Typescript

Dr. Jose L. Muñoz Tapia

Universitat Politècnica de Catalunya (UPC)

Typescript

Outline

Motivation & Concept

- Some problems of javascript appear at runtime (bad thing):
  - Functions might be called without the proper arguments.
  - Objects can change their properties.
  - Variables might change their type.
- Typescript is:
  - A superset of JS (JS is valid TS).
  - TS includes:
    - Features of new JavaScript versions (ES6, ES7 and so on).
    - Strong typing.
    - Object-oriented features: classes, constructors, etc.
  - Browsers do not support TS (never will), so:
    - TS needs to be transcompiled to JS.
    - Transcompilation is done during development.
    - We can catch many errors at compile-time.
  - Has great tooling (intellisense in code editors).

Installation

- To install:

```
$ npm install -g typescript
$ tsc --version
Version 3.7.5
```

- We create a project:

```
$ mkdir ts-hello
$ cd ts-hello
ts-hello$ code main.ts
```

- We create the file main.ts:

```
function log(msg){ console.log(msg); }
var message = "hi";
log(message);
```

- We compile it with:

```
$ tsc main.ts
```

- We will observe that we have a new file main.js (with the same content).

Using let

- Typescript supports `let`, which defines a variable with a block scope:

```
function doSomething(){
    for (let i=0; i< 5; i++){
        console.log(i);
    }
    console.log("Last value: " + i);
}

doSomething();
```

- When we compile:

```
$ tsc let.ts
let.ts(5,34): error TS2304: Cannot find name 'i'.
```

- We get a compilation error (good!).
- But still `tsc` gives a result (translating to js).
- Here, `tsc` replaces let with var.
- Note: `let` has been added too to ES6 (2015).

Double Question Marks (Nullish Coalescing)

- This is a functionality added in TS 3.7 and also in vanilla Javascript:

```
const mayBeThisThingOrThisOtherThingIfNot = thisThing || thisOtherThing

// Translates to:
if (thisThing != 0 &&
    thisThing != undefined &&
    thisThing != false &&
    thisThing != '' &&
    thisThing != null &&
    thisThing != NaN &&
    thisThing != 0n) {
    const isThisThing = thisThing;
}
else { const isThisOtherThing = thisOtherThing; }
```

- While:

```
const mayBeThisThingOrThisOtherThingIfNot = thisThing ?? thisOtherThing

// Could be written as
if (thisThing != null &&
    thisThing != undefined) {
    const isThisThing = thisThing;
} else { const isThisOtherThing = thisOtherThing; }
```

Outline

Basic Typing

- We can use **implicit** typing:

```
let count = 5;
count = 'a';  // error
let myVar;    // any type
```

- We can also use **explicit** typing:

```
let b: number;
let c: boolean;
let d: string;
let e: any;
let f: number[] = [1,2,3];
let g: any[] = [1,null,"hi"];
```

- Another syntax for typing arrays:

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

- You don't typically type `null` and `undefined` because these types only have one value.
- With types, we can use intellisense to autocomplete:

```
let message = 'abc';
message.endsWith('c');
```

Transcompilation

```
// myapp.ts
var a: number;
var b: string;
a=10;
b="hello";
```

```
$ tsc myapp.ts
```

The compiler erases the types when it compiles:

```
var a;
var b;
a = 10;
b = "hello";
```

Transcompilation Errors

- If we make an error:

```
var a: number;
var b: string;
a=true;
b="hello";
```

- The philosophy of the compiler when there are errors is to complain but it compiles anyway the code (traditional compilers stop when errors).
- TS compiler exists to alert developers about possible errors but final decisions are taken by developers.

Enums

Javascript does not have enums.

With Typescript we can define **enums**:

```
// Examples of enums:

enum Color {Red,Green,Blue};
let backgroundColor = Color.Blue; // equals 2

enum otherColor {Red = 2,Green =7 ,Blue=9};
console.log(otherColor.Red);     // prints 2

enum moreOtherColor {Red = 2, Green ,Blue};
console.log(moreOtherColor.Blue); // prints 4
```

Enums with Strings

- If you want to save the results in a DB or show to a user, data from an enum, numbers have not clear in meaning.
- To fix this, we can use strings instead of numbers in enums:

```
enum Shape {
  Square = "square",
  Rectangle = "rectangle",
  Triangle = "triangle",
  Circle = "circle"
}
console.log (Shape.Circle) // result: circle
```

- A simple application is to use enums to catch misspelling errors in comparisons:

```
if (a === 'rectungle'){
  // ...
}
```

```
if (a === Shape.Rectungle){  // fail at compile time
  // ...
}
```

Tuples

- Arrays are structures of elements of one type:

```
let myArr: number[];
```

- What if we want to create an array with multiple data types:

```
let myArr = [1,'hi',true];
```

- This is technically called a tuple:

```
let myArr: [number,boolean];
myArr = [1,true];       //ok
myArr = [100, false];  //ok
myArr = [false, 3];    //ko
```

- Notice that:
    - For arrays you specify the data type before [].
    - For tuples data types are specified like elements inside [].
- Other than that (and the low level implementation), the way tuples work and their syntax is like an array.

