Add elements to the DOM ›

Connect Dart and HTML

Contents

This tutorial is the first of a series on basic, low-level web programming with the dart:html library. If you use a web framework, some of these concepts might be useful, but you might not need to use the dart:html library at all.

What's the point?

- DartPad lets you write a simple Dart web app without HTML boilerplate.
- A Dart web app has Dart, HTML, and (usually) CSS code.
- Compile a web app's Dart code to JavaScript to run the app in any modern browser.
- An HTML file hosts your Dart code in a browser page.
- The DOM models a browser page in a tree/node structure.
- Use `querySelector()` with an ID to get an element from the DOM.
- CSS selectors are patterns used to select matching elements in the DOM.
- Use CSS rules to style elements.

To write a low-level web app with Dart, you need to understand several topics—the DOM tree, nodes, elements, HTML, and the Dart language and libraries.

The interdependencies are circular, but we have to begin somewhere, so we begin with a simple HTML file, which introduces the DOM tree and nodes. From there, you build a bare bones, stripped-down Dart app that contains just enough code to dynamically put text on the page from the Dart side.

Though simple, this example shows you how to connect a Dart app to an HTML page and one way that a Dart app can interact with items on the page. These concepts provide the foundation for more interesting and useful web apps.

About the Dart, HTML, and CSS triumvirate

If you've used DartPad, you may have already seen the Dart, HTML, and CSS tabs that let you write the code for a web app. Each of these three languages is responsible for a different aspect of the web app.

Language	Purpose
Dart	Implements the interactivity and dynamic behavior of the web app
HTML	Describes the content of the web app's page (the elements in the document and the structure)
CSS	Governs the appearance of page elements

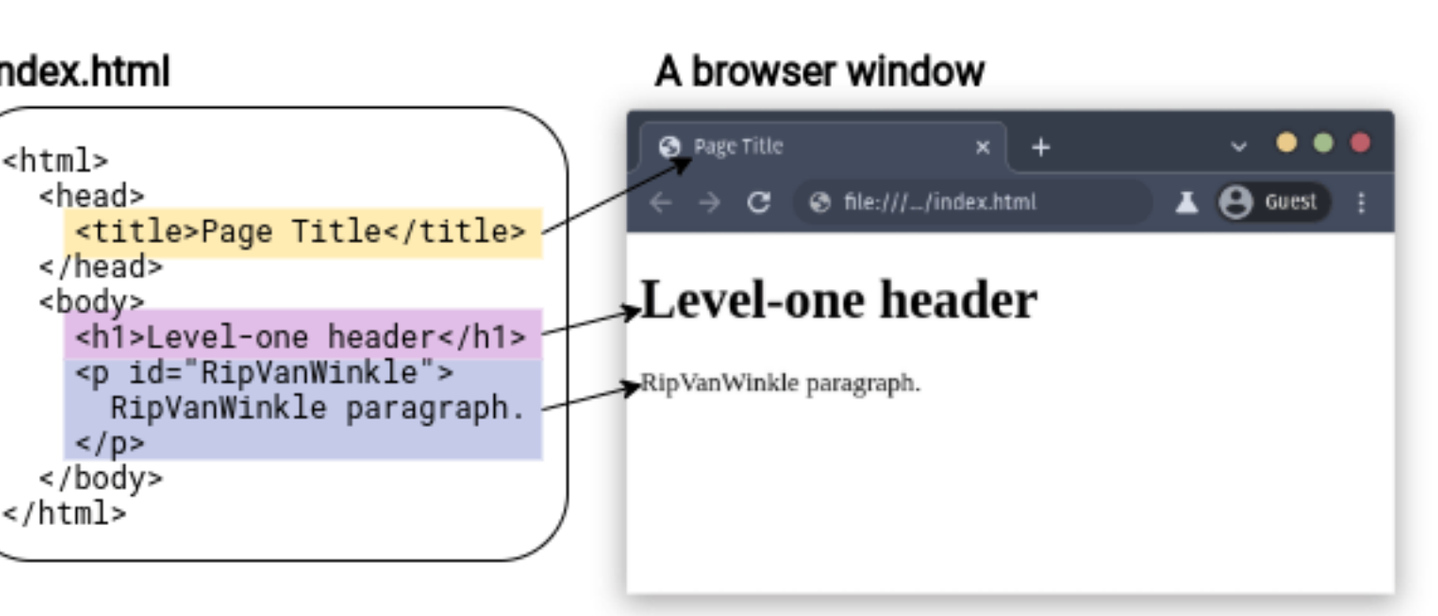
A Dart program can respond to events such as mouse clicks, manipulate the elements on a web page dynamically, and save information. Before the web app is deployed, the Dart code must be compiled into JavaScript code.

