Optional Chaining and Optional Element Access

This lesson explains a feature of TypeScript 3.7 that will reduce the boilerplate of code when checking undefined/null type.

The object property chaining situation

Any language that has the possibility of a `null` value (and `undefined` for TypeScript) gets into a situation where a nested property is needed but within the chain of nested objects, that many variables can be `null` and hence need to be checked.

> The following code does not compile.

The reason is that you are accessing a nested property but at `level1`, `level2` or `level3` there is a chance that the variable is not defined (see **line 2 to 4** which indicates optional). This would cause a null reference exception.
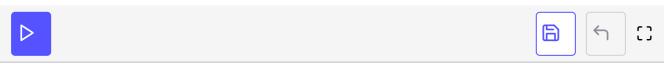
```
interface MyType {
  level1?: {
    level2?: {
      level3?: {
        name: string;
      }
    }
  }
}

let ex1: MyType = {
```

```
let ex1: MyType = {
  level1: {
    level2: {
      level3: {
        name: "Good"
      }
    }
  }
}

console.log(ex1.level1.level2.level3.name)
```

Previous art solutions to chaining

Before **TypeScript 3.7**, the only way to ensure that you wouldn't get in a situation of calling a function on an `undefined` object was to check against the `undefined` value. The **line 21** to **line 24** demonstrate the boilerplate to avoid null exception.

```
interface MyType {
  level1?: {
    level2?: {
      level3?: {
        name: string;
      }
    }
  }
}

let ex1: MyType = {
  level1: {
    level2: {
      level3: {
        name: "Good"
      }
    }
  }
}

if (ex1.level1 !== undefined
  && ex1.level1.level2 !== undefined
  && ex1.level1.level2.level3 !== undefined) {
  console.log(ex1.level1.level2.level3.name);
}
```

A shortcut often used was to leverage the *falsy* behavior of JavaScript and to reduce:

```
if (ex1.level1 !== undefined
  && ex1.level1.level2 !== undefined
  && ex1.level1.level2.level3 !== undefined) {...
```

to:

```
if (ex1.level1
  && ex1.level1.level2
  && ex1.level1.level2.level3) {...
```

While syntactically valid and a very popular approach, there are two issues with the pattern.

First, it is cumbersome and requires quite a lot of code to simply access a value. The example provided does not account for the fact that each `if` usually has an `else` that returns `undefined` instead of the value when something is `undefined` along the nested structure.

Second, if a property is checked without specifying a counter type (e.g, `undefined`) then many other values can get into the condition like `true` or `1`. It might not be an issue in the present case, but code evolves. More than once I have personally witnessed issues where a type changed and even though the code compiled, it ultimately failed because of the assumption that an object would be present and not a primitive type.

Solutions to optional chaining

Version **3.7** brought optional chaining to Typescript, which fixes the nested `undefined` object problem by removing the need to check every level. Instead, it provides an operator, the `?.` that will verify the value. If the value is `undefined` or `null` the variable will be set to `undefined`. Otherwise, the value of the property is returned. **Line 21** demonstrates how short is the optional chaining solution compared to the previous code block with many `&&`.

```
interface MyType {
  level1?: {
    level2?: {
      level3?: {
        name: string;
```

```
        }
      }
    }
  }

  let ex1: MyType = {
    level1: {
      level2: {
        level3: {
          name: "Good"
        }
      }
    }
  }

  console.log(ex1?.level1?.level2?.level3?.name);
```

As you can see in **line 21**, the code is more succinct and has fewer instance of `if` or `else` being not well handled. It also mitigates issues with the *falsy* type. The last example that demonstrates what is happening if, among the nested property is `undefined`.

```
interface MyType {
  level1?: {
    level2?: {
      level3?: {
        name: string;
      }
    }
  }
}

let ex1: MyType = {
  level1: {
    level2: undefined
  }
}

console.log(ex1 ?.level1 ?.level2 ?.level3 ?.name);
```

The transpilation is successful with a value of `undefined` and does not return into an error.

Accessing optional element

Similarly, some boilerplate code was needed to check the value before accessing it. In the case of an optional parameter, verifying that the type was assigned to a value before accessing its properties is a repetitive task.

```
interface MyType {
  action: (s: string) => void;
}

let ex1: MyType = {
  action: (s: string) => { console.log(`Run ${s}`); }
}

function execute(name: string, operation?: MyType): void {
  operation?.action(name);
}

execute("Fast", ex1);
execute("Slow");
```

The example above executes the operation only if the optional parameter is defined. But, **line 10** is executed only if the `operation` is defined. Before, you would have needed to do:

```
if(operation !== undefined){
 operation?.action(name);
}
```