Generic Return Type, Optional Parameter and Default Value

In this lesson you will cover the concept of generic return type, optional parameter and default value.

WE'LL COVER THE FOLLOWING

Function Generic Return Type

TypeScript, since **version 2.4**, can infer the type of a function's return value. Before **version 2.4**, `s.length` would give an error. The `s` was not from a `U[]` type but an empty object literal {}.

```
function arrayMap<T, U>(f: (x: T) => U): (a: T[]) => U[] {
  return a => a.map(f);
}
const lengths: (a: string[]) => number[] = arrayMap(s => s.length);
```

Functions Optional Parameters

TypeScript lets you have an optional parameters with different syntax. The first way to accomplish this is by using a union to specify the type of the argument and the type undefined.

```
function undefinedOptional1(a: number | undefined, b: number) {}
undefinedOptional1(undefined, 1);
```

The second way is to use the question mark operator after the parameter name and before the semi-colon. In this case, any subsequent argument must also be optional. The following arguments cannot be of type `undefined` or `null` or a specific type unless they are optional.

The reason is that optional arguments don't need to specify their parameter's invocation while the union with undefined still requires the user to pass undefined. A standard check against `undefined` is needed to figure out if the argument is optional or if `undefined` has been assigned. It is not possible to distinguish between optional or `undefined` because both use the same comparison.

> Note: the below code throws an error

```
function undefinedOptional2F(a?: number, b: number) {
    // Doesn't compile because a is using optional: ?
}
undefinedOptional2F(); // Doesn't compile because of b is after a the optional


function undefinedOptional3F(a: number, b: number | undefined) {}
undefinedOptional3F(a); // Doesn't compile, must pass for "b" undefined if not needed
```

Function with a default value

It is also possible to have a default value for one of the parameters. Again, the syntax leverages existing knowledge, which is to use the single equal sign to set a variable to the argument. One difference is that you can set the default at every position in the signature. The reason is that contrary to the optional scenario, the default argument is always a value. The type is optional since TypeScript can infer it from the assignment which is always present by default, but it's still a good practice to make it explicit. With the default parameters, the function uses the value when it doesn't receive any value from the caller. The second case is when the function receives the value undefined.

```
function funcWithDefault(p1: string = "v1", p2?: number, p3 = true) {
    console.log("P1", p1); // v1 since undefined
    console.log("P2", p2); // undefined
    console.log("P3", p3); //boolean by inference
}

funcWithDefault(undefined);
```