HTML is a language for describing web pages. Using tags, HTML sets up the initial page structure, puts elements on the page, and embeds any scripts for page interactivity. HTML sets up the initial document tree and specifies element types, classes, and IDs, which allow HTML, CSS, and Dart programs to refer to the same elements.

CSS, which stands for *Cascading Style Sheets*, describes the appearance of the elements within a document. CSS controls many aspects of formatting: type face, font size, color, background color, borders, margins, and alignment, to name a few.

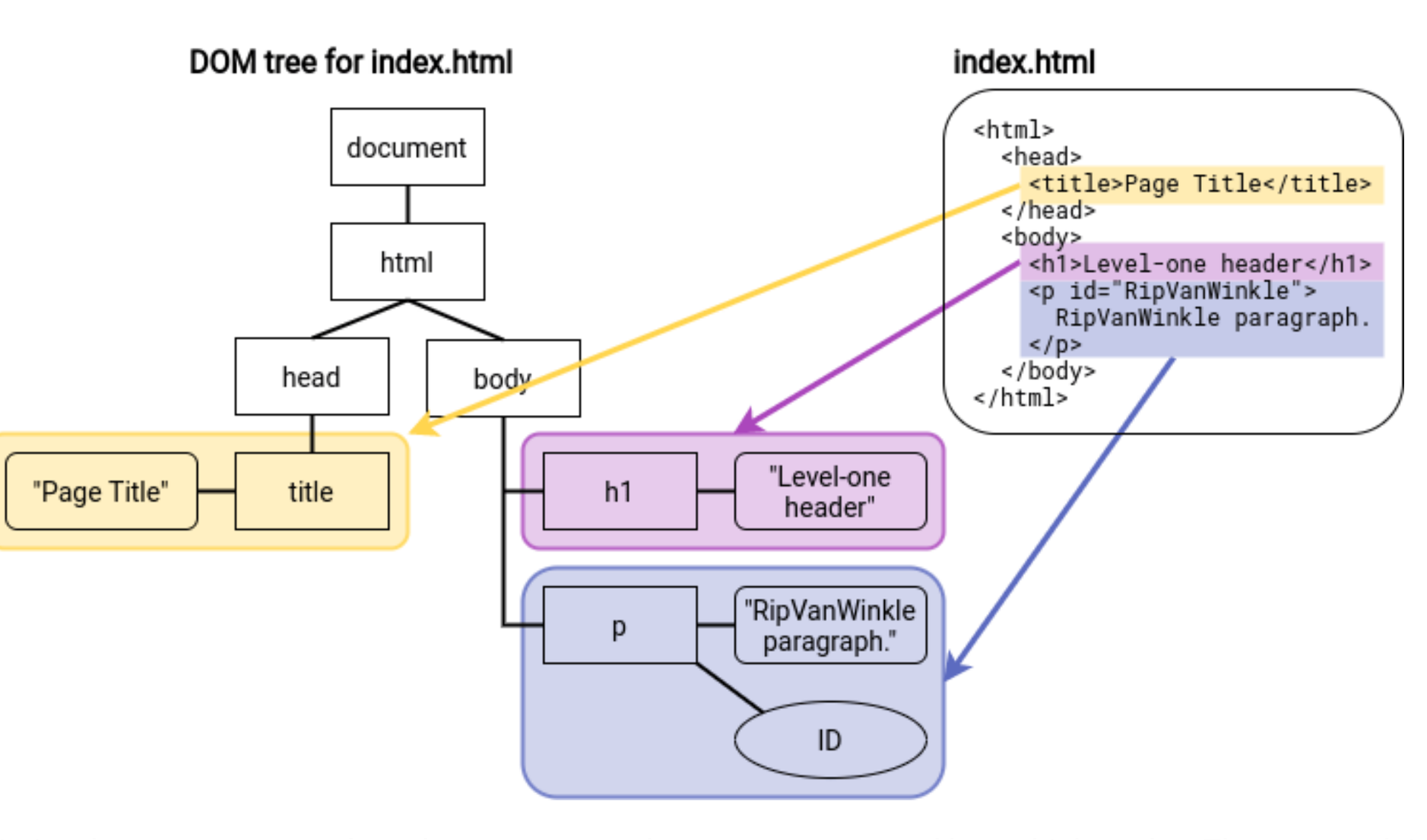
About the DOM

The Document Object Model (DOM) represents the structure of a web document as a tree of nodes. When an HTML file is loaded into a browser, the browser interprets the HTML