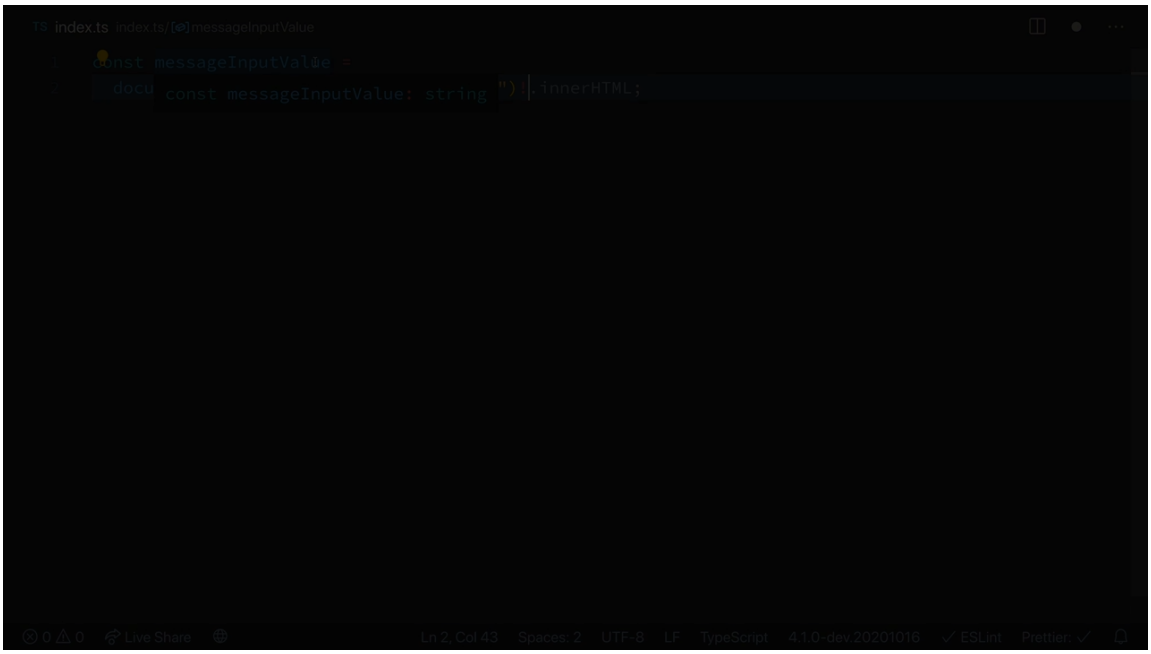

Often, we can't know for certain whether a variable actually has a value or if it is `null` or `undefined`. For example, we can use `document.getElementById()` to get a DOM element with a specific ID. However, if no element exists with the ID that we provide, the function will return `null`. If we have the `strictNullChecks` flag in tsconfig.json turned on, TypeScript will warn us that the element might not exist and require us to perform a runtime check to make sure the value is present. This helps us avoid `Cannot read property "x" of undefined` errors.

Lets use `document.getElementById` to demonstrate possible ways to handle `null` and `undefined` values.

Truthy Evaluation

If we naïvely try to change the `innerText` of an HTML element, we will likely get a type error.

```
document.getElementById("messageInput").innerText = "
// Type Error: Object is possibly 'null'.
```

The easiest and most commonly used type guard is an `if` condition that checks

truthiness. This means passing in the value to our conditional without doing any kind of check using strict equals (`===`). JavaScript will allow the condition to pass if the value is truthy; if the value is falsy, it will not pass. If our conditional passes, TypeScript knows that inside of our conditional, the value cannot be `null` or `undefined` .

```
const messageInput = document.getElementById("message
if (messageInput) {
  messageInput.innerText = "Alex";
}
```

Be aware that this also excludes legitimate values that are falsy, such as `0` , `''` , and `false` . You'll have to do additional checks if you expect your variable might be one of those falsy values.

