Introduction to TypeScript

Session-1

CLARUSWAY
WAY TO REINVENT YOURSELF

Did you finish pre-class material?

Students choose an option

Session Topics

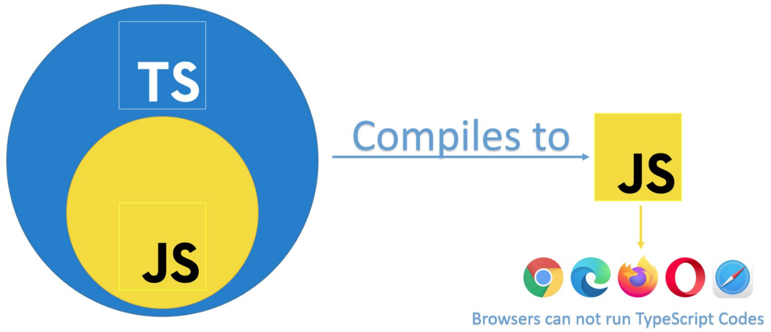
- What is the TypeScript?
- Setup Development Environment
- Type Annotation
- Types in TypeScript
- Type Assertions
- Functions

CLARUSWAY
WAY TO REINVENT YOURSELF

What is the TypeScript?

What is the TypeScript?

- Anders Hejlsberg began working on typescript at Microsoft in 2010, and the first version of typescript was given to the public in 2012. (ts 0.8).
- It is a superset of JavaScript. That is, it's JavaScript with a bunch of additional features.
- TS codes are not be run directly like. TS codes are compiled into JavaScript. Your valid JS code is also valid TS code.



CLARUSWAY
WAY TO REINVENT YOURSELF

What is the TypeScript?

- **Cross-platform:** TS runs on any platform that JavaScript runs on. The TS compiler can be installed on any operating system
- **Object oriented language:** Powerful features such as classes, interfaces, and modules.
- **Static type-checking:** TS uses static typing, done using type annotations. Type checking is done at compile time.
- **Optional static typing:** Allows optional static typing if you would rather use JavaScript's dynamic typing.
- **DOM manipulation:** just like JavaScript
- **ES 6 features:** TS includes features of ES6

TypeScript Pros

- Strict typing
- Structural typing
- Type annotations
- Type inference

TypeScript Cons

- Not true static typing (this feature is optional for TS)
- One more JavaScript to learn
- Adding extra step — transpiling
- Bloated code (more lines of coding)

Setup Development Environment

Setup Development Environment

- Install typescript using node.Js package manager (npm).
- Install the typescript plug-in in your IDE.

```
> npm install -g typescript
>tsc -v
 Version 4.x.x
```

tsconfig.json

- The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project.
- The tsconfig.json file specifies the root files and the compiler options required to compile the project.
- tsc --init command creates a configuration file called tsconfig.json
- tsc command will generate .js files for all .ts files.

tsconfig.json

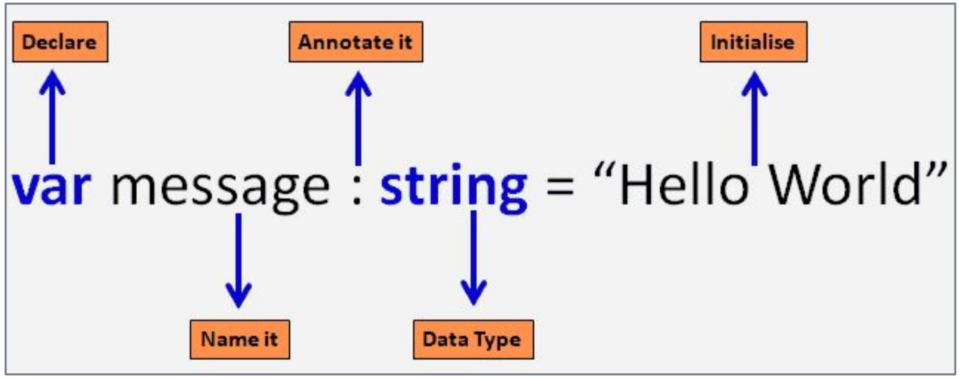
allowJs	Allow JavaScript files to be compiled. Default value is false.
alwaysStrict	Parse in strict mode and emit "use strict" for each source file. Default value is false.
target	Specify ECMAScript target version.
outDir	The location in which the transpiled files should be kept.
noEmitOnError	Disable emitting files if any type checking errors are reported.
noUnusedParameters	Raise an error when a function parameter isn't read.
removeComments	Disable emitting comments.
noImplicitAny	Enable error reporting for expressions and declarations with an implied 'any' type.
strictNullChecks	When type checking, take into account 'null' and 'undefined'.