Outline

Typing Function Arguments

- Typing arguments and return values of functions is very useful:

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

- If we call it with `add('hi',2)` JS converts 2 to string and concatenates.
- If we want to force that the function is only called with numbers we can type it:

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

- Return value:
  - In the previous function, the type of the returned value is `: number`.
  - We can use `: void` if there is no return value.
  - You can also remove the return type from the function declaration (using the implicity type of what is returned).

Optional Arguments

- By default, typescript enforces that the arguments exactly match the function signature.
- What if we want a function that might receive different sets of arguments? E.g. add two or three numbers:

```
function add (a:number, b: number, c?: number) : number {
    return a+b;
}
```

- Optional arguments:
  - ? means optional.
  - Optional args cannot be followed by required arguments.
  - Typescript also has syntax for default values for optional values:

```
function add (a:number, b: number, c? = 0: number) : number {
    return a+b+c;
}

```

  - Note. With default values we can get rid of ?

Typing Arguments of Arrow Functions

- We can use arrow functions with TS:

```
let doLog = (message) => console.log(message);
let doLog = () => console.log('hi');
```

- We can also type them:

```
let square = (x: number): number => {
  console.log("squaring");
  return x * x;
}
```

Return Implicit Typing

- Recall that we can use implicit typing:

```
let a = 'hello';
a = true; // we get an error
```

- This can be also used with return values of functions:

```
function greet() : string {
    return 'good morning';
}

let greeting = greet(); // also makes an implicit type assignment
greeting = true; // we get an error
```

- In VScode you can hover with the mouse to see the implicit assignations.

Any Type

- When making a declaration without assignment, the type that you get is **any**.
- So, implicit typing **DOES NOT WORK** if the declaration and assignment are in different lines.

```
let myvar; // the type of myvar is any
```

- We can check this explicitly:

```
let myvar: any;
myvar = 3;
myvar = true;
```

- The type **any** means that you don't do type checking (**any** gives you the flexibility of changing the type as with JS).
- The **any** type is useful when migrating JS code into TS.

Function Variables of Type **any**

```
function alwaysTrue(arg:string): boolean { return true; }

let anotherAlwaysTrue;           // anotherAlwaysTruetype is any
anotherAlwaysTrue = alwaysTrue;  // wrong don't assign functions like this
let myInt:number = 5;
myInt = anotherAlwaysTrue("hi"); // this works and compiler does not complain
console.log(`${typeof myInt}`)   // myInt type is boolean !!!
```

You should assign variables when defined to get the typing:

```
let yetAnotherAlwaysTrue = alwaysTrue;  // should assign like this!
```

We can define our variable to contain as type the type of a function:

```
let myEcho: (arg: string) => string;

myEcho = arg => arg;

console.log(myEcho);

myEcho = arg => 7; // compilation error
```

If we try to change the type, as expected the compiler produces an error.

Typing Functions with Interfaces

We can define function types using an interface:

```
// let myEcho: (arg: string) => string;

interface EchoFn {
  (arg: string): string
}

let myEcho: EchoFn;

myEcho = arg => arg;

console.log(myEcho);

myEcho = arg => 7; // compilation error
```

Possible Types and Ending Exclamation

- Consider the following example in which we define a function that can receive and return values of different types:

```typescript
// The function can recieve an argument x of type string or undefined and return the same types
function myfunc(x: string | undefined): string | undefined {
  return x;
}

let b: string;
b = myfunc('hello');
```

- If we strictly compile the file:

```
$ tsc --strict index.ts
index.ts:6:1 - error TS2322: Type 'string | undefined' is not assignable to type 'string'.
  Type 'undefined' is not assignable to type 'string'.

6 b = myfunc('hello');
```

- In this case, we can use an exclamation at the end of our variable to express that we know that the variable is **not going to be `null`** or **`undefined`**.

```typescript
b! = myfunc('hello'); // With ! at the end of the variable we say I am sure it will not be null or undefined
```

Outline

Annotated Objects

- Consider the following function:

```
let draw = (x,y) => {
    //...
};
```

- We might have so many parameters so better pass an object.

```
let draw = (point) => {
    //...
};

draw({ x : 3, y : 7 });
```

- Problem, the object might be incorrect.
- A solution is to annotate the object, we can use inline notation:

```
let draw = (point: {x: number, y: number}) => {
    // ...
};
```

Interfaces

- A more reusable solution is to use an interface:

```
interface Point {
    x: number,
    y: number
}

let draw = (point: Point) => {
    // ...
};
```

- By convention, interfaces use Pascal naming (capital first letter).
- Interfaces only have declarations, not implementations.
- We can also include signatures of methods in interfaces.

```
interface Point {
    x: number,
    y: number,
    draw: () => void }
```

- Notice that we do not need to pass the coordinates to `draw()` (in an object, methods have access to properties).

Catching Errors with Types

Errors at compile-time

The most important feature of Typescript is that it allows us to catch errors thanks to types at compile time.

