DOM Manipulation

DOM Manipulation

An exploration into the `HTMLElement` type

In the 20+ years since its standardization, JavaScript has come a very long way. While in 2020, JavaScript can be used on servers, in data science, and even on IoT devices, it is important to remember its most popular use case: web browsers.

Websites are made up of HTML and/or XML documents. These documents are static, they do not change. The *Document Object Model (DOM)* is a programming interface implemented by browsers to make static websites functional. The DOM API can be used to change the document structure, style, and content. The API is so powerful that countless frontend frameworks (jQuery, React, Angular, etc.) have been developed around it to make dynamic websites even easier to develop.

TypeScript is a typed superset of JavaScript, and it ships type definitions for the DOM API. These definitions are readily available in any default TypeScript project. Of the 20,000+ lines of definitions in *lib.dom.d.ts*, one stands out among the rest: `HTMLElement` . This type is the backbone for DOM manipulation with TypeScript.

> You can explore the source code for the [DOM type definitions]

Basic Example

Given a simplified *index.html* file:

```html
<!DOCTYPE html>
<html lang="en">
  <head><title>TypeScript Dom Manipulation</title></head>
  <body>
    <div id="app"></div>
    <!-- Assume index.js is the compiled output of index.ts -->
    <script src="index.js"></script>
  </body>
</html>
```

Let's explore a TypeScript script that adds a `<p>Hello, World!</p>` element to the `#app` element.

```
// 1. Select the div element using the id property
const app = document.getElementById("app");

// 2. Create a new <p></p> element programmatically
const p = document.createElement("p");

// 3. Add the text content
p.textContent = "Hello, World!";

// 4. Append the p element to the div element
app?.appendChild(p);
```

After compiling and running the *index.html* page, the resulting HTML will be:

```
<div id="app">
  <p>Hello, World!</p>
</div>
```

The Document Interface

The first line of the TypeScript code uses a global variable `document`. Inspecting the variable shows it is defined by the `Document` interface from the *lib.dom.d.ts* file. The code snippet contains calls to two methods, `getElementById` and `createElement`.

`Document.getElementById`

The definition for this method is as follows:

```
getElementById(elementId: string): HTMLElement | null;
```

Pass it an element id string and it will return either `HTMLElement` or `null`. This method introduces one of the most important types, `HTMLElement`. It serves as the base interface for every other element interface. For example, the `p` variable in the code example is of type `HTMLParagraphElement`. Also, take note that this method can return `null`. This is because the method can't be certain pre-runtime if it will be able to actually find the specified element or not. In the last line of the code snippet, the new *optional chaining* operator is used to call `appendChild`.

`Document.createElement`

The definition for this method is (I have omitted the *deprecated* definition):

```
createElement<K extends keyof HTMLElementTagNameMap>(tagName: K, options?: Ele
createElement(tagName: string, options?: ElementCreationOptions): HTMLElement;
```

This is an overloaded function definition. The second overload is simplest and works a lot like the `getElementById` method does. Pass it any `string` and it will return a standard HTMLElement. This definition is what enables developers to create unique HTML element tags.

For example `document.createElement('xyz')` returns a `<xyz></xyz>` element, clearly not an element that is specified by the HTML specification.

> For those interested, you can interact with custom tag elements using the `document.getElementsByTagName`

For the first definition of `createElement`, it is using some advanced generic patterns. It is best understood broken down into chunks, starting with the generic expression: `<K extends keyof HTMLElementTagNameMap>`. This expression defines a generic parameter `K` that is *constrained* to the keys of the interface `HTMLElementTagNameMap`. The map interface contains every specified HTML tag name and its corresponding type interface. For example here are the first 5 mapped values:

```
interface HTMLElementTagNameMap {
    "a": HTMLAnchorElement;
    "abbr": HTMLElement;
    "address": HTMLElement;
    "applet": HTMLAppletElement;
    "area": HTMLAreaElement;
        ...
}
```

Some elements do not exhibit unique properties and so they just return `HTMLElement`, but other types do have unique properties and methods so they return their specific interface (which will extend from or implement `HTMLElement`).

Now, for the remainder of the `createElement` definition: `(tagName: K, options?: ElementCreationOptions): HTMLElementTagNameMap[K]`. The first argument `tagName` is defined as the generic parameter `K`. The TypeScript interpreter is smart enough to *infer* the generic parameter from this argument. This means that the developer does not have to specify the generic parameter when using the method; whatever value is passed to the `tagName` argument will be inferred as `K` and thus can be used throughout the remainder of the definition. This is exactly what happens; the return value `HTMLElementTagNameMap[K]` takes the `tagName` argument and uses it to return the corresponding type. This definition is how the `p` variable from the code snippet gets a type of `HTMLParagraphElement`. And if the code was `document.createElement('a')`, then it would be an element of type `HTMLAnchorElement`.

The `Node` interface

The `document.getElementById` function returns an `HTMLElement`. `HTMLElement` interface extends the `Element` interface which extends the `Node` interface. This prototypal extension allows for all `HTMLElements` to utilize a subset of standard methods. In the code snippet, we use a property defined on the `Node` interface to append the new `p` element to the website.