Type Annotation

CLARUSWAY
WAY TO REINVENT YOURSELF

Type Annotation

- TypeScript is a typed language. We need to specify the type of variables, function parameters, and object properties.
- Type annotation is not mandatory. Compiler will check the types of variable and avoid errors when dealing with the data types.
- We annotate a variable by using a colon (:) followed by its type.



Types in TypeScript

CLARUSWAY
WAY TO REINVENT YOURSELF

Data Type - Basic Types

- Boolean: The most basic data type is the simple true/false value, which JavaScript and TypeScript call a boolean value.

```
let isDone: boolean = false;
```

- Number: All numbers in TypeScript are either floating point values or BigIntegers.

```
let decimal: number = 6;
```

- String: The type string to refer to textual data types.

```
let color: string = "blue";
```

Data Type - Arrays

- Array is a collection of values. Array types can be written in one of two ways.

- In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

- The second way uses a generic array type, Array<elemType>:

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

Data Type - Tuples

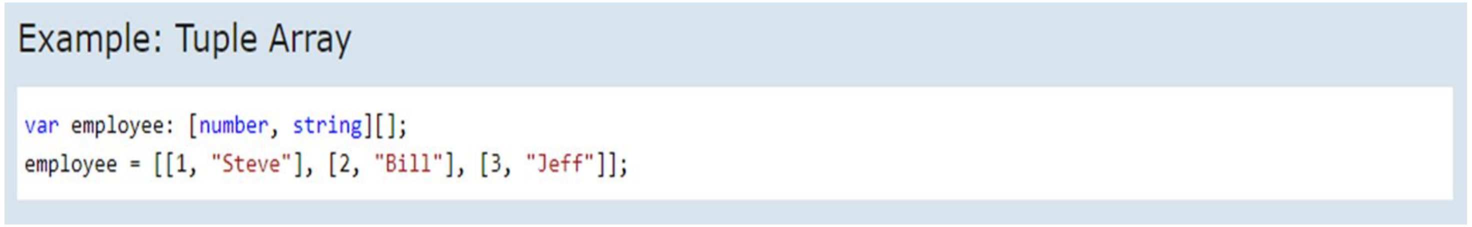
- Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same.

- For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];

// Initialize it
x = ["hello", 10]; // OK

// Initialize it incorrectly
x = [10, "hello"]; // Error
```

Data Type - Tuples

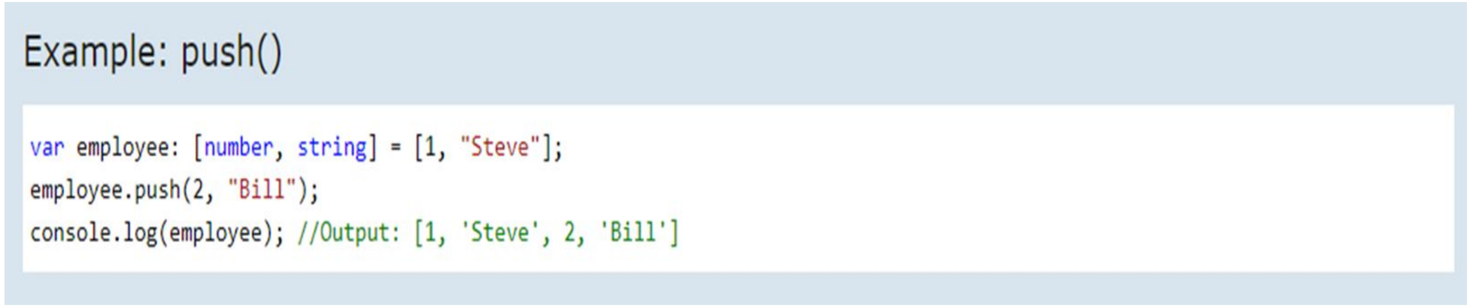
- You can declare an array of tuple also.



Example: Tuple Array