- With Javascript, the following code will produce an error **at run time**:

```
function sayName(o) { console.log(o.name) }
const bottle = { litres : 1 };
sayName(bottle); // undefined
```

- In Typescript, we can fix this:

```
interface Named { name: string; }
function sayName(o:Named) { console.log(o.name) }
sayName(bottle);
```

- Now, **we get the error when we compile the previous code**.
- The compiler knows that the input object should have `.name` property.

Classes

- From OOP there is a concept called cohesion.
- This means all related stuff must be kept together.
- A class groups functions and properties that are highly related:

```
class Point {
    x: number;                  // field
    y: number;                  // field
    draw () {                   // method
        // implementation...
    }
    getDistance(another: Point){  // method
        // implementation...
    }
}
```

- Now we declare a variable as being a Point object:

```
let point: Point;
```

- Let's implement draw:

```
 draw () {   //method
      console.log('x:' + this.x + 'y:' + this.y);
  }
```

Creating Instances with new

- If we call point.draw we get an error...

```
let point: Point;
point.draw();   // error
```

- We need to create an instance of the object (allocate memory):

```
let point: Point = new Point();

// equivalent and shorter than previous statement
let point = new Point();
```

- Finally, we can set fields to make it work:

```
point.x = 3;
point.y = 9;
```

Constructors

- To create an instance, it is better to use a `constructor`:

```
class Point {
    x: number;
    y: number;
    constructor (x: number,y: number){
        this.x = x;
        this.y = y; }
    draw () { console.log(`x: ${this.x} y: ${this.y}`); }
}
let point = new Point(3,9);
point.draw();
```

- Using **?** makes a parameter optional:

```
class Point {
    x: number;
    y: number = 7;
    constructor (x: number,y?: number){
        this.x = x;
        this.y = y;
    }
    draw () {   //method
        console.log(`x: ${this.x} y: ${this.y}`);
    }
}
```

- One you make a param optional, all the right side params must be also optional!

Visibility: Access Modifiers

- If we don't want anybody to change a coordinate, e.g. p.x = 9
- We can use the access modifier **private**.
- We have 3 access modifiers: **public (default), private** and **protected**.

```
class Point {
    private x: number;
    private y: number;
    constructor (x?: number,y?: number){
        this.x = x;
        this.y = y;
    }
    draw () {
        console.log('x:' + this.x + 'y:' + this.y);
    }
}
```

- The **protected** modifier acts much like the private modifier with the exception that members declared **protected** can also be accessed by instances of deriving classes.

Access Modifier in Constructor (*)

A trick: when prefixing in a constructor with the access modifier, TS generates a field with the same name (this trick reduces the code):

```
class Point {
    private x: number;
    private y: number;
    constructor (x?: number,y?: number){
        this.x = x;
        this.y = y;
    }
    draw () {
        console.log('x:' + this.x + 'y:' + this.y);
    }
}
```

With access modifier:

```
class Point {
    constructor (public x?: number, private y?: number){
        // no need:
        // this.x = x;
        // this.y = y;
    }

    draw () {
        console.log('x:' + this.x + 'y:' + this.y);
    }
}
```

Readonly Modifier

- As its name states, makes a variable, property or function parameter "readonly":

```
class Person {
    readonly name: string = "Test";
}

let aPerson = new Person();
aPerson.name = "hi"; // error
```

- You can set a readonly property in two places:

```
// When defined
class Person {
    readonly name;
    constructor (name: string){
        this.name = name;
    }
}
```

```
// In the constructor
class Person {
    constructor (readonly name: string){
    }
}
```

Getters & Setters (*)

- One question may arise: how to access private fields?
- The solution is to use methods that act as properties: these are getters and setters.
- Example:

```
class Point {
    constructor (private _x?: number, private _y?: number){  }

    get x(){ return this._x; }

    set x(value){
        if (value < 0)
            throw new Error('value cannot be less than 0');
        this._x = value; }

    draw () { console.log('x:' + this.x + 'y:' + this.y); }
}

let point = new Point(1,3);
let x = point.x;
```

Inheritance: Overriding Methods

- Consider the following snippet:

```
class Person {
    firstName: string;
    lastName: string;
    greet() {
        console.log('hi');
    }
}

class Programmer extends Person {

}

let aProgrammer = new Programmer();
aProgrammer.greet();
```

- We call a method on the parent (greet).
- We can override methods in children classes.

Inheritance: Using super

- We can also use the parent methods with the keyword ”`super`”:

```
class Programmer extends Person {
    greet() {
        console.log('hello world');
    }
    greetLikeNormalPeople() {
        super.greet();
    }
}
```

- Typically, `super` is used in constructors to call the parent constructor.