Optional Chaining

Things can get verbose if we are trying to access properties deep within an object's structure. We have to add several layers of `if` statements, or use the logical AND operator (`&&`) to sort out whether the object property exists or not.

```
const messageInputElement = document.getElementById("
if (messageInputElement) {
  const parentElement = messageInputElement.parentEle
  if (parentElement) {
    const messageInputParentInnerHTML = parentElement
    if (messageInputParentInnerHTML) {
      // ...
    }
  }
}
```

Optional Chaining is a relatively new JavaScript operator which allows you to attempt to access properties on an object whether or not they actually exist. This can be really useful when we want to access deep properties of objects that

might not actually exist. We can prepend a question mark to our "dot property access" (`.`) operator to create the Optional Chaining operator (`?.`) that lets us safely access any property, even optional or undefined properties. If the property doesn't actually exist or is `null` or `undefined` , JavaScript will quietly return `undefined` for the whole expression.

```
const messageInputParentInnerHTML = document.getEleme
  "messageInput",
)?.parentElement?.innerHTML;
messageInputParentInnerHTML; // const messageInputPar
```

Isn't that nicer to read?

Our variable has a type of `string | undefined` because if the expression evaluates all the way to the end, `innerHTML` is a string; but if the expression fails anywhere along the way, either if `.getElementById()` returned null or if our `#messageInput` element doesn't have a parent element, the entire expression will return `undefined` . We can then do a truthiness check to see whether `messageInputParentInnerHTML` is a `string` or not.

```
const messageInputParentInnerHTML = document.getEleme
  "messageInput",
)?.parentElement?.innerHTML;

if (messageInputParentInnerHTML) {
  console.log(
    `The current contents are ${messageInputParentInn
  );
}
```

Also note that Optional Chaining only fails if the property is `null` or `undefined` ; falsy values like `0` and `false` will pass through correctly.

We can use optional chaining to optionally access items on arrays as well. We just prepend the optional chaining operator to the square brackets of our index access. In this next example, our API could correctly return a list of teachers, or it

could return an error message.

```
function handleTeacherAPIResponse(response: {
  teachers?: string[];
  error?: string;
}) {
  const firstTeacher = response.teachers?.[0];
  firstTeacher;
}
```

Finally, we can optionally call methods on objects that may or may not exist. To do this, we prepend the optional chaining operator to the parentheses which call the function.

```
async function makeAPIRequest(
  url: string,
  log?: (message: string) => void,
) {
  log?.("Request started.");
  const response = await fetch(url);
  const data = await response.json();

  log?.("Request complete.");

  return data;
}
```

If we were to not pass a value in for the `log` function, the calls to that function would silently fail without throwing an error. It's as if we wrapped the function call in a truthiness check for the function.

```
async function makeAPIRequest(url, log) {
  if (log) {
    log('Request started.)
  }
```

```
  }
  // ... etc.
}
```

Nullish Coalescing

When you have the option, using a default value for variables that might be undefined is really helpful. Typically we use the logical OR operator (`||`) to assign the default value.

```
const messageInputValue =
  document.getElementById("messageInput")?.value || "
messageInputValue; // const messageInputValue: string
```

The only problem with this approach is it checks against falsy values, not just `null` and `undefined` . That means when our `#messageInput` field is empty, it will still give us the default value since empty `string` is falsy.

The Nullish Coalescing operator solves this by only checking if a value is `null` or `undefined` . It works the same way as the logical OR, but with two question marks (`??`) instead of two vertical bars.

```
const messageInputValue =
  document.getElementById("messageInput")?.value ?? "
messageInputValue; // const messageInputValue: string
```

Non-null Assertion

Sometimes, you just know better than the type checker. If you positively know that a particular value or property is not `null` or `undefined` , you can tell the type checker using the Non-null Assertion operator (`!.`), which looks a lot like the Optional Chaining operator. Using this operator essentially tells the type checker "There's no way this property could possibly be `null` or `undefined` , so just pretend like it's the correct type."

```
const messageInputValue = document.getElementById("me
  .value;

messageInputValue; // const messageInputValue: string
```

Notice we don't have to add a default value. TypeScript trusts that we know what we are doing and will give us the value with the appropriate type, no questions asked. Be aware that, unlike the other two operators, the Non-null assertion operator only exists in TypeScript. If you tried using it in JavaScript, it would throw a syntax error.

Remember, any time we override the default behavior of the type checker, we run the risk of introducing type errors that the type checker cannot catch for us. Using the Non-null Assertion operator removes the type safety which we wanted TypeScript to give us in the first place, so if it is at all possible, handle `null` and `undefined` properties with one of the other methods before resorting to Non-null Assertions.

Next Lesson or discuss in the Community