```
var employee: [number, string][];
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

- Add Elements into Tuple



Example: push()

```
var employee: [number, string] = [1, "Steve"];
employee.push(2, "Bill");
console.log(employee); //Output: [1, 'Steve', 2, 'Bill']
```

Data Type - Enum

- A helpful addition to the standard set of data types from JavaScript is the enum. An enum is a way of giving more friendly names to sets of numeric values.

- By default, enums begin numbering their members starting at 0. You can change this by manually setting the value that may contain both string and numeric values.

```
enum Color {
 Red,
 Green,
 Blue,
}
let selectedColor : Color = Color.Green;
console.log(selectedColor) // output: 1
```

Data Type - any

- We can not always have prior knowledge about the type of some variables, especially when there are user-entered values from third party libraries.
- In such cases, we need a provision that can deal with dynamic content.
- To do so, we label these values with the any type:

```
let looselyTyped: any = 4;

let arr: any[] = ['John', 212, true];
```

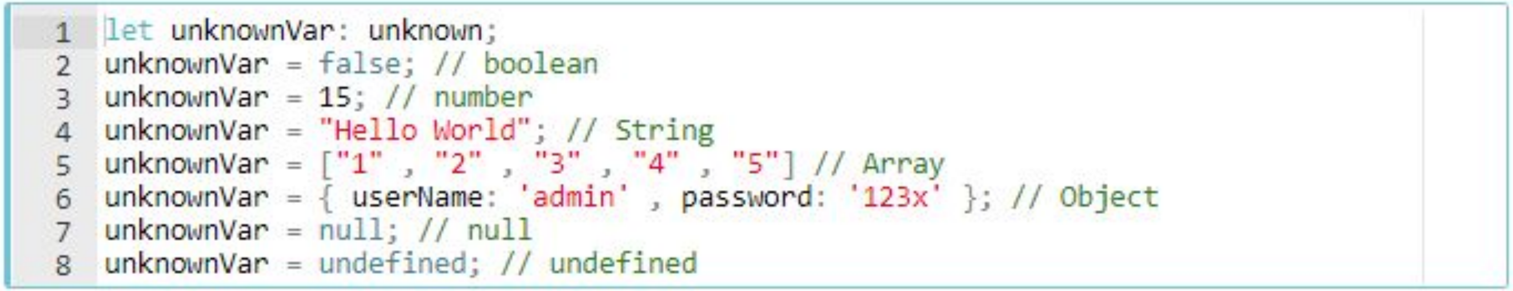
Data Type - unknown

- We may need to describe the type of variables that we do not know when we are writing an application.
- These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API.
- In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

```
let notSure: unknown = 4;

notSure = "maybe a string instead";
```