`Node.appendChild`

The last line of the code snippet is `app?.appendChild(p)`. The previous, `document.getElementById`, section detailed that the *optional chaining* operator is used here because `app` can potentially be null at runtime. The `appendChild` method is defined by:

```
appendChild<T extends Node>(newChild: T): T;
```

This method works similarly to the `createElement` method as the generic parameter `T` is inferred from the `newChild` argument. `T` is *constrained* to another base interface `Node`.

Difference between `children` and `childNodes`

Previously, this document details the `HTMLElement` interface extends from `Element` which extends from `Node`. In the DOM API there is a concept of *children* elements. For example in the following HTML, the `p` tags are children of the `div` element

```
<div>
  <p>Hello, World</p>
  <p>TypeScript!</p>
</div>;

const div = document.getElementsByTagName("div")[0];

div.children;
// HTMLCollection(2) [p, p]

div.childNodes;
// NodeList(2) [p, p]
```

After capturing the `div` element, the `children` prop will return an `HTMLCollection` list containing the `HTMLParagraphElements`. The `childNodes` property will return a similar

`NodeList` list of nodes. Each `p` tag will still be of type `HTMLParagraphElements`, but the `NodeList` can contain additional *HTML nodes* that the `HTMLCollection` list cannot.

Modify the HTML by removing one of the `p` tags, but keep the text.

```
<div>
  <p>Hello, World</p>
  TypeScript!
</div>;

const div = document.getElementsByTagName("div")[0];

div.children;
// HTMLCollection(1) [p]

div.childNodes;
// NodeList(2) [p, text]
```

See how both lists change. `children` now only contains the `<p>Hello, World</p>` element, and the `childNodes` contains a `text` node rather than two `p` nodes. The `text` part of the `NodeList` is the literal `Node` containing the text `TypeScript!`. The `children` list does not contain this `Node` because it is not considered an `HTMLElement`.

The `querySelector` and `querySelectorAll` methods

Both of these methods are great tools for getting lists of dom elements that fit a more unique set of constraints. They are defined in *lib.dom.d.ts* as:

```
/**
 * Returns the first element that is a descendant of node that matches selecto
 */
querySelector<K extends keyof HTMLElementTagNameMap>(selectors: K): HTMLElemen
querySelector<K extends keyof SVGElementTagNameMap>(selectors: K): SVGElementT
querySelector<E extends Element = Element>(selectors: string): E | null;

/**
 * Returns all element descendants of node that match selectors.
 */
querySelectorAll<K extends keyof HTMLElementTagNameMap>(selectors: K): NodeLis
querySelectorAll<K extends keyof SVGElementTagNameMap>(selectors: K): NodeList
querySelectorAll<E extends Element = Element>(selectors: string): NodeListOf<E
```

The `querySelectorAll` definition is similar to `getElementsByTagName`, except it returns a new type: `NodeListOf`. This return type is essentially a custom implementation of the standard JavaScript list element. Arguably, replacing `NodeListOf<E>` with `E[]` would result in a very similar user experience. `NodeListOf` only implements the following properties and methods: `length`, `item(index)`, `forEach((value, key, parent) => void)`, and numeric indexing. Additionally, this method returns a list of *elements*, not *nodes*, which is what `NodeList` was returning from the `.childNodes` method. While this may appear as a discrepancy, take note that interface `Element` extends from `Node`.

To see these methods in action modify the existing code to:

```
<ul>
  <li>First :)</li>
  <li>Second!</li>
  <li>Third times a charm.</li>
</ul>;

const first = document.querySelector("li"); // returns the first li element
const all = document.querySelectorAll("li"); // returns the list of all li ele
```

Interested in learning more?

The best part about the *lib.dom.d.ts* type definitions is that they are reflective of the types annotated in the Mozilla Developer Network (MDN) documentation site. For example, the `HTMLElement` interface is documented by this HTMLElement page on MDN. These pages list all available properties, methods, and sometimes even examples. Another great aspect of the pages is that they provide links to the corresponding standard documents. Here is the link to the W3C Recommendation for HTMLElement.

Sources:

- ECMA-262 Standard
- Introduction to the DOM

Customize

Site Colours: System

Code Font: Cascadia

Popular Documentation Pages

Everyday Types
All of the common types in TypeScript

Creating Types from Types
Techniques to make more elegant types

More on Functions
How to provide types to functions in JavaScript

More on Objects
How to provide a type shape to JavaScript objects

Narrowing
How TypeScript infers types based on runtime behavior

Variable Declarations
How to create and type JavaScript variables

TypeScript in 5 minutes
An overview of building a TypeScript web app

TSConfig Options
All the configuration options for a project

Classes
How to provide types to JavaScript ES6 classes

Community

- Get Help
- Blog
- GitHub Repo
- Community Chat
- @TypeScript
- Mastodon
- Stack Overflow
- Web Repo

Using TypeScript

- Get Started
- Download
- Community
- Playground
- TSConfig Ref
- Why TypeScript
- Design
- Code Samples

Made with ♥ in Redmond,
Boston, SF & Dublin