and displays the document in a window. The following diagram shows a simple HTML file and the resulting web browser page in Chrome.



HTML uses tags to describe the document. For example, the simple HTML code above uses the `<title>` tag for the page title, `<h1>` for a level-one header, and `<p>` for a paragraph. Some tags in the HTML code, such as `<head>` and `<body>`, are not visible on the web page, but do contribute to the structure of the document.

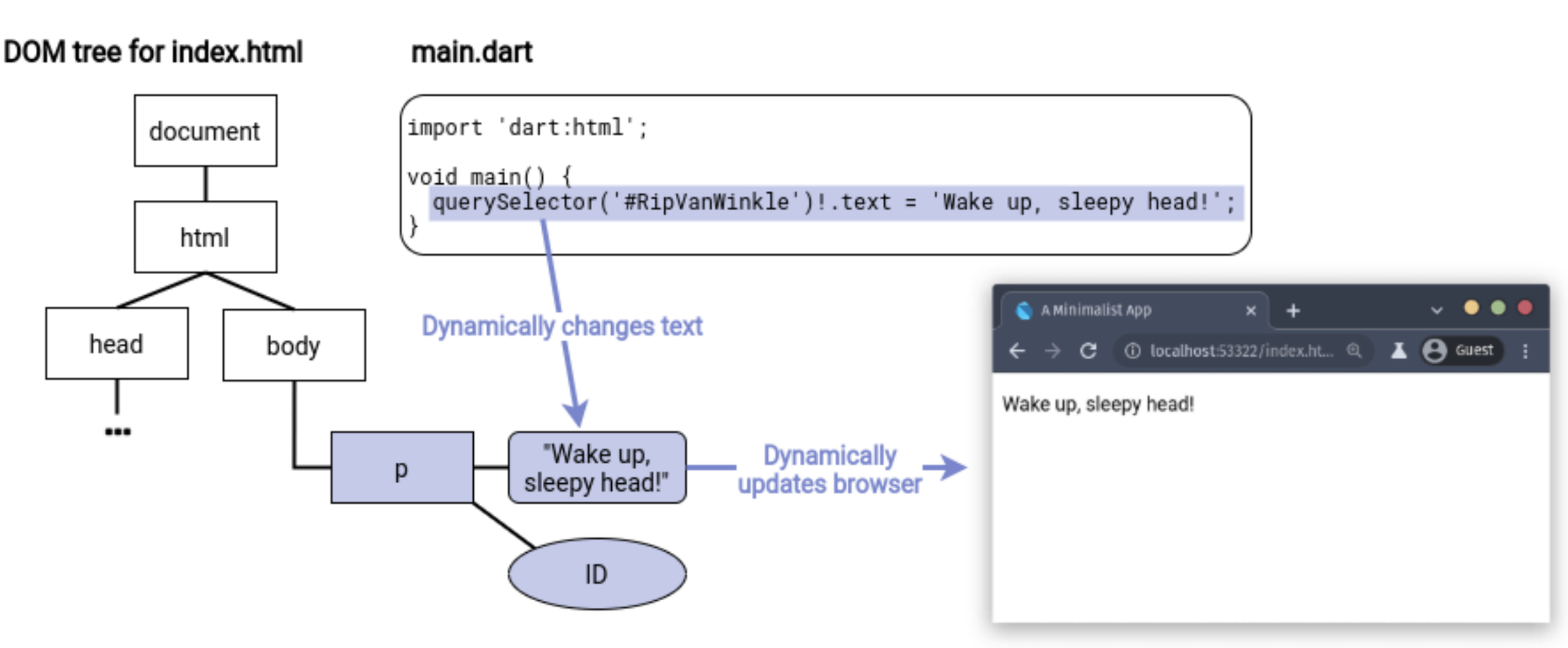
In the DOM, the document object sits at the root of the tree (it has no parent). Different kinds of nodes in the tree represent different kinds of objects in the document. For example, the tree has page elements, text nodes, and attribute nodes. Here is the DOM tree for the simple HTML file above.