Objects & Polymorphism

- Polymorphism in objects is the idea that you can have multiple instances of multiple classes referred to use a certain parent type.
- In our previous example, aProgrammer is of type `Programmer` but also `Person`.
- We can type it:

```
let aProgrammer: Person = new Programmer();
```

- But then, despite aProgrammer is of type Programmer, we cannot call Programmer methods only Person methods:

```
aProgrammer.greetLikeNormalPeople(); // error
```

- In addition, when we call greet we use the method of the instance because aProgrammer is an instance of Programmer:

```
aProgrammer.greet(); // get "hello world"
```

Classes that Implement Interfaces

Let's implement the **Person** interface:

```
interface Person {
    firstName: string;
    lastName: string;
    getFullName(): string;
}

class Foo implements Person {
  firstName: string;
  lastName: string;
  getFullName(): string { //if this method not implemented or use another name for it, leads to an Error
    return this.firstName + this.lastName;
  }
}
let foo = new Foo()
foo.firstName = "Joe";
foo.lastName = "Smith";
console.log(foo.getFullName());
```

Duck Typing (*) i

- Consider the following example:

```
let aPerson: Person = new Foo(); // This works, we can use the interface as type
```

- But also, you can create an object that has the properties of the interface.
- This will be valid too (from a class that does not implement the interface).

```
let someObj = {
    firstName: "Test",
    lastName: "Test",
    getFullName: () => console.log("Test")
};
```

- TS allows us to use someObj as an instance of Person.

```
aPerson = someObj; // ok because someObj has the same structure.
```

- This is called "duck typing".

Duck Typing (*) ii

- Duck typing means that "if looks like a duck, it's a duck" ;-)
- someObj has to had all the props of the interface (if not we get an error).
- someObj can have more props but then, they are not accessible from aPerson:

```
let someObj = {
    firstName: "Test",
    lastName: "Test",
    foo: "Test",
    getFullName:  () => console.log("Test")
};

aPerson = someObj;
aPerson.foo   //error!
```

Outline

- Generics allow us to parametrize types.
- E.g. in functions you pass parameters or arguments for different behaviors:

```
function echo(arg){
    return arg;
}

echo(1);    // returns number
echo('hi'); // returns a string
```

- So, how do I type this function? input and return?
- A solution can be use any:

```
function echo(arg: any): any {
    return arg;
}
```

- But then, we loose the fact that the return type is the same as the argument type:

```
let myString: string = echo(1); // this works but we would like not to.
```

Generics Concept

- Generics allow us to do express conditions over types:

```
function echo<T>(arg: T): T {
    return arg;
}
```

- With <T> we tell TS that echo is a generic function.
- Then, we can type with this generic type our args and return.
- Note. You don't have to use T but it is a common name for this placeholder.
- Now the following fails:

```
let myString: string = echo(1);
```

Conditions for Generics i

- Consider the following **personEcho()** function:

```
function personEcho<T>(person: T): T{
  return person;
}

class Person {
  constructor (public firstName: string, public lastName: string){ }
  getfullname () { return this.firstName + " " + this.lastName; }
}

class Admin extends Person {}

class Manager extends Person {}

let admin = new Admin('a','a');
let manager = new Manager('b','b');

console.log(personEcho(admin)); // Admin { firstName: 'a', lastName: 'a' }
console.log(personEcho(manager)); // Manager { firstName: 'b', lastName: 'b' }
console.log(personEcho(7)); // Problem: this also works, but we want to only echo Persons!
```

Conditions for Generics ii

- We could get rid of generics:

```
function personEcho(person: Person): Person{
    return person;
}
```

- But then, foo becomes of type Person:

```
let foo = personEcho(admin);
```

- We want to express that our function operates with classes that extend `Person`:

```
function personEcho<T extends Person>(person: T): T {
    return person;
}
```

- We define a restriction for the place holder, `<T extends Person>` means that the type must extend Person.

Definitions and Assignments (* remembering) i

- Consider the following functions:

```
function echo<T>(arg: T): T {
    return arg;
}

function isString<T>(arg:T):boolean {
    return (typeof(arg)==='string')?true:false;
}

let myEcho;
myEcho = echo;
console.log(myEcho(5));
```

- Now, suppose that we change the value of `myEcho()` as follows:

```
myEcho = isString;

console.log(myEcho(5)); //result: false
```

- We changed the type of `myEcho()`!

- As usual, remember to **define and assign** to get the type:

```
let myEcho: <T>(arg: T) => T;
```

- Now, if someone tries to assign a function not fulfilling the type we get an error from the compiler:

```
myEcho = isString;
// Compiler error: Type '<T>(arg: T) => boolean' is not assignable to type '<T>(arg: T) => T'.
```

Generics in Classes

- Classes can use generics which are listed in angle brackets following the name of the class:

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

Using Multiple Generics

```
class KeyValuePair<T,U> {
    constructor(private key: T,private  val: U) {
        this.key = key;
        this.val = val;
    }
    display():void {
        console.log(`Key = ${this.key}, val = ${this.val}`);
    }
}

let kvp1 = new KeyValuePair<number, string>(1, "Steve");
kvp1.display(); //Output: Key = 1, Val = Steve

let kvp2 = new KeyValuePair<string, string>("CEO", "Bill");
kvp2.display(); //Output: Key = CEO, Val = Bill
```

Classes Implementing Interfaces

- We can create classes that implement interfaces:

```
interface IKeyValuePair<T,U> {
    key : T;
    val : U;
    display():void
}

class KeyValuePair<T,U> implements IKeyValuePair<T,U> {
    constructor(public  key: T, public val: U) {
        this.key = key;
        this.val = val;
    }
    display():void {
        console.log(`Key = ${this.key}, val = ${this.val}`);
    }
}

let kvp1 = new KeyValuePair<number, string>(1, "Steve");
kvp1.display(); //Output: Key = 1, Val = Steve

let kvp2 = new KeyValuePair<string, string>("CEO", "Bill");
kvp2.display(); //Output: Key = CEO, Val = Bill
```

Outline

Type Assertions (”Casting”)

Type Assertions

Type assertions are used when **you know more than the compiler**.

