t type-checked.
- The compiler flag **noImplicitAny** can be used to flag any implicit any as an error.

implicit_any.ts

```
let x; //let x: any

function greet(name){ //function greet(name: any): void
  console.log(`Hello, ${name}`);
}

greet(x);
```

TYPESCRIPT: IMPLICIT ANY

implicit_any.ts

```
let x; //let x: any

function greet(name){ //function greet(name: any): void
  console.log(`Hello, ${name}`);
}

greet(x);
```

```
@luigi ➜ D/O/T/W/2/e/TypeScript $ tsc .\implicit_any.ts
@luigi ➜ D/O/T/W/2/e/TypeScript $ tsc -noImplicitAny .\implicit_any.ts
implicit_any.ts:3:16 - error TS7006: Param 'name' implicitly has an 'any' type.

3 function greet(name){ //function greet(name: any) : string
                 ~~~~


Found 1 error in implicit_any.ts:3
```

TYPESCRIPT: OBJECT TYPES

To define an **object type**, we list its properties and their type

```
function printCourse(course: {name: string, credits: number}) {
  console.log(`Course name: ${course.name}`);
  console.log(`Course credits: ${course.credits}`);
}

printCourse({name: "Web Technologies", credits: 6}); //works fine

printCourse({name: "Software Engineering"}); //compile error
// TS2453: Property 'credits' is missing in type '{ name: string; }' but required
// in type '{ name: string; credits: number; }’

printCourse({name: "Algebra", credits: 6, year: "First"}); //compile error
// TS2353: Object literal may only specify known properties, and 'year' does not
// exist in type '{ name: string; credits: number; }'
```

TYPESCRIPT: OPTIONAL PROPERTIES

Object types can specify one or more **optional** properties, by adding a «?» after the property name

```
function printCourse(course: {name: string, credits?: number}) {
  console.log(`Course name: ${course.name}`);
  console.log(`Course credits: ${course.credits}`);
  //we should check whether course.credits is undefined!
}

printCourse({name: "Web Technologies", credits: 6}); //works fine
printCourse({name: "Software Engineering"}); //works fine
//prints "Course credits: undefined"!

printCourse({name: "Algebra", credits: 6, year: "First"}); //compile error
// TS2353: Object literal may only specify known properties, and 'year' does not
// exist in type '{ name: string; credits: number; }'
```

TYPESCRIPT: UNION TYPES

- **Union types** are obtained by **combining** two or more types
- Combined types are defined using a list of types separated by «|»
- Represent values that can belong to any of the combined types

```
function printError(code: number | string){
  console.log(`Error code: ${code}`);
}

printError(404);
printError("401 Unauthorized");
printError({code: "403", message: "Forbidden"}); //compile error
// TS2345: Argument of type '{ code: string; message: string; }' is not
// assignable to parameter of type 'string | number'
```

TYPESCRIPT: WORKING WITH UNION TYPES

- TypeScript ensures that operations on a union type are valid on each possible type in the union!

union.ts

```
function printError(code: number | string){
  console.log(`Error code: ${code.toUpperCase()}`); //compile error
}
```

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\union.ts
union.ts:2:35 - error TS2339:
Property 'toUpperCase' does not exist on type 'string | number'.
  Property 'toUpperCase' does not exist on type 'number'.

2   console.log(`Error code: ${code.toUpperCase()}`);
                                    ~~~~~~~~~~~


Found 1 error in union.ts:2
```

TYPESCRIPT: WORKING WITH UNION TYPES

- In such cases, we need to narrow down the type for TypeScript using code and specific checks (e.g.: using `typeof`, or `Array.isArray(x)`)
- TypeScript can infer specific types for particular branches

union.ts

```
function printError(code: number | string){
  if(typeof code === "string"){
    console.log(`Error: ${code.toUpperCase()}`); //in this branch code: string
  } else {
    console.log(`Error: ${code}`); //in this one code: number!
  }
}
```

TYPESCRIPT: TYPE ALIASES

- We’ve been using Object types and Union types by specifying them directly in type annotations
- We might want to use the same types multiple times,
    - As Software Engineers we want to write **DRY** (Don’t Repeat Yourself) code!
- **Type aliases** are a way to assign a specific name to a type
- The syntax for a type alias involves using the **type keyword**