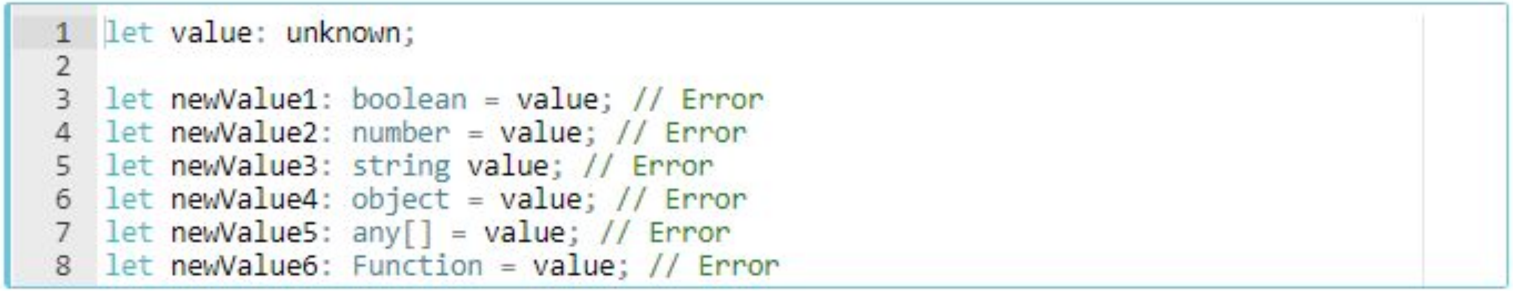
Data Type - unknown

You can assign anything to an unknown type:



```
let unknownVar: unknown;
unknownVar = false; // boolean
unknownVar = 15; // number
unknownVar = "Hello World"; // String
unknownVar = ["1" , "2" , "3" , "4" , "5"] // Array
unknownVar = { userName: 'admin' , password: '123x' }; // Object
unknownVar = null; // null
unknownVar = undefined; // undefined
```

But cannot assign unknown to any other types:



```
let value: unknown;

let newValue1: boolean = value; // Error
let newValue2: number = value; // Error
let newValue3: string value; // Error
let newValue4: object = value; // Error
let newValue5: any[] = value; // Error
let newValue6: Function = value; // Error
```

CLARUSWAY
WAY TO REINVENT YOURSELF

Data Type - void

- void is a little like the opposite of any: the absence of having any type at all.

- You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {

 console.log("This is my warning message");

}
```

CLARUSWAY
WAY TO REINVENT YOURSELF

Data Type - never

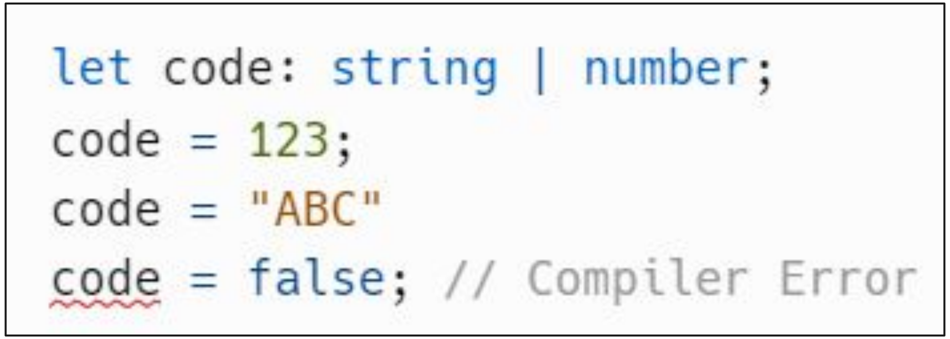
- Typescript introduced a new type never, which indicates the values that will never occur.

- The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

```
// Function returning never must not have a reachable end point

function error(message: string): never {

 throw new Error(message);

}
```

Data Type - union

- Typescript allows us to use more than one data type for a variable or a function parameter. This is called union type