Type assertions allow us to express a more specific type.

```
/* Inside an HTML file we have: <input id="myInp" type="text">  */

/* We select the element with getElemntByID which returns an HTMLElement */
let myInp = document.getElementById("myInp");
```

- Notice that **we know** that the element is of the more specific type `HTMLInputElement` (an input element).
- We want to assert that the type of `myInp` variable is `HTMLInputElement`.
- In this way, we can use properties only available in `HTMLInputElement`.
- In other languages type assertions are called ”casting”.

Syntax of Type Assertions

- We have two syntax to express type assertions:

```
let myInp = <HTMLInputElement>document.getElementById("myInp");   // option 1
```

```
let myInp = document.getElementById("myInp") as HTMLInputElement;   // option 2
```

- Now we can use the value property only available in `HTMLInputElement`:

```
console.log(myInp.value);
```

- Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only as-style assertions are allowed.

Outline

Type Aliases (`type`)

- Types define the form of data.
- Precisely, the most important aspect of Typescript is defining types and enforcing rules considering these types.
- For defining types, the language provides two similar syntaxes: `type` and `interface` (we will see differences next).

Type Aliases

A type alias creates a new name for a type. We can use type aliases to make the code more readable and self descriptive.

type vs. **interface**

- In most cases, **types** and **interfaces** are very similar.
- See the following examples:

```
interface PointInterface {
    x: number
    y: number
}

interface CounterFn { (start:number): string }
```

```
type PointType = {
    x: number
    y: number
}

type CounterFn = (start: number) => string
```

- Note that:
    - The **interface** syntax is a **definition**.
    - The **type** syntax is an **assignment**.
- In particular cases, a syntax is more convenient that the other or even a syntax is simply not possible at all as we will show next.

Simple Type Aliases

- The name of variable say that it is a mark of a student:

```
let mark: number = 85 ;
```

- To be more precise you can define a type:

```
type Mark = number;
let mark: Mark = 85;
mark = "Pass"; // Error
```

- Define a Grade with the Type of string:

```
type Grade = string ; // Fail, Pass, Credit, Distinction
let grade: Grade = "Credit";
grade = 75; //Error
```

Intersection Types

- With **type** we can elegantly define **intersection types**.
- An intersection type combines multiple types into one.
- This allows you to add together existing types to get a single type that has all the features you need:

```
type ExtendedPerson = Person & Serializable & Loggable;
```

- An object of type ExtendedPerson will **have all members of all three types**.

https://www.typescriptlang.org/docs/handbook/advanced-types.html#intersection-types

Union Types

- With **type** we can elegantly define **union types**:

```
type Result = Mark | Grade;

let result : Result = "Distinction";
result = 85;
result = "Fail";
```

- Now the result variable can be either number or string (Mark or Grade).
- If we have a value that has a union type, we can **only access members that are common to all types in the union**.

https://www.typescriptlang.org/docs/handbook/advanced-types.html#union-types

Generic Type Aliases

- We can use Generics for defining a type:

```
type Pair<T> = [T,T];
let point: Pair<number> = [10,5];
let keyValue: Pair<string> = ["Joe", "Fail"];
```

- The point variable is defined as a pair of two numbers using `Pair<number>`
- The keyValue is defined as pair of two strings using `Pair<string>`

Utility Types

TypeScript provides several utility types to facilitate common type transformations.

These utilities are available globally:

```
Partial<T>
Omit<T,K>
Readonly<T>
Record<K,T>
Pick<T,K>
Exclude<T,U>
Extract<T,U>
NonNullable<T>
ReturnType<T>
InstanceType<T>
Required<T>
ThisType<T>
```

More info at:
`https://www.typescriptlang.org/docs/handbook/utility-types.html`

Let’s see some examples...

Partial<T>

Partial constructs a type with all properties of T set to optional:

```
interface Todo { title: string; description: string; }

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
    return { ...todo, ...fieldsToUpdate };
}

const todo1 = { title: 'organize desk', description: 'clear clutter'};
const todo2 = updateTodo(todo1, { description: 'throw out trash'});
```

Omit<T,K>

Constructs a type by picking all properties from `T` and then removing the properties of `K`:

```
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

type TodoPreview = Omit<Todo, 'description'>;

const todo: TodoPreview = {
    title: 'Clean room',
    completed: false,
};
```

Example with Object/Function (*) i

- Let's elaborate a more complex example using an object that acts as both a function and an object.
- To do so, you have to define a type for a function (callable object) and the static properties that you want:

```
// With interface
interface Counter {
    (start: number): string; // callable part
    interval : number; // static props & methods
}
```

```
// With type alias
type Counter = {
    (start: number) : string; // callable part
    interval : number; // static properties & methods
}
```

```
const myFunctionObject: Counter =  ( (start: number) => {
    myFunctionObject.interval = start;
    return `Start is ${start}`; } ) as Counter;
myFunctionObject.interval = 7;
```

We need to use `as Counter` because interval is mandatory but we only set the callable part in the next line.

