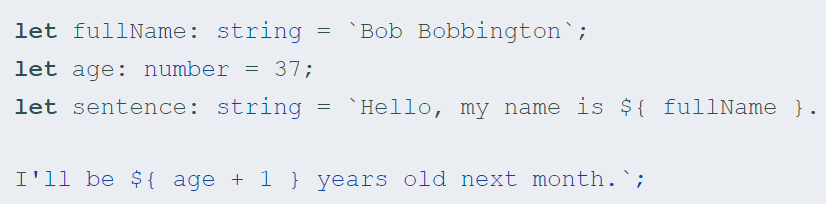
**Template Strings:**



* Nice typescript system for building and injecting strings. Note the backtick character on these special strings. `.

**Arrays**:

* Same as JavaScript:
* Let list: number[] = [1, 2, 3];
* Let list: Array<number> = [1, 2, 3];

**Enum:**

* C# style datatype that lets you basically correspond names to values. By default enums begin numberin at 0. You can change that manually by setting the value one of its members.
* Enum Color {Red = 1, Green, Blue}
* Let c: Color = Color.Green;
* You can also treat it as a dictionary with the number as an index, say if you have a value but not the corresponding name (let colorName = Color[2]).

**Any:**

* A type declaration that says ‘opt this variable out of type checking’. Note that declaring something an Any type makes it pretty fluid – think of it as safe mode. TypeScript will let you call any function you want to on it, then check if that function exists at runtime. If you instead declare the thing as an Object (like you could in JavaScript) the compiler will check and say ‘nah bro can’t compile, Object doesn’t have a .toFixed class!)
* Any’s also useful if you have an array with mixed types or similar.

**Void:**

* Absence of a type. Good as a return in a function that doesn’t include a value.
* Void vars can only be assigned a value of null.
* Note that **Null and Undefined** exist as they would in JS.

**Never**:

* Represents the type of values that never occur (i.e. a function that always throws an error or never returns). So basically if your function returns never it has to have an unreachable endpoint (because it’ll always throw or return an error).

**Object**:

* As in JS the object type is anything beyond a primitive type. It’s not a string, a number, a bool, an enum, it’s something more complex.

**Type Assertations**:

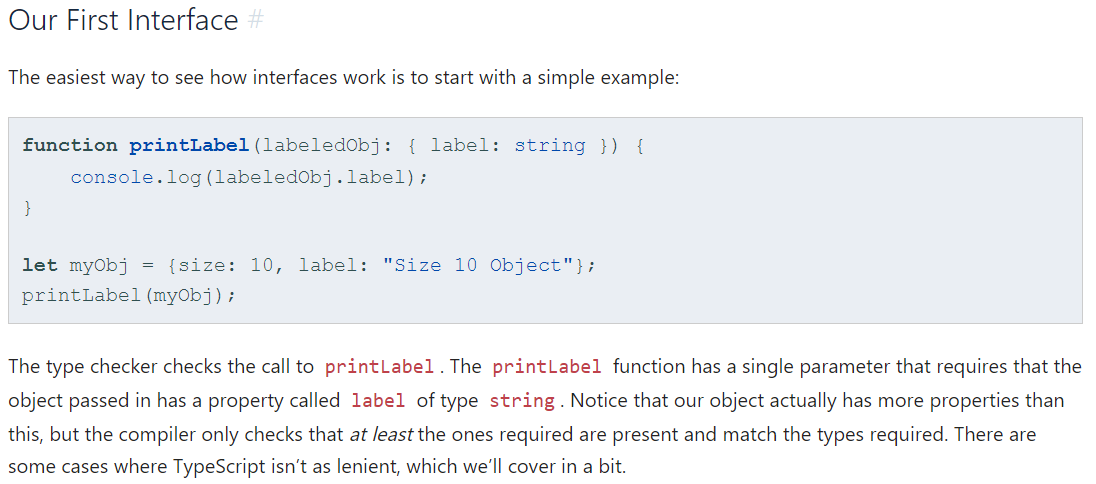
* When you know more about a variable than TypeScript does – say you know a variable will be a string at a given point, but TypeScript doesn’t, so the compiler won’t let you call a string method on it. Two optios:

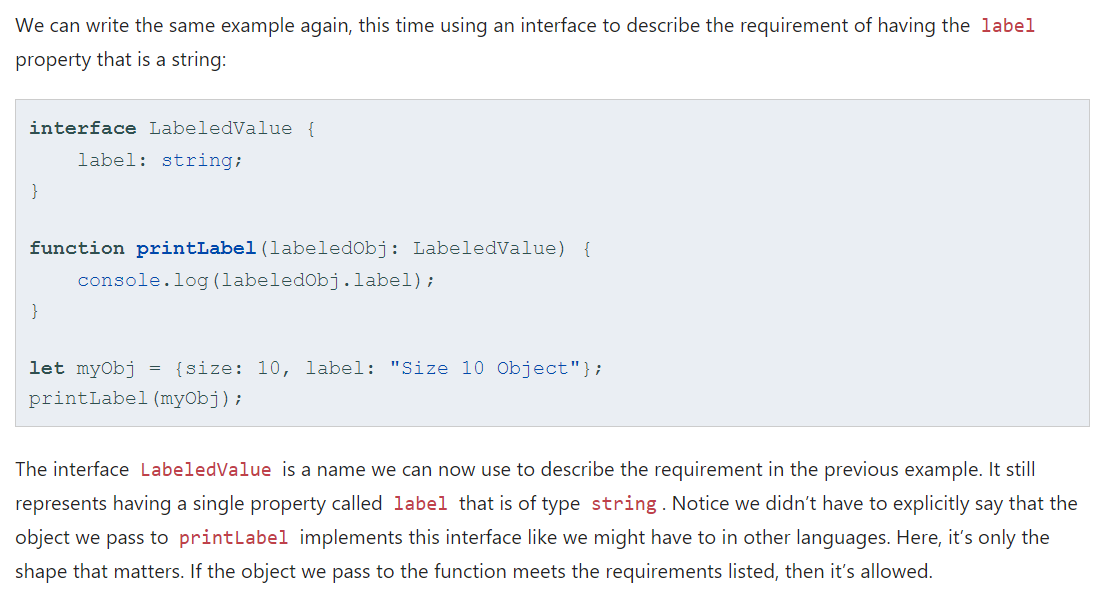
Let someValue: any = “this is a string, but the TS compiler doesn’t know it.”;

Let strLength: number = (<string>someValue).length;

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

Interfaces





So TypeScript does duck typing – it cares about the shape of data when checking Type. If it has wings, and bill, and feet, it’s a duck. Functions can take objects that need to have specific properties. Rather than say we want a duck object that has x, y, z, a, b, c., we can declare an interface with those properties, then required an object with a type of that interface. This means our function will only take an object that has all of those properties (if it has others too that’s cool – this is a minimum requirement to be valid input). Note that you don’t explicitly have to use an ‘implements’ keyword to say that the object we’re passing (printLabel) is implementing a certain interface, it just figures that out.

Note you can also add optional params to an interface:

Interface squareConfig {

Color?: string;

Width: number;

}

We’re requiring that any object passed as a squareConfig, but it doesn’t necessarily have to have a color.

**Readonly properties:**

* Some properties can only be modified at declaration. You’d add readonly to it:

Interface Point {

readonly x: number;

readyonly y: number;

}

**Readonly vs Const**

* Variables use const. Properties use readonly. If you’re declaring an