

Lesson

Tuesday

# Introduction to Programming

## (/introduction-to-programming)

### / JavaScript and Web Browsers

## (/introduction-to-programming/javascript-and-web-browsers)

## / Understanding Web APIs: Interfaces (Object Types) and Inheritance

Text

In the last lesson, we learned about HTML DOM elements. These elements are objects and they have properties that developers can access to get and set DOM element data. Things like:

- Getting the value of the `id` attribute of a DOM element.
- Adding a brand new attribute to a DOM element.
- Setting the text of a DOM element, like for heading or paragraph elements.
- Setting inline styles of a DOM element with the `style` attribute.

In terms of Web APIs, these HTML element objects are called "interfaces". If you remember, Web API **interfaces** are simply different types of objects. In this lesson, we'll learn how to find the name of the Web API interface for the HTML DOM element that

we're working with. Why bother? Well, when we know the name of the Web API interface we're working with, we can use it to access the right MDN documentation to learn more about it.

We'll also learn how interfaces share functionality with each other through a mechanism called **inheritance**. Inheritance is a mechanism that many programming languages (like JavaScript, C# and Ruby!) use to share functionality between two objects. We won't get too deep in this, just enough to understand all of the Web API interfaces that provide functionality to the DOM elements that we work with.

As you work through this lesson, keep in mind the car analogy! In these lessons, we're "looking under the hood" of browser Web APIs to understand more deeply how these technologies are structured. **You won't be required to demonstrate an understanding of this information on your independent project**, but it will improve your resourcefulness as a developer and it is crucial to your long term growth.

## Web API Interfaces (Object Types)

---

For Web APIs, an **interface** is a package of functionality that the browser makes available to developers in order for them to use a browser structure or tool in their code. Specifically for Web APIs, interfaces are always objects. (Note that this isn't true for all APIs.) To distinguish between different interfaces, they have different names. For example, `window` and `Document` are two interfaces.

It's less common to use the terminology "interface" and way more common to call `window` and `Document` "objects", because they are in fact objects! In a previous lesson, we made clear that we'll favor calling all Web API interfaces "objects" in LearnHowToProgram.com.

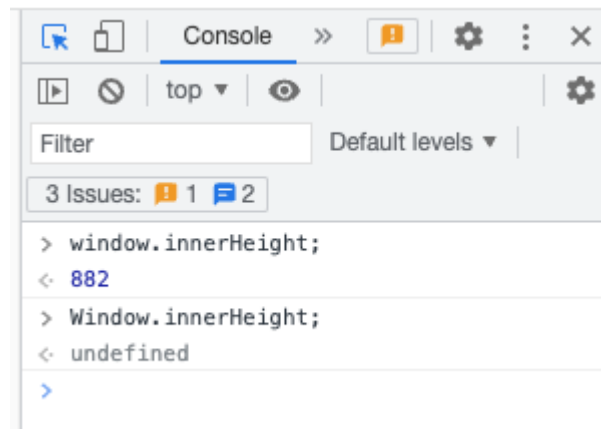
However, there's an important technical distinction to learn about objects: there's the actual object with data that's saved to a variable, and then there's the **type** of the object, or **object type**.

Let's take a moment to understand the difference between an object type and an actual object.

## Object Types versus Actual Objects

Let's look at an example: `window` is an actual object with data saved in a variable, and `Window` (<https://developer.mozilla.org/en-US/docs/Web/API/Window>) with a capital `w` is the object type.

Notably, we can't access the object type `Window` (with a capital `w`) to use its properties. Check out the image below that's an example of this using the DevTools console. In the first input we access the `window` object and get the value of the `innerHeight` property. This works as expected because we're using the actual object. In the second input, we enter `Window.innerHeight` and get `undefined`. This doesn't work because we're referencing the object type.



We can think of an object type like a recipe to make cookies — it defines exactly what a cookie contains, but it isn't the cookie itself. On the other hand, the actual object with data corresponds to the actual cookie that we can eat.

**To distinguish between object types and variables that contain objects, we use different casing:**

- lowerCamelCase for variables that are set to actual objects with data, like `window` and `document`. lowerCamelCase starts

with a lowercase letter, and if the variable name is more than one word, we remove all spaces and capitalize the first letter of each subsequent word.

- UpperCamelCase for object types like `window` (<https://developer.mozilla.org/en-US/docs/Web/API/Window>) and `Document` (<https://developer.mozilla.org/en-US/docs/Web/API/Document>).