```
let code: string | number;
code = 123;
code = "ABC"
code = false; // Compiler Error
```

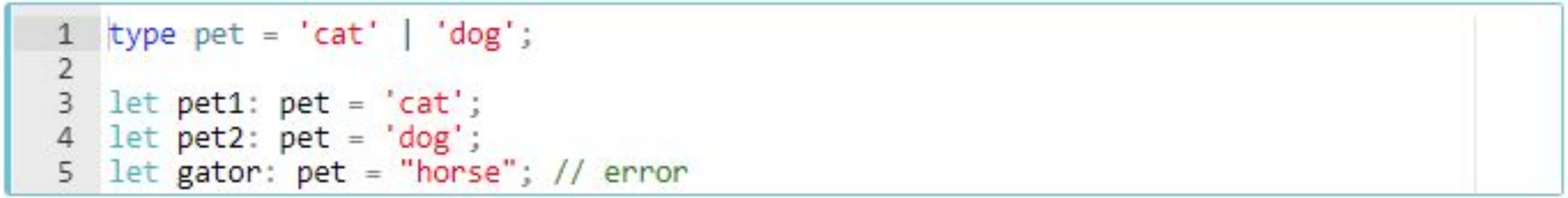
Type Aliases

- It’s important to create dynamic and reusable code. Using TypeScript aliases will help you to accomplish DRY principle.
- We have to create our type before we start using it. We define the type alias with the type keyword.

```
type Point = {
 x: number;
 y: number;
}; // Point is a type now and we can use it

function printCoord(pt: Point) {
 console.log("The coordinate's x value is "+ pt.x);
 console.log("The coordinate's y value is "+ pt.y);
}

printCoord({ x: 100, y: 100 });
```

String Literals

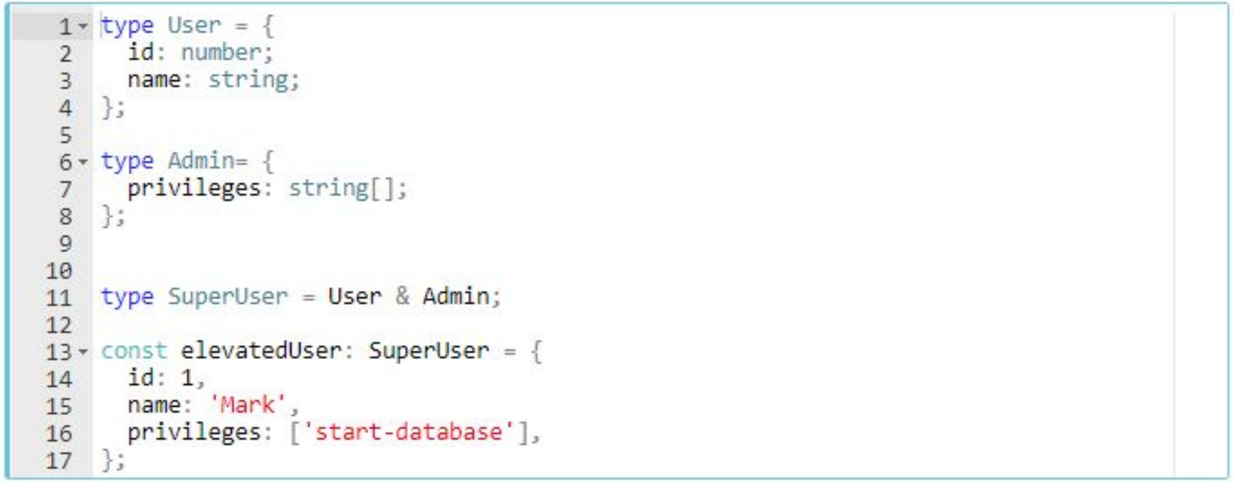
- String literals allow us to use a string as a type.
- String literals become even more powerful when used with union types. Union types are used to define values that can be of more than one type. With union types, the | character is used to separate the different possible types:



```
type pet = 'cat' | 'dog';

let pet1: pet = 'cat';
let pet2: pet = 'dog';
let gator: pet = "horse"; // error
```

Intersection

- Although intersection and union types are similar, they are employed in completely different ways.
- An intersection type is a type that merges several kinds into one. This allows you to combine many types to create a single type with all of the properties that you require.
- An object of this type will have members from all of the types given. The ‘&’ operator is used to create the intersection type.

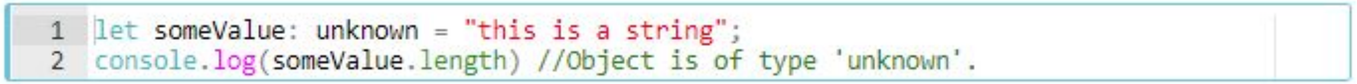