Example with Object/Function (*) ii

In general, it is usually a good practice to dissect definitions in two parts:

```
//via interface
interface CounterFn {
    (start: number): string
}

interface CounterStatic {
    interval: number
}

interface Counter extends CounterFn, CounterStatic {}
```

```
// via type alias
type CounterFn = (start: number) => string

type CounterStatic = {
    interval : number
}

type Counter = CounterFn & CounterStatic;
```

Implements with Union Type Aliases (*) i

- With type aliases you can use union types (which are "or" conditions in type definitions).
- In this case, you cannot use **`implements`** on a class:

```
// This will trigger compile errors
class Point {
    x: number;
    y: number;
}
interface Shape { area(): number;   }
type Perimeter = { perimeter(): number;  }
type RectangleShape = (Shape | Perimeter) & Point;

class Rectangle implements RectangleShape {
    x = 2;
    y = 3;
    area() {
        return this.x * this.y;
    }
}
```

- A class can only implement an object with types statically known (and here we have or conditions).

Implements with Union Type Aliases (*) ii

- Where unions make sense is for object definition via object literal:

```
const rectangle : RectangleShape = {
    x: 12,
    y: 133,
    perimeter() {
        return 2 * (rectangle.x + rectangle.y);
    },
    area () {
        return rectangle.x * rectangle.y
    },
}
```

- If you remove perimeter() **OR** area() the code compiles.
- But, the previous code does not compile if you remove **BOTH** methods (does not fulfill the or condition).

Extend Interfaces with Unions (*)

- You cannot use **extends** on an interface with type alias if you use **union** operator within your type definition:

```
type ShapeOrPerimeter = Shape | Perimeter
interface RectangleShape extends ShapeOrPerimeter, Point {}
```

- Again, similarly to class **implements** usage, interface is a "static" blueprint.
- The compiler will complain saying that an interface can only extend an object type or intersection of object types with statically known members.

Declaration Merging for Type Alias (*)

Merging works with **`interface`** but doesn't work with **`type alias`**:

```
interface Box {
    height: number
    width: number
}

interface Box { scale: number }

const box : Box = { height:5, width: 6, scale: 10 }
```

```
// Doesn't work with type aliases, type is an unique entity
type Box = {
    height: number
    width: number
}
type Box = { scale: number }

const box : Box = { height:5, width: 6, scale: 10 }
```

- Declaration merging is important for 3rd party libraries.
- Gives the consumer the option to extend them if some definition are missing.
- This is the only use case, where you definitely should always use interface instead of type alias!

Outline

Type Guards Motivation

Type Guards

Type guards allow checking the type of a variable at run time. They act like a ”filter” to decide if a variable is of a certain type or not.

There are 3 types of type guards:

1. `typeof`: available in JS/TS checks basic types: number, string, object, ...
2. `instanceof`: available in JS/TS checks if an object is an instance of an object created with a constructor (i.e. using `new`).
3. **Custom guard**: A custom guard is a function and it is useful for checking types of objects not built with a constructor (without `new`).
   The type guard function receives an object and returns a boolean saying if the object is of a specific type or not.
   The return is expressed using the keyword ”`is`” as we will see later.

Type Guard with **typeof**

- The following code illustrates the use of **typeof**:

```
function logMessage (msg: string | Error): void {
  if(typeof msg === 'string'){       // the type guard
      console.log(msg.toUpperCase()) // toUpperCase() only available on string
  } else{
      console.error("Error: " + msg.message); // message only available on error
  }
}

try{
    logMessage("This is a normal Message"); // result: THIS IS A NORMAL MESSAGE
    throw new Error('Whoops!');
} catch (e) {
    logMessage(e); // result: Error: Whoops!
}
```

- Notice that the editor can detect the types when you hover.

Type Guard with **instanceof**

- Consider that we want to calculate the area of a Circle and a Rectangle.
- We want to distinguish each geometry to do the proper computation.
- The problem with **typeof** is that it always returns "object" for any object:

```
class Rectangle {
  constructor (public width: number, public height: number){}
}
class Circle {
  constructor (public radius: number) {}
}

console.log (typeof(rec));   // result: 'object'
console.log (typeof(cir));   // result: 'object'
```

- In this case, we should use **instanceof**:

```
function getArea(geometry: Rectangle | Circle): number {
  if (geometry instanceof Circle){ return Math.PI * Math.pow(geometry.radius,2) }
  else { return geometry.height * geometry.width; }
}
let rec = new Rectangle(10,5);
let cir = new Circle(5);
console.log(getArea(rec));
console.log(getArea(cir));
```

Custom Guards i

- Problem with `instanceof`:
  - You cannot check the type of an object not created with `new`.
  - In particular, you cannot check the type of an object that implements an interface (cannot use `instanceof` with interfaces).
- In this case, you have to define a custom function as type guard.
- Let's see an example, first we define an interface for Point:

```
interface Point {
  x: number;
  y: number;
}
```