```
type <name> = <definition>
```

TYPESCRIPT: TYPE ALIASES

```typescript
type Credits = number | string;
type Course = {
  name: string,
  credits: Credits
}

function printCourse(course: Course) {
  console.log(`Course name: ${course.name}`);
  console.log(`Course credits: ${course.credits}`);
}

let webtech: Course = {name: "Web Technologies", credits: 6};
let softeng: Course = {name: "Software Engineering", credits: "10 CFU"};

printCourse(webtech); //Name: Web Technologies, Credits: 6
printCourse(softeng); //Name: Software Engineering, Credits: 10 CFU
```

TYPESCRIPT: INTERFACES

- Interface declarations are another way of naming object types

```
interface Course {
  name: string;
  credits: number;
}
function printCourse(course: Course) {
  console.log(`Name: ${course.name}, Credits: ${course.credits}`);
}
let webtech: Course = {name: "Web Technologies", credits: 6};

printCourse(webtech);
printCourse({name: "Software Engineering", credits: 10});
```

TYPESCRIPT: STRUCTURALLY TYPED NATURE

TypeScript is a **structurally-typed** language

- It only cares about **structure** and **capabilities** of types when determining type compatibility, not about names!
- Java, on the contrary, features a **nominative** type system.

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

function printPerson(p: Person) { console.log(`Name: ${p.name}, age: ${p.age}`);}

let person: Person = {name: "Janet", age: 20};
let pet: Pet    = {name: "Chuck", age: 3};

printPerson(person); //Name: Janet, age: 20
printPerson(pet);    //Name: Chuck, age: 3
```

[1] Paquet, J., & Mokhov, S. A. (2010). Comparative studies of programming languages; course lecture notes. Section 3.3. *arXiv preprint:* https://arxiv.org/pdf/1007.2123.pdf

TYPESCRIPT: EXTENDING INTERFACES

```
interface Pet {
  name: string
}

interface Bird extends Pet {
  flies: boolean
}

let chuck: Bird = {name: "Chuck", flies: true};
let quentin: Bird = {flies: false}; //compile error
//TS2322: Type '{ flies: false; }' is not assignable to type 'Bird'.
// Property 'name' is missing in type '{ flies: false; }' but required in type
// 'Pet'.
```

TYPESCRIPT: TYPES VS INTERFACES

Type aliases and **interfaces** are very similar and often interchangeable

- A key distinction is that types cannot be re-opened to add new properties, while interfaces are always extendable

```
interface Pet {
  name: string
}

interface Pet {
  age: number
}

let matt: Pet = {name: "Matt", age: 2};
```

```
type Pet = {
  name: string
}

type Pet = { //TS2300: Dup. identifier
  age: number
}

let matt: Pet = {name: "Matt", age: 2};
```

TYPESCRIPT: EXTENDING TYPE ALIASES

```
type Pet = {
  name: string
}

type Bird = Pet & { //intersection type!
  flies: boolean
}

let chuck: Bird = {name: "Chuck", flies: true};
```

- Similarly to Union types, **Intersection types** combine multiple types into one. Created using the «&» character.
- The resulting type has the properties of each single type

TYPESCRIPT: TYPE ASSERTIONS

- Sometimes, as a dev, you will know better than TypeScript!
- For example, if you use `document.getElementById()`, TypeScript will only be able to infer that the call will return an `HTMLElement`.
- But you might know that the perticular element will be of a more specific type!

```
const nameInput = document.getElementById("name") as HTMLInputElement;
```

- **Beware**: not really an assertion (as in the testing domain)!
  - Not enforced at all at runtime, just at compile-time!