## Getting the Object Type of HTML DOM Elements

So what interfaces (object types) are we working with when we access HTML DOM elements? Let's find out by working through an example. In the last lesson, we used this code to get the H1 DOM element and save it to a variable called `h1` :

```
let h1 = document.querySelector("h1");
```

The `h1` variable represents an actual object — the Heading element in the DOM. To find out the object type, we need to do the following in the DevTools console:

```
> let h1 = document.querySelector("h1");  
> Object.prototype.toString.call(h1);    // we pass in h1 as  
the argument to .call()  
'[object HTMLHeadingElement]'
```

In the above example, when we pass in the `h1` variable to the `.call()` method, we get a return value of `[object HTMLHeadingElement]`. This means that the `h1` variable is a specific type of object called `HTMLHeadingElement`. If we look on MDN, we can see that the `HTMLHeadingElement` object type (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLHeadingElement>) represents heading elements `<h1>` through `<h6>` in the DOM.


`Object.prototype.toString.call()`; is a way to get the exact type of an object, and it offers more specific information than using the `typeof` operator on objects. If we used the `typeof` operator, we'd just get `'object'` returned to us, which isn't helpful!

Note that we won't spend time understanding how `Object.prototype.toString.call()`; works because it's not important to be successful with this section's material. However, we can still use this code when we want to check the exact type of an object.

## Other DOM Element Object Types

There's a DOM element object type for every possible HTML element. If we had an image element in our webpage, we'd be working with an object of the type `HTMLImageElement`. Can you guess the name of the DOM element object for paragraphs? The correct answer is `HTMLParagraphElement`. How about the name of the DOM element object for an anchor element? The correct answer is `HTMLAnchorElement`.

To see a complete list of all HTML DOM element objects that represent specific HTML elements, like `HTMLHeadingElement` or `HTMLParagraphElement`, visit this link:

-  **HTML Element Interfaces**  
([https://developer.mozilla.org/en-US/docs/Web/API/HTML\\_DOM\\_API#html\\_element\\_interfaces\\_2](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API#html_element_interfaces_2))

## Sharing Functionality through Inheritance

It's very common for Web API interfaces to share functionality with each other through a mechanism called **inheritance**. Inheritance is a more advanced subject that we'll learn more about when we spend a course section on object-oriented JavaScript. For now,

understand that inheritance is a mechanism through which objects can share the functionality of other objects. Let's revisit our `h1` element as an example to better understand this.

```
let h1 = document.querySelector("h1");
```

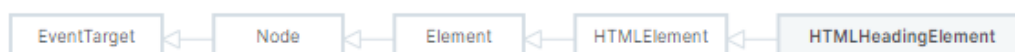
The `h1` variable contains an object of the type `HTMLHeadingElement`. Well, `HTMLHeadingElement` **inherits** functionality from many other objects. In fact, this is true for all DOM element objects, like `HTMLImageElement`.

Check out the following picture from MDN's docs on the `HTMLHeadingElement`. In the image we can see a diagram that shows the other object types (all of which are Web APIs) that `HTMLHeadingElement` inherits from:

- `HTMLElement` (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>)
- `Element` (<https://developer.mozilla.org/en-US/docs/Web/API/Element>)
- `Node` (<https://developer.mozilla.org/en-US/docs/Web/API/Node>)
- `EventTarget` (<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget>)

## HTMLHeadingElement

The `HTMLHeadingElement` interface represents the different heading elements, `<h1>` through `<h6>`. It inherits methods and properties from the `HTMLElement` interface.



This diagram is describing a **chain of inheritance**:

`HTMLHeadingElement` inherits from `HTMLElement`, which inherits from `Element`, which inherits from `Node`, which inherits from `EventTarget`. This means that all of the properties that belong to the `HTMLElement` object also belong to the `HTMLHeadingElement`. And, all of the properties that belong to the `Element` object also belong to the `HTMLElement` object, which also belong to the `HTMLHeadingElement`.

Inheritance between object types is similar to how every human inherits genes from their parents and grandparents. Your genes are uniquely yours (like with the `HTMLHeadingElement`), but you also share genes with your predecessors (`HTMLElement`, `Element`, and so on).

What's the point of inheritance? It allows coding languages and browser structures to be more organized and flexible: instead of just using one object type, we use many object types and share functionality between them. Notably, `HTMLHeadingElement` inherits from `HTMLElement`, `Element`, `Node`, and `EventTarget` because all of these object types together describe different information and functionality related to HTML DOM elements in general: things like the element's tag name, attributes, and inner text.