Notice that some tags, such as the `<p>` paragraph tag, are represented by multiple nodes. The paragraph itself is an element node. The text within the paragraph is a text node (and in some cases, might be a subtree containing many nodes). And the ID is an attribute node.

Except for the root node, each node in the tree has exactly one parent. Each node can have many children.

An HTML file defines the initial structure of a document. Dart or JavaScript can dynamically modify that document by adding, deleting, and modifying the nodes in the DOM tree. When the DOM is changed, the browser immediately re-renders the window.



The diagram shows a small Dart program that makes a modest change to the DOM by dynamically changing a paragraph's text. A program could add and delete nodes, or even insert an entire subtree of nodes.

Create a new Dart app

1. Go to the DartPad.
2. Click the **New Pad** button to undo any changes you might have made the last time you visited DartPad.
3. Click **Dart**.
4. Select **HTML** (below **Dart**), so you can edit HTML and CSS in DartPad.

Note: These instructions feature DartPad, which hides some HTML boilerplate code. If you want to use any other editor, then we recommend starting with a small Dart web app sample and modifying the non-script tags inside the `<body>` section. HTML and Dart connections shows the full HTML code.

Edit the HTML source code

1. Click **HTML**, at the upper left of DartPad. The view switches from Dart code to the (currently non-existent) HTML code.
2. Add the following HTML code:

```
<p id="RipVanWinkle">
  RipVanWinkle paragraph.
</p>
```

3. Expand the output pane to see how a browser would render your HTML.

About the HTML source code

This HTML code is similar to the HTML code in the various diagrams earlier in this tutorial, but it's even simpler.

In DartPad you need only the tags you really care about—in this case: `<p>`. You don't need surrounding tags such as `<html>` and `<body>`. Because DartPad knows where your Dart code is, you don't need a `<script>` tag.

HTML and Dart connections shows the full HTML code that you need to run your web app outside DartPad.

The paragraph tag has the identifier `RipVanWinkle`. The Dart code you create in the next step uses this ID to get the paragraph element.

Edit the Dart source code

1. Click **Dart** at the upper right of DartPad. The view switches from HTML code to Dart code.
2. Change the Dart code to the following:

```
import 'dart:html';

void main() {
  querySelector('#RipVanWinkle')!.text = 'Wake up, sleepy head!';
}
```

3. Click **Run** to execute your code.

The text in the output pane changes to "Wake up, sleepy head!"

About the Dart source code

Let's step through the Dart code.

Importing libraries

The import directive imports the specified library, making all the classes and functions in that library available to your program.