TYPESCRIPT: TYPE LITERALS

In addition to string and number types, we can refer to **specific** strings and numbers in type positions

```
let alignment: "center";

alignment = "center";
alignment = "left"; //TS2322: Type '"left"' is not assignable to type '"center"'
```

- In the example, `alignment` can only have the `center` value
- That’s not very useful…
- Literals can be combined with Unions to express a much more useful concept: types that correspond to a certain set of know values!

TYPESCRIPT: TYPE LITERALS

```
type Alignment = "center" | "left" | "right";

let alignment: Alignment;

alignment = "left";   //ok

alignment = "right";  //ok

alignment = "centre"; //compile error
// TS2820: Type '"centre"' is not assignable to type 'Alignment'.
// Did you mean '"center"'?
```

TYPESCRIPT: ENUMS

Enums allow developers to define a set of named constants

```
interface Order{ isPremium: boolean }

enum Priority {
  Low,
  Medium,
  High
}

function computePriority(s: Order): Priority {
  if(s.isPremium)
    return Priority.High;
  return Priority.Low;
}

console.log(Priority.Medium); //1
```

TYPESCRIPT: FUNCTION TYPES

- What if we want to specify that a function takes as input a callback which expects specific parameters and produces a given output?
- Functions can be described using **function type expressions**

```
type stringDecoratorFunction = (x: string, mode: boolean) => string;
```

- The function type above indicates a function that takes as inputs a **string** and a **boolean** argument, and returns a **string**

```
let fn: () => string; //fn is a function that takes no args and returns a string

fn = () => {return "Hello Web Tech"};     //ok
fn = (x: string) => {return `Hello ${x}`} //TypeError!
```

TYPESCRIPT: FUNCTION TYPES

```typescript
enum Case { Uppercase, Lowercase }

function GREET(name: string, decorator: (x: string, mode: Case) => string){
  return decorator(name, Case.Uppercase);
}

function stringDecorator(x: string, mode: Case) {
  if(mode === Case.Lowercase)
    return `>>> Hello ${x} <<<`.toLowerCase();
  else
    return `>>> Hello ${x} <<<`.toUpperCase();
}

console.log(GREET("Web Technologies", stringDecorator));
```

GENERICS

- A good deal of efforts in Software Engineering goes towards building **reusable** software
- Developing components that can seamlessly operate on both current and future data is crucial for building up large software systems
- **Generics** is one of the main tools to write code that can seamlessly work with a variety of types

HELLO GENERICS: THE IDENTITY FUNCTION

Suppose we need to implement the **identity** function, i.e., a function that returns back whatever is passed in.

```
function identity(x){ //function identity(x: any): any
  return x;
}

let y: string = "Hello";

let z = identity(y); //z: any!
```

- Using **any** (implicitly or explicitly) is certainly generic enough
- But we're loosing information about what the type was when the function returns!

HELLO GENERICS: THE IDENTITY FUNCTION

```
function identity(x: string): string {
  return x;
}

let y: string = "Hello";

let z = identity(y); //z: string (but only works with string!)
```

We could give the identity function a specific type

- It would only work with **`string`**s though…

TYPESCRIPT: GENERICS

TypeScript allows us to define **type variables**, a special kind of variables that works on types rather than values

- The type variable is declared after the function name, between «<» and «>».
- Below, the variable is called **Type** (but it could any valid identifier as a name).

```
function identity<Type>(x: Type): Type {
  return x;
}

let s: string = identity<string>("Hello"); //explicitly set Type to string
let n: number = identity(42); //let automatic type inference do its magic
let o: {title: string, artist: string} = identity({
  title: "Sultans of swing",
  artist: "Dire Straits"
});
```

TYPESCRIPT: GENERICS

```
function identity<Type>(x: Type): Type { return x; }

let s: string = identity<string>("Hello"); //explicitly set Type to string
let n: number = identity(42); //let automatic type inference do its magic
let o: {title: string, artist: string} = identity({
  title: "Sultans of swing", artist: "Dire Straits"
});
```

- This is **not** the same as using the **any** type
- We’re preserving the information on the input type!