```
type User = {
  id: number;
  name: string;
};

type Admin= {
  privileges: string[];
};


type SuperUser = User & Admin;

const elevatedUser: SuperUser = {
  id: 1,
  name: 'Mark',
  privileges: ['start-database'],
};
```

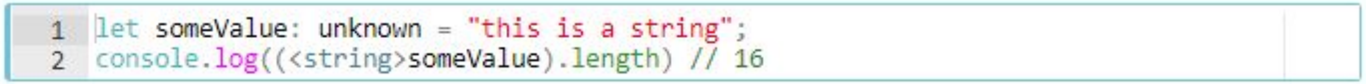
Type Assertions

Type Assertions

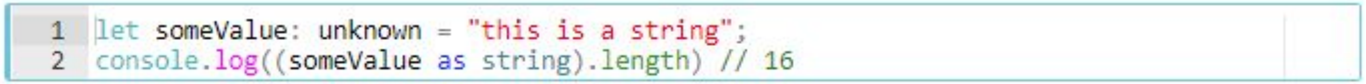
- Type assertion is a technique that informs the compiler about the type of a variable.
- Type assertion is similar to typecasting but it doesn’t reconstruct code.
- You can use type assertion to specify a value type and tell the compiler not to deduce it.
- When we want to change a variable from one type to another such as any to number etc, we use Type assertion.



```
let someValue: unknown = "this is a string";
console.log(someValue.length) //Object is of type 'unknown'.
```



