Undefined Versus Null

In this lesson you will see the ubiquitous type undefined which is used to define something that does not exist.

undefined

A variable declared but not initialized is `undefined`. Undefined is not quite the same as the type `null`. In both cases, an assignment can set `undefine` or `null` to a variable explicitly. The following code does not compile because the variable is consumed before initialization and TypeScript, when configured to be strict, does not allow for interaction with an unassigned variable.

```
let variableNotInitialized:string;
console.log(variableNotInitialized);
```

TypeScript must be made stricter in order to prevent it from assigning null or undefined to every type. You must set TypeScript's `strictNullChecks` option to `true` to block the possibility implicitly assigning null and undefined to our variables (available since TypeScript 2.0).

```
// This code would work before TypeScript 2.0
let numberOnlyNotNullOrUndefined: number = 123;
numberOnlyNotNullOrUndefined = null;
numberOnlyNotNullOrUndefined = undefined;
console.log(numberOnlyNotNullOrUndefined)
```
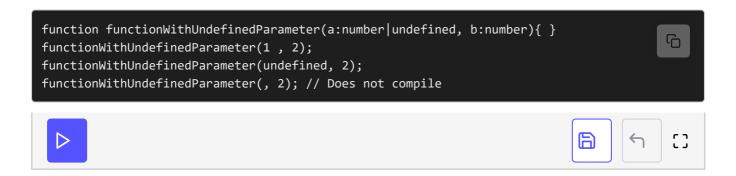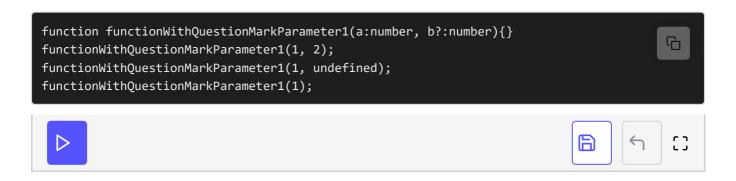
Being forced to be explicit about all possible values forces developers to use `union` or the question mark to optionally define the variable, which allows undefined. A nullable number is considered to be two types. Dual type (or more) is possible with a union. The union uses the pipe character between the main type (for example, a number) and null.

```
// This code would work before TypeScript 2.0
let numberNullOrUndefined: number | null | undefined = 123;
numberNullOrUndefined = null;
numberNullOrUndefined = undefined;
console.log(numberNullOrUndefined)
```

The union of any other type and `undefined` makes the type optional. Using question mark syntax or union with `undefined` produces a similar result with only minor differences.

```
function functionWithUndefinedParameter(a:number|undefined, b:number){ }
functionWithUndefinedParameter(1 , 2);
functionWithUndefinedParameter(undefined, 2);
functionWithUndefinedParameter(, 2); // Does not compile
```

The difference is that with `| undefined`, the parameter must be passed with the value or `undefined`. However, with `?` you can pass `undefined` or nothing at all.

```
function functionWithQuestionMarkParameter1(a:number, b?:number){}
functionWithQuestionMarkParameter1(1, 2);
functionWithQuestionMarkParameter1(1, undefined);
functionWithQuestionMarkParameter1(1);
```

The question mark is more succinct but also doesn't allow a non-undefined parameter to follow in a function signature. The following code does not

compile.

```
function functionWithQuestionMarkBeforeARequiredParameter(a:number?, b: number{}
```

This lesson was a glimpse of the values used for variables that are not defined or optionally defined. At anytime a value does not hold a valid value, `undefined` is the best option.

null

As discussed in the last section, increasing the strictness of TypeScript forces developers to use a question mark to define the variable optionally define their variables, which allows `undefined`. This distinction is desired to ensure consistency in our code. With a clean-cut assertion by type of what can be `undefined` (optional), `null` or with a specific value, the code becomes more comprehensible.

For example, a nullable number is two types, not a single one. Dual type (or more than one) is possible with the concept of *union*. The *union* uses the pipe character between the main type (for example, a number) and null.

```
let canBeANumberOrNull: number | null;
canBeANumberOrNull = 10;
canBeANumberOrNull = null;
// canBeANumberOrNull = undefined; // Uncomment and it will not transpile
```

The *union* of any other type and `undefined` makes the type *optional*. Using *question mark* syntax or *union* with `undefined` produces the same result with only minor differences. The *question mark* is more succinct but also does not allow a non-undefined parameter to follow in a function signature.

```
function function1(a:number|undefined, b:number){ }

function function2(a?:number, b:number){ } // Doesn't compile because a is using optional ? a
```

Optional parameters should always come after non-optional parameters. A union with undefined means that the user must specify `undefined` which is not really optional since it must be explicit.

```
function function1(a:number|undefined, b:number){
    console.log(a);
    console.log(b);
}
function function3(a:number, b?:number){
    console.log(a);
    console.log(b);
}

function1(1, 2);
function1(undefined, 2)
function3(3)
function3(3, undefined)
```

It is good practice is to avoid using `null` as much as possible and rely instead on `undefined`. This helps to avoid having to handle `undefined` **and** `null` as well as the actual type. We chose `undefined` instead of `null` because of the natural tendency of JavaScript to lean toward `undefined`. The following code does not compile. However, the same code in JavaScript (without the type specified) would print `undefined`.

```
let aNumber: number; // cannot be undefined  but is undefined
console.log(aNumber);
```

Members in a class set to a single type (without a union) cannot be defined as `null` or `undefined` explicitly but will be `undefined` until initialized. The time window between creates a state where the variable is `undefined`. A variable can be `undefined` regardless of the type. The following code would return `undefined` normally, but with strictness turned up, TypeScript will not compile instead returns an exception indicating that the variable must be initialized.

```
class MyClass{
```

```
    private a: number; // Error here
    constructor(){
        console.log(this.a); // It does not compile
    }
}
```

The following code demonstrates that `null` adds an additional level of complexity that *most* of the time can be avoided by using a type or `undefined`.

```
function f1(p1: null | undefined | number, p2: undefined | number, p3?: number ) {
    console.log(p1);
    console.log(p2);
    console.log(p3);
}
```

There are several use cases where `undefined` can be handy. One case is when a function or class does not require the variable; for example, an optional parameter or defined members that are not used all the time.

Another case is for optional data. Data can be optional and handled with a *default* behavior or value when code needs to access the value of the variable. Often, third-party libraries provide default values but let users customize function calls. The third-party code checks to see if the option is defined (not-undefined) and if so, uses it. Otherwise, if `undefined`, the library will use the default value.

```
function functionWithDefault(p1: number = 1) {
    console.log(p1);
}
functionWithDefault();
functionWithDefault(undefined);
functionWithDefault(100);
```

When data is pulled from external sources, undefined may be used. The variable starts `undefined` until the data arrives from the external sources. The
```

following code does not compile but exposes the idea.

```
function getDelayedValue(){
    let dataIsNotYetThere: string | undefined;

    dataIsNotYetThere = fetchData();

    return dataIsNotYetThere; // It "should" be with a value at that point
}
```

Undefined and optional transpile into the same code as if nothing were assigned to the variable. This is because JavaScript is not aware of the concept. For example, a numeric variable with no value will simply be `undefined` (not `null`).

A good use case for `null` is to differentiate between:

1. An uninitialized variable: should be `undefined`
2. A variable clearly representing the absence of something: should be `null`
3. A variable representing a value: should not be `undefined`, neither `null`

For example, consider the situation where you fetch data. The function can return an actual value, null if the data is actually present but has no value, or undefined if the data has not yet been computed.