TYPESCRIPT: GENERICS

- There can be multiple type variables as well

```
function merge<T1, T2>(x: T1, y: T2): {field: T2, otherField: T1} {
  return {field: y, otherField: x};
}

let x = merge("Sultans of Swing", 42);
//x has type {field: number, otherField: string}
```

- And we can also work with type-parametric arrays

```
function getFirst<T>(arr: T[]): T {
  return arr[0];
}

let first: string = getFirst(["hello", "web", "technologies"]);
console.log(first); //hello
```

TYPESCRIPT: GENERIC CLASSES

```
class CustomCollection<T> {
  list: T[] = [];

  addElement(element: T): number {
    return this.list.push(element);
  };

  getRandomElement(): T{
    let selectedIndex: number = Math.floor(Math.random()*this.list.length);
    return this.list[selectedIndex];
  }
}

let c = new CustomCollection<string>();
c.addElement("hello"); c.addElement("web"); c.addElement("technologies");
console.log(c.getRandomElement());
```

TYPESCRIPT: GENERIC TYPE CONSTRAINTS

```
class Animal { species: string; }
class Bird extends Animal { canFly: boolean; }
class Snake extends Animal { isVenomous: boolean; }

function animalInfo<T extends Animal>(animal: T): void {
  console.log(animal.species);
}

let a: Animal = new Animal(); a.species = "Dog";
let b: Bird = new Bird(); b.species = "Kiwi"; b.canFly = false;
let c: Snake = new Snake(); c.species = "Cobra"; c.isVenomous = true;

animalInfo(a); //Dog
animalInfo(b); //Kiwi
animalInfo(c); //Cobra
animalInfo({species: "Spider"}); //Spider (!!!)
```

TYPESCRIPT: OTHER TYPES TO KNOW ABOUT

TypeScript includes also some additional types, that might be useful especially in the context of functions, and that we should know about:

- `void`
- `unknown`
- `never`

TYPESCRIPT: VOID

- **void** is the return type of functions that do not return a value
- It is the inferred return value for functions that do not have a return statement, or have an empty return statement

```
function f(): void {
  return;
}

function g(){ // function g(): void
  console.log("Hello");
}

let x: void = g();
```

TYPESCRIPT: UNKNOWN

- **unknown** represents any possible value.
- It is similar to the **any** type, but safer (does not allow any operation!)

```
function greet(a: any){
  console.log(a.name); //OK, but possible runtime error
}

function saferGreet(a: unknown){
  console.log(a.name); //Compile error TS18046: 'a' is of type unknown
}

type namedObject = {name: string};

function notSoSafeGreet(a: unknown){
  console.log((a as namedObject).name);
}
```

TYPESCRIPT: THE NEVER TYPE

- Some functions never return a value
- The **never** type represents values which are *never* observed

```
function f(x: string | number){
  if(typeof x === "string"){
    return x.toLowerCase();
  } else if (typeof x === "number") {
    return x.toPrecision(2);
  } else {
    return x; //x has type never in this branch
  }
}

function g(x: any): never { //g never returns (it always throws an error)
  throw new Error("Oops!");
}
```

TYPESCRIPT: STRICTNESS LEVELS

- Different users expect different things from TypeScript
- By default, TypeScript offers an opt-in experience
  - Types are optional, **any** is used when a precise type cannot be inferred
  - We can be more strict about inferred **any**s by using the **noImplicitAny** flag
- To avoid being too intrusive, some checks are disabled by default
  - For example, **null** and **undefined** can be assigned to any type
  - Forgetting to explicitly handle null/undefined values is the cause of countless bugs in the world (some called it the billion dollar mistake!)
  - The **strictNullChecks** flag can be used to ensure that null and undefined values are explicitly handled