The following `import` statement imports Dart's HTML library, which contains key classes and functions for programming the DOM:

```
import 'dart:html';
```

This program imports Dart's HTML library, which contains key classes and functions for programming the DOM. Key classes include:

Dart class	Description
Node	Implements a DOM node.
Element	A subclass of Node; implements a web page element.
Document	Another subclass of Node; implements the document object.

The `dart:core` library, which is automatically imported, contains many other useful classes, such as: `List`, a parameterized class that can specify the type of its members. An instance of `Element` keeps its list of child Element objects in a `List<Element>`.

Using the querySelector() function

This app's `main()` function contains a single line of code that is a little like a run-on sentence with multiple things happening one after another. Let's deconstruct it.

`querySelector()` is a top-level function provided by the `dart:html` library that gets an `Element` object from the DOM.

```
querySelector('#RipVanWinkle')!.text = 'Wake up, sleepy head!';
```

The argument to `querySelector()` is a string containing a CSS selector that identifies the object. Most commonly CSS selectors specify classes, identifiers, or attributes. We'll look at these in more detail later, when we add a CSS file to the mini app. In this case, RipVanWinkle is the unique ID for a paragraph element declared in the HTML file, and `#RipVanWinkle` specifies that ID.

Another useful function for getting elements from the DOM is `querySelectorAll()`, which returns multiple `Element` objects via a list of elements—`List<Element>`—all of which match the provided selector.

Handling nullable elements

Because DOM elements might be missing, the `querySelector()` function returns a nullable result, as indicated by the `?` in the function's return type (`Element?`). A `null` return value means that no element matches the specified CSS selector.

To use the `text` property of a returned `Element?`, the type must be promoted to `Element`. Because we wrote the HTML and know that the element is always present, we can use the null assertion operator (`!`) when referring to the element's properties:

```
querySelector('#RipVanWinkle')!.text = 'Wake up, sleepy head!';
```

If the element with the `#RipVanWinkle` ID isn't guaranteed to be present in the DOM, you can instead use the conditional member access operator (`?.`) to set `text` only if the returned element is not `null`:

```
querySelector('#RipVanWinkle')?.text = 'Wake up, sleepy head!';
```

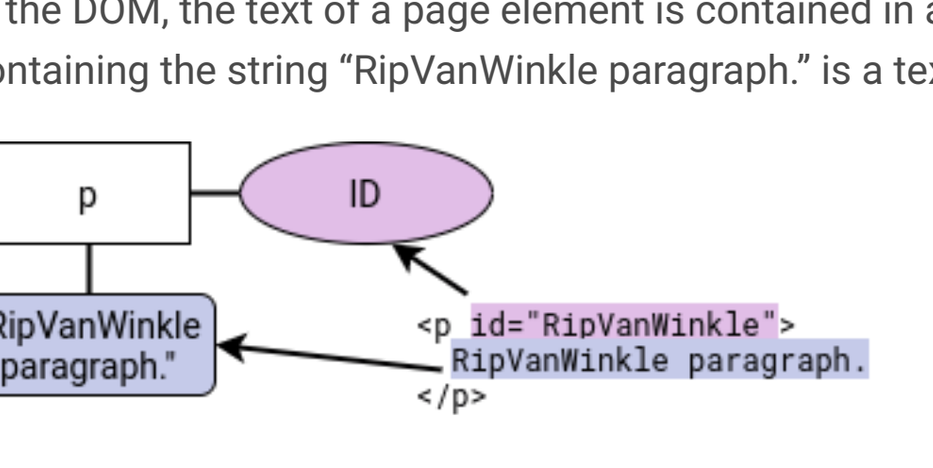
If you plan to access the element multiple times and know its type, another option is to typecast the queried element to the expected type:

```
final paragraph = querySelector('#RipVanWinkle') as ParagraphElement;
paragraph.text = 'Wake up, sleepy head!';
```

To learn more about nullable types and null safety in general, see Sound null safety.

Setting the text of an Element

In the DOM, the text of a page element is contained in a child node, specifically, a text node. In the following diagram, the node containing the string "RipVanWinkle paragraph." is a text node.



