Nullish Coalescing

This lesson explains the new nullish coalescing operator.

WE'LL COVER THE FOLLOWING ^

- Nullish coalescing for an object
- Nullish coalescing for primitive

Nullish coalescing for an object

TypeScript 3.7 brought the *nullish coalescing* operator: ?? On the left side of the operator is a variable or condition. On the right side is the result **if** the condition is **null** or **undefined**.

One use case for this operator is to check if a variable is set to define a default value.

The invocation of **line 19** of the **myFunction** does not have a second parameter. It causes the code to use the default value at **line 8**.

```
interface Options {
  opt1: number;
  opt2: boolean;
}
function myFunction(id: number, options?: Options): void {
  let optionsOrDefault: Options;
  if (options === undefined) {
    optionsOrDefault = { opt1: -1, opt2: false }; // Default
  } else {
    optionsOrDefault = options;
  }

  console.log(`myFunction with id ${id}`);
  if (optionsOrDefault.opt2) {
    console.log(`--> Has option 1 with value ${optionsOrDefault.opt1}`);
  }
}

myFunction(1); // Uses default from line 8
myFunction(2, { opt1: 2, opt2: true });
```



Now, it is possible to reduce the amount of code with the nullish coalescing operator. **Line 6** is shorter because of the ?? syntax which uses the right side of the operator only of options is undefined.

```
interface Options {
  opt1: number;
  opt2: boolean;
}
function myFunction(id: number, options?: Options): void {
  let optionsOrDefault: Options = options ?? { opt1: -1, opt2: false };

  console.log(`myFunction with id ${id}`);
  if (optionsOrDefault.opt2) {
    console.log(`--> Has option 1 with value ${optionsOrDefault.opt1}`);
  }
}
myFunction(1);
myFunction(2, { opt1: 2, opt2: true });
myFunction(3, { opt1: 3, opt2: false });
```

It is worth noting that an alternative to this example is to use a default value instead of an optional one. In the following code, line 5 has a default value defined after the = sign. But, this is only because of the nature of the example.

```
interface Options {
   opt1: number;
   opt2: boolean;
}
function myFunction(id: number, options: Options = { opt1: -1, opt2: false }): void {
   console.log(`myFunction with id ${id}`);
   if (options.opt2) {
     console.log(`--> Has option 1 with value ${options.opt1}`);
   }
}
myFunction(1);
myFunction(2, { opt1: 2, opt2: true });
myFunction(3, { opt1: 3, opt2: false });
```





Any use cases that need a fallback when a value is **null** or **undefined** is a good candidate for *nullish coalesing*.

Nullish coalescing for primitive

So far, the lesson's examples have been focused on objects, but _nullish coalescing also works on primitives. For example, if the value is undefined, you can set a default value with a primitive type.

```
function operation(): number | undefined {
  return Math.random() > 0.5 ? 500 : undefined;
}
let x: number = operation() ?? -1;
console.log(x);
```

The example above has a function that returns a <code>number</code> or an <code>undefined</code> type. However, the type expected is a <code>number</code>. If the function cannot be changed to always return a number, then a solution is to verify that the value is a number or else has a default number value. The <code>nullish coalescing</code> works toward that solution by providing a compact syntax.