TYPESCRIPT: STRICT NULL CHECKS

strict.ts

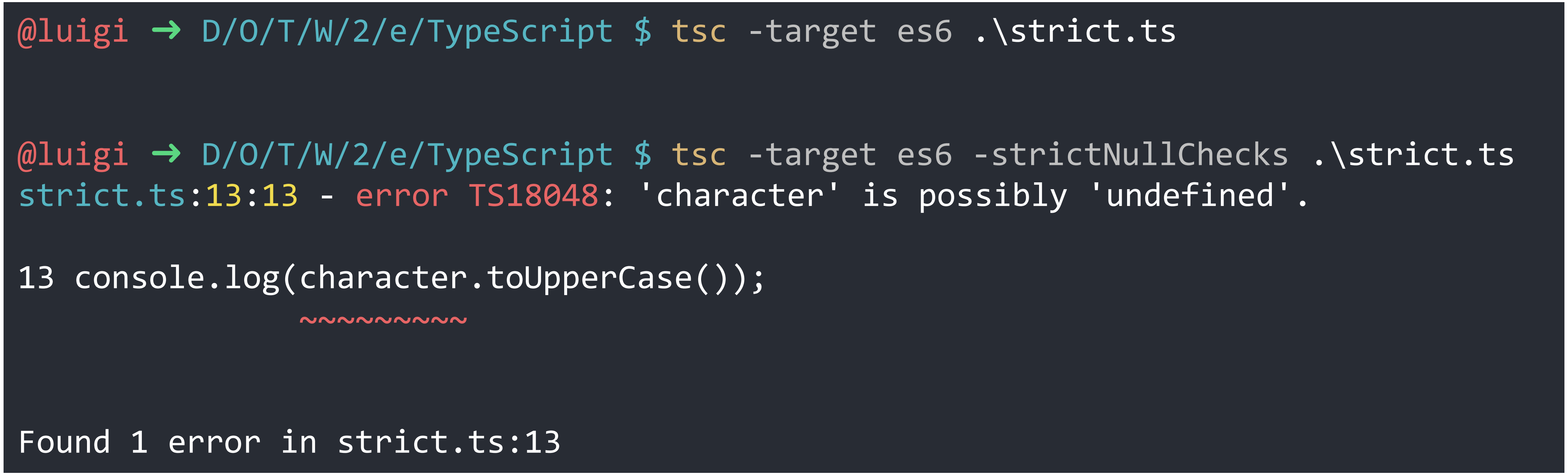
```
class CustomCollection<T> {
  list: T[] = [];
  add: (x: T) => number = (element: T) => {
    return this.list.push(element);
  }
}

let x = new CustomCollection<string>();
x.add("Sam"); x.add("Cliff"); x.add("Mama");

let character = x.list.find((val: string) => {
  return val === "Amelie"
});

console.log(character.toUpperCase()); //character might be undefined!
```

TYPESCRIPT: STRICT NULL CHECKS



```
@luigi ➜ D/O/T/W/2/e/TypeScript $ tsc -target es6 .\strict.ts


@luigi ➜ D/O/T/W/2/e/TypeScript $ tsc -target es6 -strictNullChecks .\strict.ts
strict.ts:13:13 - error TS18048: 'character' is possibly 'undefined'.

13 console.log(character.toUpperCase());
               ~~~~~~~~~


Found 1 error in strict.ts:13
```

REFERENCES (1/2)

- **The TypeScript Handbook**
  Available at https://www.typescriptlang.org/docs/handbook/intro.html
  ⚠️ You do not need to know the *entire* handbook! Use it as a reference for the parts that were discussed in these slides.
- **Lecture Notes for the Comparative Studies of Programming Languages Course (Revision 1.9)**
  By Paquet, J., & Mokhov, S. A. (2010)
  Available at https://arxiv.org/pdf/1007.2123.pdf
  ⚠️ If you want to learn more about type system design in programming languages (e.g.: duck typing, structural vs nominative type systems), check out Section 3.3. This is **not** required for the Web Technologies course!

REFERENCES (2/2)

- **To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub**
  By Justus Bogner and Manuel Merkel (2022)
  Available at https://dl.acm.org/doi/pdf/10.1145/3524842.3528454
  ℹ️ Interesting read. The authors compare JavaScript and TypeScript applications on multiple grounds, ranging from code quality and readability to bug proneness and bug resolution times.