```
let someValue: unknown = "this is a string";
console.log((<string>someValue).length) // 16
```



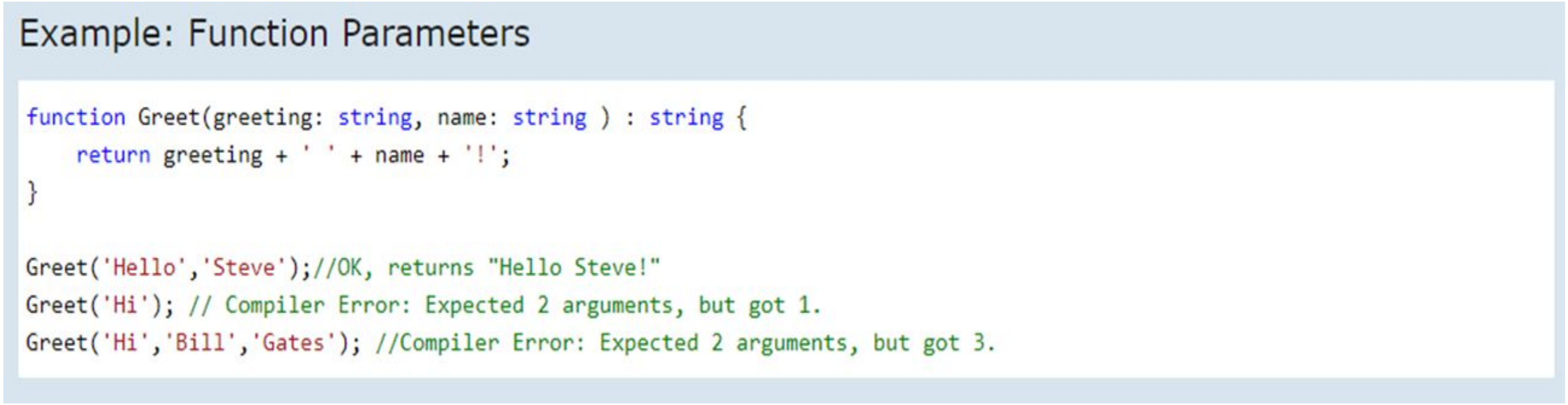
```
let someValue: unknown = "this is a string";
console.log((someValue as string).length) // 16
```

Functions

CLARUSWAY
WAY TO REINVENT YOURSELF

Functions

- Functions should return a type, and also function parameters should have types.
- The compiler expects a function to receive the exact number and type of arguments as defined in the function signature. For example, if a function expects 3 parameters, the compiler checks whether 3 parameters exists with exact type or not.

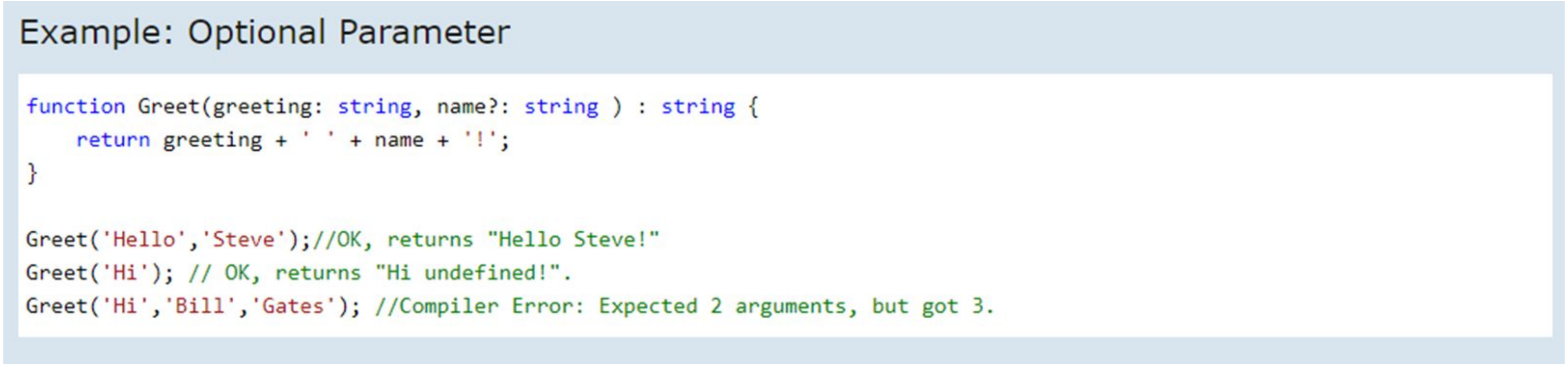


Example: Function Parameters

```
function Greet(greeting: string, name: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

Functions

- In some cases, some of function parameters may be optional.
- Use the ? symbol after the parameter name to make a function argument optional.
- All optional parameters must follow required parameters and should be at the end.