- Next, we change the code for Rectangle and Circle to include an anchor point:

```
class Rectangle {
  constructor (public topLeft: Point, public width: number, public height: number){}
}
class Circle {
  constructor (public center:Point, public radius: number) {}
}
```

Custom Guards ii

- Now, we can create instances of these new objects:

```
let rec = new Rectangle( { x:20, y:30 }, 10 , 5);
let cir = new Circle( { x:30, y:40 }, 5);
```

- Consider that we want to create a method that returns the centre of given object, let's try the following code:

```
let a: Point = {x:20, y:40};
function logCenter(geometry: Point | Rectangle | Circle): void {
  if(geometry instanceof Point) {  // This causes errors!
    console.log (geometry.x, geometry.y);
  } else if (geometry instanceof Rectangle) {
    console.log(geometry.topLeft.x + geometry.width/2, geometry.topLeft.y + geometry.height/2) ;
  } else { console.log(geometry.center); }
}
logCenter(a);
```

- The problem is that `instanceof` works only with instances of objects.

Custom Guards iii

- The compiler will complain:

```
// 'Point' only refers to a type, but is being used as a value here.
// Property 'x' does not exist on type 'Point | Rectangle | Circle'.
// Property 'y' does not exist on type 'Point | Rectangle | Circle'.
// Property 'center' does not exist on type 'Point | Circle'.
```

- To fix this, let's create a custom guard to check the type in this case:

```
function isPoint(geometry: any): geometry is Point {
  return typeof geometry.x === 'number' && typeof geometry.y === 'number';
}
```

- Then, replace the type guard for the point as follows:

```
// if(geometry instanceof Point) /* ... */
if(isPoint(geometry)){ /* ... */ }
```

Outline

Elaborated Intersection Type (*)

Next, we provide an example of a function that receives two arguments and extends the first argument with the properties of the second argument:

```
class Person { constructor(public name: string) { } }

interface Loggable { log(name: string): void; }

class ConsoleLogger implements Loggable { log(name) { console.log(`Hello, I'm ${name}.`); } }

function extend<First, Second>(first: First, second: Second): First & Second {
    const result: Partial<First & Second> = {};
    for (const prop in first) {
      if (first.hasOwnProperty(prop)) {
          (result as First)[prop] = first[prop];
      }
    }
    for (const prop in second) {
      if (second.hasOwnProperty(prop)) {
          (result as Second)[prop] = second[prop];
      }
    }
    return result as First & Second;
    }

// Use our function to extend jim with log method:
const jim = extend(new Person('Jim'), ConsoleLogger.prototype);
jim.log(jim.name);
```

Generics and "+" Operator (*) i

- Consider that you have the following two functions:

```
function add(a:number, b:number):number  { return a + b ; }
function addString(a:string, b:string):string  { return a + b ; }

console.log(add(5,6));  //result: 11
console.log(addString("Joe "," Smith")); // result: Joe Smith
```

- Now, using Generics you try to define a single type:

```
function add<T> (a:T, b:T):T  { return a + b ;}

console.log(add (5,6));
console.log(add('Joe ', 'Smith')); // result: Joe Smith
```

- We will get a compiler error:

```
Operator '+' cannot be applied to types 'T' and 'T'.
```

Generics and "+" Operator (*) ii

- It is obvious we can use **+** operator for numbers and strings.
- But when we use **<T>** it means any type can be used.
- So it means it is possible to pass objects to the methods:

```
let obj1 = {
    position: 'Cashier',
    type: 'hourly'};
let obj2 = {
    position: 'Manager',
    type: 'weekly'};

function add (a ,b) {
    return a + b;
}
console.log(add(obj1,obj2));
```

- In the previous example adding these objects makes no sense.
- This is why the compiler prevents us from using **+** operator.

Generics and ”+” Operator (*) iii

- To Make it work we have to restrict the types for T only to string and number.
- In addition, we have to explicitly define the type of the return value using the ternary operator:

```
function add<T extends string | number>(a: T, b: T): T extends string ? string : number  {
    return  <any>a + <any>b; // cast to any as unions cannot be added, still have proper typings applied
}
console.log(add(5, 6)); // result: 11
console.log(add('Joe ', 'Smith')); // result: Joe Smith
```

Outline

Modules i

- Essentially, a module is a file with functionality.
- Consider the following snippet:

```
class Point {
    constructor (private x?: number, private y?: number) {
    }

    draw () {
        console.log('x:' + this.x + 'y:' + this.y);
    }
}

let point = new Point(1,3);
point.draw();
```

- We have the definition and usage of `Point` in the same file.
- We are going to use a module to move point definition to `point.ts`

Modules ii

- Then, we will call it from **`main.ts`**

```
// point.ts:
export class Point {
    constructor (private x?: number, private y?: number){ }
    draw () { console.log('x:' + this.x + 'y:' + this.y); }
}
```

- We can export one or more types such as classes, functions, simple variables or objects.

```
// main.ts:
import { Point } from './point'; // Note that we do not use .ts
                                 // comma separated for multiple imports
let point = new Point(1,3);
point.draw();
```

- Note. When you import you also execute the code in the module.
- Typically, we create an index.ts file per directory that imports all the ts files.