More complex text, such as text with style changes or embedded links and images, would be represented with a subtree of text nodes and other objects.

In Dart, you can simply use the Element `text` property, which has a getter and setter that walk the subtree of nodes for you and extract or set their text:

```
querySelector('#RipVanWinkle')!.text = 'Wake up, sleepy head!';
```

However, if the text node has styles (and thus a subtree), getting text and then setting it immediately is likely to change the DOM, as a result of losing subtree information. Often, as with our RipVanWinkle example, this simplification has no adverse effects.

The assignment operator (`=`) sets the text of the `Element` returned by the `querySelector()` function to the string "Wake up, sleepy head!".

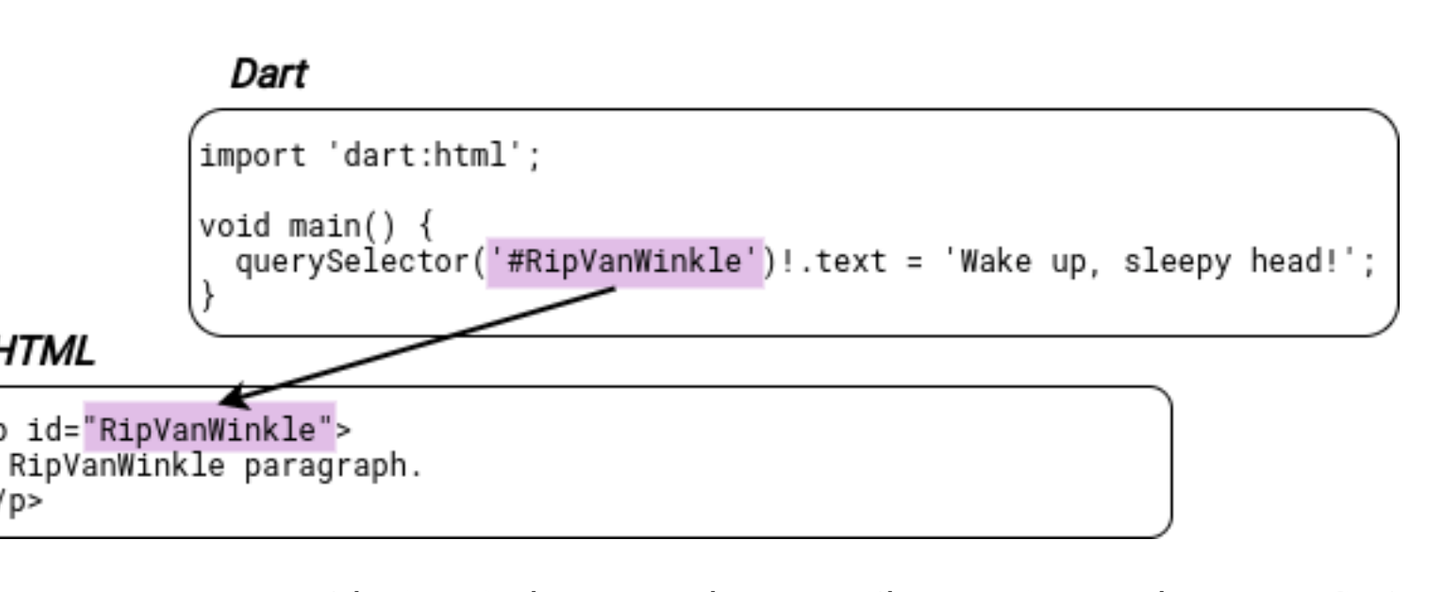
```
querySelector('#RipVanWinkle')!.text = 'Wake up, sleepy head!';
```

This causes the browser to immediately re-render the browser page containing this app, thus dynamically displaying the text on the browser page.

HTML and Dart connections

The Dart web app changed the text in the browser window dynamically at runtime. Of course, placing text on a browser page and doing nothing else could be accomplished with straight HTML. This little app only shows you how to make a connection from a Dart app to a browser page.