We don't need to worry about understanding exactly how inheritance works now, we just need to accept that the object type that we're working with (like `HTMLHeadingElement`) has access to functionality that is originally defined in other object types.

## The Implications of Inheritance

The important implication of inheritance for us is that functionality like the `tagName` property or the `removeAttribute()` method that we might call on our `<h1>` element could be defined in the `HTMLHeadingElement` object or any of the objects that it inherits from. This can be confusing at first, but more importantly, it can make it harder to navigate MDN documentation to find information about an HTML DOM element that you are working with.

Later in this course section we'll go over practical tips for researching about DOM elements and events. Right now, the goal of this lesson is to simply build awareness about the Web API interfaces (object types) that provide the functionality for the various DOM elements.

Let's review all of the properties and methods we learned about in the last lesson and see exactly which Web API interface each one belongs to:

- `className` belongs to the `Element` object type.
- `id` belongs to the `Element` object type.
- `style` belongs to the `HTMLElement` object type.
- `innerText` belongs to the `HTMLElement` object type.
- `tagName` belongs to the `Element` object type.
- `getAttribute()` belongs to the `Element` object type.
- `setAttribute()` belongs to the `Element` object type.
- `hasAttribute()` belongs to the `Element` object type.
- `removeAttribute()` belongs to the `Element` object type.

Note that when we say a property "belongs to" an object type, we mean that the property was originally defined in that object type. As we can see in this example, when we're working with our `h1` DOM element of the type `HTMLHeadingElement`, we're able to access properties from the objects it inherits from, in this case, `Element` and `HTMLElement`:

```
// h1 variable is of the type HTMLHeadingElement
> let h1 = document.querySelector("h1");
// tagName property is inherited from Element
> h1.tagName;
"H1"
// innerText property is inherited from HTMLElement
> h1.innerText;
"Best Chocolate Chip Cookies"
```



Notice how we don't reference `HTMLHeadingElement` to access it directly in the above examples, or `EventTarget`, `Node`, `Element`, or `HTMLElement` for that matter. That's because they are object types.

Instead, we use the built-in `document` methods (`document.querySelector()` and `document.getElementById()`) to access HTML DOM elements, and we let our browsers handle returning the correct object to us. This is unlike the built-in `window` and `document` variables that we directly reference in our code to access their properties, like with `window.innerHeight` or `document.body`.

## Takeaways

---

We don't need to remember all of the names of the object types in this lesson or worry about referencing MDN documentation now.

**The goal of this lesson is to simply build awareness about the Web API interfaces (object types) that provide the functionality for the various DOM elements.** Here are the important concepts to take away:

- Web API interfaces are just different types of objects.
- The object type is different than the actual object. The type is like the recipe for an object, describing the available properties and methods. The actual object is like the food that's made from the recipe, with the actual data and values.
  - Object types are written in UpperCamelCase.
  - Actual objects are saved in variables which are written in lowerCamelCase.
- We can call on the variable containing the actual object to access its data (properties) and call its methods:  
`window.innerHeight`. We can't call on an object type:  
`Window.innerHeight`. If we do that, we'll get `undefined`.
- We can call `Object.prototype.toString.call()` to find the exact type of an object. This method takes one argument, the variable containing the object that you want to find the exact type of. Doing so is optional and for exploratory purposes if

you want to research for documentation online for a specific DOM element.

- Inheritance is the mechanism through which objects share functionality with each other. For example, the `HTMLHeadingElement` object shares functionality with four other objects: `HTMLElement`, `Element`, `Node`, `EventTarget`. Each object type provides different functionality related to HTML elements in the DOM: accessing attributes, data, adding event handling, and more.

In the next lesson, we'll get a chance to practice finding the exact type of the DOM elements we are working with, and access/manipulate DOM elements.

[Previous \(/introduction-to-programming/javascript-and-web-browsers/accessing-html-element-attributes-and-properties-in-the-dom\)](#)

[Next \(/introduction-to-programming/javascript-and-web-browsers/practice-accessing-html-element-attributes-and-properties-in-the-dom\)](#)

Lesson 48 of 75

Last updated more than 3 months ago.

[disable dark mode](#)



© 2023 Epicodus (<http://www.epicodus.com/>), Inc.