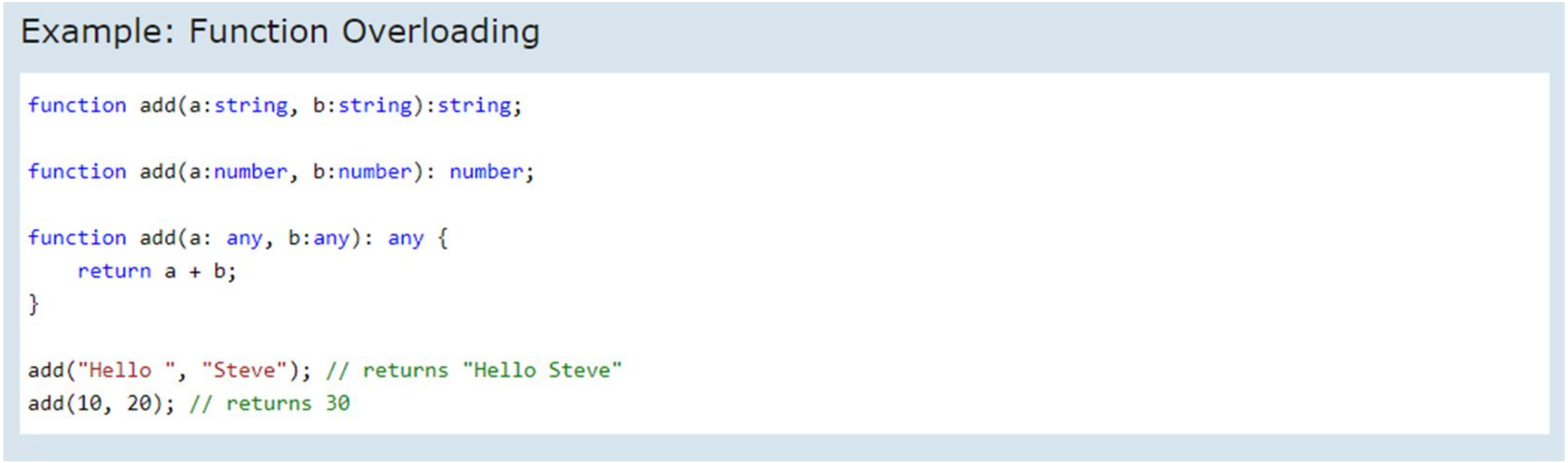


Example: Optional Parameter

```
function Greet(greeting: string, name?: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // OK, returns "Hi undefined!".
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

Functions - Overloading

- Typescript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.



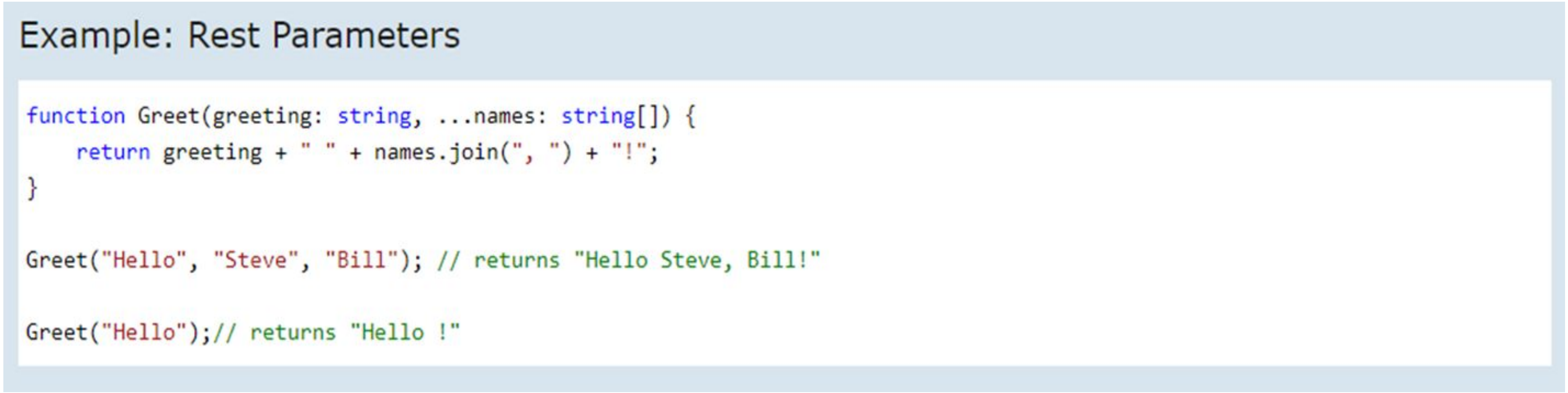
Example: Function Overloading

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {
    return a + b;
}

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

Functions - Rest Parameters

- Typescript has rest parameters to accommodate n number of parameters easily.

- When the number of parameters is not known or can vary, we can use rest parameters. In JavaScript, this is achieved with the "arguments" variable. In typescript, we can use the rest parameter denoted by ellipsis. Rest parameters must come last in the function definition, otherwise the Typescript compiler will give an error.



Example: Rest Parameters

```
function Greet(greeting: string, ...names: string[]) {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```

THANKS!