Modules Under the Hood i

External modules are used to organize/encapsulate your code AND to locate your code at runtime.

- In practice, you have two choices at runtime:
    - Combine all transpiled code into one file.
    - Use external modules and have multiple files and require some other mechanism to get at those files.
- External modules originated with server-side JS:
    - There is a one-to-one correspondence between an external module and a file on the file system.
    - You can use the file system directory structure to organize external modules into a nested structure.

Modules Under the Hood ii

If you want to use external modules on the client side, in a browser:

- It gets more complex as there's no equivalent to the file system that has the module available for loading.
- So now on the client side you need a way to bundle all those files into a form that can be used remotely in the browser.
- Module bundlers are who allow runtime resolution of your external modules in the browser.

Namespaces

- External modules are used to organize code AND to locate your code at runtime.
- Namespaces are a TypeScript-specific way to organize code (older than ES6 modules).
- Namespaces are simply named JavaScript objects in the global namespace.
- A namespace is declared like this:

```
namespace Mynamespace {
  export const foo = 123;
}

console.log(Mynamespace.foo)
```

- Inside the namespace you can create whatever you need: classes, functions, etc.

Namespaces Across Files

Unlike modules, they can span multiple files, and can be concatenated using `--outFile`:

```
// file1.ts
namespace Mynamespace {
  export const foo = 123;
}
```

```
// file2.ts
namespace Mynamespace {
  export const bar = 567;
}
```

```
// app.ts
console.log(Mynamespace.foo,Mynamespace.bar);
```

We have to compile the previous files using `--outFile`:

```
$ tsc --outFile app.js file1.ts file2.ts app.ts
```

The order is important, `app.ts` must go at the end so previous definitions in the namespace are available (otherwise you get errors).

`app.js` is the compiled output.

Using `<reference>`

We can define the order of the files with `<reference />`:

```
/// <reference path="file1.ts" />
/// <reference path="file2.ts" />
console.log(Mynamespace.foo,Mynamespace.bar)
```

Now the following compiles:

```
$ tsc --outFile app.js app.ts file1.ts file2.ts
```

We can even omit the other files (compiler uses the references):

```
$ tsc --outFile app.js app.ts
```

In fact `--outFile` makes the compiler interpret the triple-slash references and imports.

Note. Still some frameworks use namespaces (mainly for being cross-file) but namespaces are **being less used** (deprecated?, modules are much more used).

Compiler arguments i

- Consider the following example:

```javascript
// test.js
let a;
a = "hi";
```

```
$ tsc test.js
$ tsc test.js --outFile myfile.js --watch
$ tsc --help
```

- We can create a config file (`tsconfig.json`) with:

```
$ tsc --init
```

- When we run `tsc` with a config file in place, the compiler looks for all the ts files.
- Notice that when we compile our previous source code we get a error.

Compiler arguments ii

- This is because of the option:

```
"strict": true
```

- This means that all variables must be typed.
- Other options:

```
"outDir": is to specify the output directory
"noEmit": true
"noEmitOnError": true
```

- When compiling with accessors we will get an error because accessors are only available for ECMAScript 5 or higher.
- We need to pass the `--target` parameter to `tsc`:

```
$ tsc *.ts --target ES5
```

Creating a TS Project

- We can create a project with **npm** as usual:

```
$ npm init -y
$ tsc --init
```

- We will create a file **index.ts** that will compile to **index.js**.

```
// person.ts
export class Person {
    lastName: string;
    firstName: string;
}
```

```
// index.ts
import {Person} from './person';

let foo = new Person();
foo.firstName = "joe";
foo.lastName = "smith";
console.log(foo);
```

```
$ tsc
$ node out/index.js
```

- Create a script in packages.json:

```
"start": "tsc && node out/index.js",
```

TS Dependencies

- We probably want to install dependencies.
- We are going to install a JS package called lodash and we will use it in our TS code.

```
$ npm install lodash
```

- Now, we need to import lodash in our ts source code.
- We want to use an existing function called reverse.

```
import * as _ from 'lodash';

let array = [1,2,3,4]
_.   //autocomplete does NOT work!!
```

- This is because `lodash` is written in JS not with TS.
- One option is to install the version of `lodash` written in TS:

```
$ npm install lodash-ts
```

Type Definitions i

- Another option is to use ”type definitions”:
    - Types definitions allow us to define typing for JS source code.
    - ”Type definitions” do not implement the libraries but contain typing definitions for the lib API.
    - These type definitions are also available for many libraries in npm repos.
    - There is a separate package for these definitions that are in d.ts files and you get auto-complete and type-checking.
    - The common naming for this package is to call it with the prefix @types.
    - Many libraries have their TS definitions with @types.

- You can install the type definitions of lodash as follows:

```
$ npm install @types/lodash --save-dev
```

- Now, we have all the type information and intellisense.

```
import * as _ from 'lodash';

let array = [1,2,3,4]
console.log(_.reverse(array));
```