Unknown: A Better any

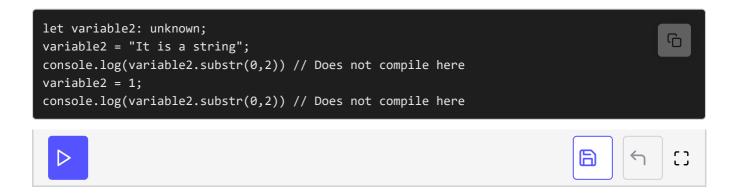
In this lesson, you will see the type 'unknown'.

The unknown type is halfway between a specific explicit type and the type any which allows everything. Declaring a variable as unknown allows us to set a wide variety of types without allowing unwanted access to properties or the value of a type. The following code demonstrates that a variable with type any can be assigned a string and then use a function of the string type.

Later, the variable is assigned to a number which does not have substr function. However, TypeScript does not catch an attempt to invoke a function that does not exist.

```
let variable1: any;
variable1 = "It is a string";
console.log(variable1.substr(0,2)) // Output "it"
variable1 = 1;
console.log(variable1.substr(0,2)) // Crash
```

Changing the type from any to unknown indicates to TypeScript that the type can receive any value but should be used cautiously. It does not allow the function to be invoked.



The only way to access hidden underneath properties or values is to explicitly tell TypeScript variable's type. This can be done by *casting* or by using a *type*

assertion. Here is an example that lets an unknown variable use the string

function substr. variable3 is of unknown type but explicitly cast by asserting its type as string.

Forcing a type is not recommended because it can lead to specifying the wrong one. For example, variable3 may be a number asserted to be string. Asserting an unknown type is dangerous and should be used with caution.

```
let variable3: unknown;
variable3 = "It is a string";
let variable3String = variable3 as string;
console.log(variable3String.substr(0,2))
```

unknown and null can both be validated without using == or === and because of JavaScript. Both are *falsy*.

```
let und: string | undefined = undefined;
if(und) {
   console.log(und)
} else {
   console.log("The value is undefined")
}
```

In case you need to display a value in an object that has many undefined/null (or optional) fields, several checks are required. The following example shows that only the last object displays the string because the others are nested with undefined values.

```
interface ObjectC {
    m3: string;
}
interface ObjectB {
    m2?: ObjectC;
}
interface ObjectA {
    m1?: ObjectB;
}
function print(o: ObjectA): void {
```

```
if (o.m1) {
        if (o.m1.m2) {
            console.log(o.m1.m2.m3);
        }
    }
const obj1: ObjectA = {
    m1: undefined,
};
const obj2: ObjectA = {
    m1: {
        m2: undefined,
};
const obj3: ObjectA = {
    m1: {
        m2: {
           m3: "Yeah!",
        },
};
print(obj1);
print(obj2);
print(obj3);
```

TypeScript 3.7 and up allows us to shortcut the conditions of null and undefined by using *optional chaining*. Optional chaining uses ?. and returns undefined if in the chain of ?. contains a property that is null or undefined. Otherwise, it returns the value. If you change the previous example to use optional chaining, the code is reduced to:

```
interface ObjectC {
    m3: string;
}
interface ObjectB {
    m2?: ObjectC;
}
interface ObjectA {
    m1?: ObjectB;
}

function print(o: ObjectA): void {
    if(o.m1?.m2){
      console.log(o.m1.m2.m3);
    }
}

const obj1: ObjectA = {
    m1: undefined,
};
const obj2: ObjectA = {
    m1. f
```

```
m1: {
          m2: undefined,
     },
};
const obj3: ObjectA = {
          m1: {
                m2: {
                      m3: "Yeah!",
                      },
        },
print(obj1);
print(obj2);
print(obj3);
```

In the same vein, TypeScript has *nullish coalescing* that allows the code to be reduced before invoking something that can be **null** or **undefined**. If you run the following code several times, once in a while you will get the value from the function and sometimes the one from the default value.

```
function getValue(): string | undefined{
  if (Math.random() > 0.5){
    return undefined;
  }
  return "Good";
}

let value = getValue();
if(!value){
  value = "Default"
}
console.log(value);
```

With TypeScript, since 3.7, it is possible to use ?? to avoid the if statement.

```
function getValue(): string | undefined{
  if (Math.random() > 0.5){
    return undefined;
  }
  return "Good";
}

let value = getValue() ?? "Default";
console.log(value);
```







A few lessons ago, you learned that the constructor of the Boolean object uses unknown. The constructor could take any, but with unknown, the type is sure to remain the same inside the boolean's constructor and keep the code inside the constructor in order to access a limited range of properties. This wraps up our discussion of the unknown type.