In DartPad, the only visible connection between the Dart code and the HTML code is the `RipVanWinkle` ID.



To run your app outside DartPad, you need to compile your Dart code to JavaScript. Use the `webdev build` command to compile your app to deployable JavaScript. Then you need to make another connection between the HTML and generated JavaScript: you must add a `<script>` tag to the HTML to tell the browser where to find the compiled Dart code.

Here's the full HTML code for this app, assuming that the Dart code is in a file named `main.dart`:

```
<!DOCTYPE html>

<html>
  <head>
    <title>A Minimalist App</title>
    <script defer src="main.dart.js"></script>
  </head>
  <body>
    <p id="RipVanWinkle">
      RipVanWinkle paragraph.
    </p>
  </body>
</html>
```

The `<script>` element specifies the location of the compiled Dart code.

Give the app some style with CSS

Most HTML uses cascading style sheets (CSS) to define *styles* that control the appearance of page elements. Let's customize the CSS for the mini app.

1. Click **CSS** at the upper left of DartPad. The view switches from Dart code to the (non-existent) CSS code.
2. Add the following CSS code:

```
#RipVanWinkle {
  font-size: 20px;
  font-family: 'Roboto', sans-serif;
  text-align: center;
  margin-top: 20px;
  background-color: SlateBlue;
  color: Yellow;
}
```

The display in the output pane immediately changes to reflect the new styles, which apply only to the page element that has the ID `RipVanWinkle`.

About CSS selectors

IDs, classes, and other information about elements are established in HTML. Your Dart code can use this information to get elements using a CSS selector—a pattern used to select matching elements in the DOM. CSS selectors allow the CSS, HTML, and Dart code to refer to the same objects. Commonly, a selector specifies an ID, an HTML element type, a class, or an attribute. Selectors can also be nested.

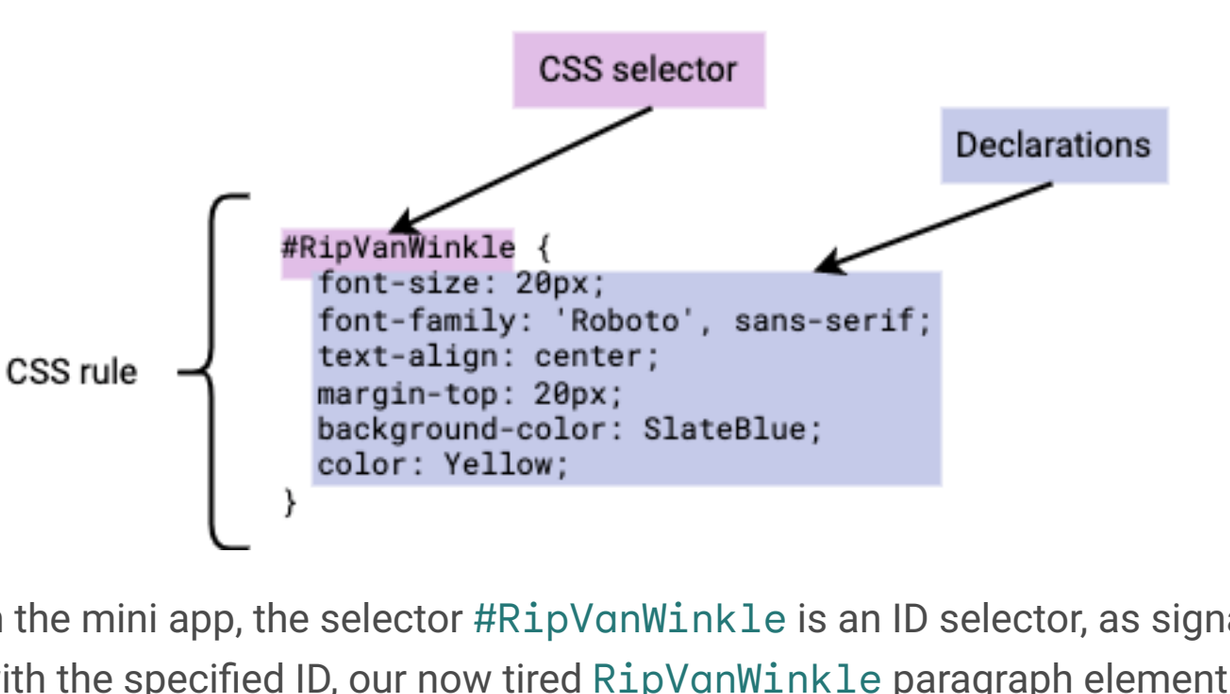
CSS selectors are important in Dart programs because you use them with `querySelector()` and `querySelectorAll()` to get matching elements from the DOM. Most often Dart programs use ID selectors with `querySelector()` and class selectors with `querySelectorAll()`.

Here are some examples of CSS selectors:

Selector type	Example	Description
ID selector	`#RipVanWinkle`	Matches a single, unique element
HTML element	`p`	Matches all paragraphs
HTML element	`h1`	Matches all level-one headers
Class	*`.classname`*	Matches all items with the class *classname*
Asterisk	`*`	Matches all elements
Attribute	`input[type="button"]`	Matches all button input elements

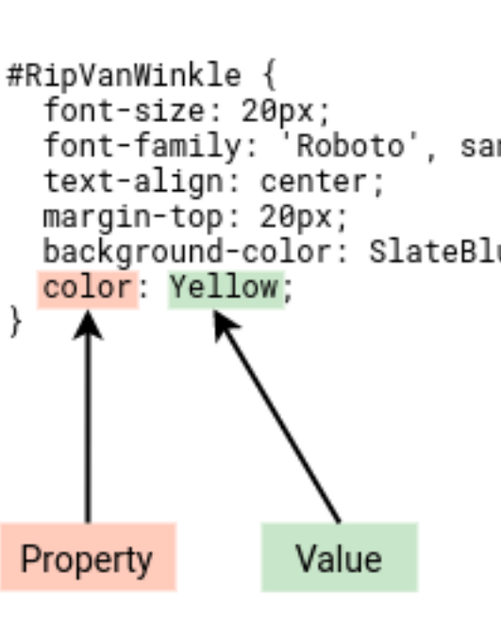
Tip: As you saw, the mini app used a CSS selector, the ID `RipVanWinkle`, even when there was no CSS file. You do not need a CSS file for a Dart program. Nor do you need a CSS file to use CSS selectors. CSS selectors are established in the HTML file and used by the Dart program to select matching elements.

Let's look at the CSS code for the mini app. The CSS file for the mini app has one CSS rule in it. A CSS rule has two main parts: a selector and a set of declarations.



In the mini app, the selector `#RipVanWinkle` is an ID selector, as signaled by the hashtag (`#`); it matches a single, unique element with the specified ID, our now tired `RipVanWinkle` paragraph element. `RipVanWinkle` is the ID in the HTML file. It is referred to in the CSS file and in the Dart code using a hashtag(`#`). Classnames are specified in the HTML file without a period (`.`) and referred to in the CSS file and in Dart code with a period (`.`).

Between the curly brackets of a CSS rule is a list of declarations, each of which ends in a semi-colon (`;`). Each declaration specifies a property and its value. Together the set of declarations define the *style sheet* for all matching elements. The style sheet is used to set the appearance of the matching element(s) on the web page.



The CSS rule for the `RipVanWinkle` paragraph specifies several properties; for example, it sets the text color to Yellow.

Other resources

- The language tour provides thorough coverage of the Dart language.
- The Dart tools page lists IDEs and editors that have Dart plugins.
- The DartPad documentation walks through the basics of using the DartPad web editor.

What next?

The next tutorial, Add elements to the DOM, shows you how to dynamically change the HTML page by adding elements to the DOM.

Add elements to the DOM ›