Join Dart and Flutter at Google I/O 2022 live from Shoreline Amphitheatre and online May 11-12. Register now
The Flutter and Dart teams are hiring. Learn more

‹ Connect Dart and HTML | Remove DOM elements ›

Add elements to the DOM

Contents

What's the point?

- In Dart, page elements are of type `Element`.
- An `Element` knows its parent.
- An `Element` keeps its children in a `List<Element>`.
- Change the DOM by adding or removing children of elements.
- Respond to user input with an `EventListener`.

Note: This page uses embedded DartPads to display runnable examples. If you see empty boxes instead of DartPads, go to the DartPad troubleshooting page.

As you learned in the previous tutorial, the DOM represents the structure of a web page document using a simple tree structure. Each node in the tree represents an item on the page. Each node in the tree keeps track of both its parent and its children. In Dart, the Node class contains the methods and properties that implement a node's tree functionality.

HTML page elements are one kind of node that can be in the DOM tree. They have a rectangular area on the page and can receive events. Examples of elements include heading elements, paragraph elements, table elements, button elements, and so on.

In Dart, elements are implemented by the `Element` class, which is a subclass of Node. Because the nodes you care about most are usually elements, this tutorial focuses on `Element`, rather than on Node.

Running the Todo app

In this tutorial, you will be working with a sample web app that is a partial implementation of a todo list. This program dynamically changes the DOM, and therefore the web page, by adding elements to the DOM tree.

Try it! Click **Run** to start the web app. Then type in the app's text field, and press return. The app adds an item to the list. Enter a few items into the input field.

Dart | HTML | CSS — Install SDK | Format | Reset | Run

```
body {
  font-family: 'Roboto', sans-serif;
  background-color: WhiteSmoke;
  margin: 15px;
  color: black;
}

h2 {
  color: black;
}

#to-do-input {
  font-family: 'Roboto', sans-serif;
  font-size: 14px;
  font-weight: normal;
  padding: 5px 0px 5px 5px;
```

UI Output

Console — no issues

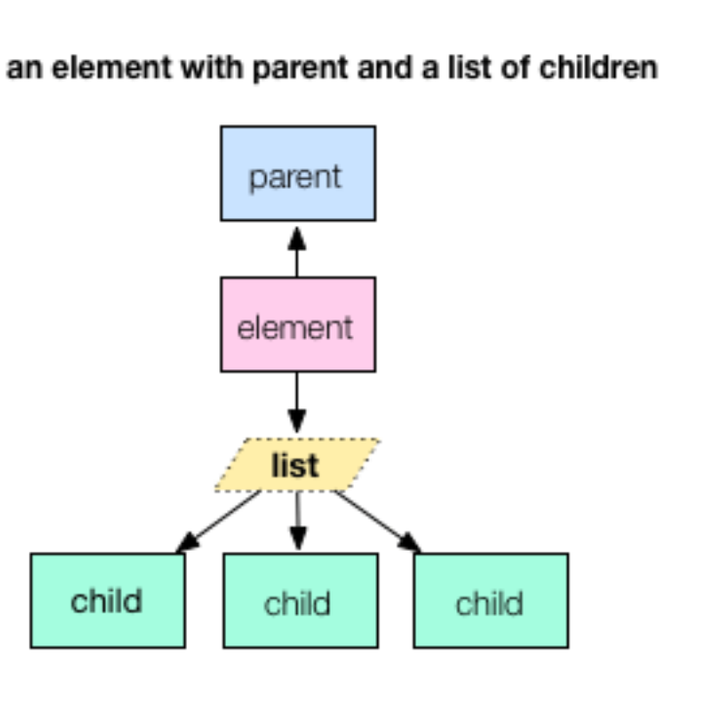
This is the beginning of an app to manage a list of things to do. Right now, this app is for procrastinators only because the program can only add items to your to do list but not remove them.

About parent and child Elements in Dart

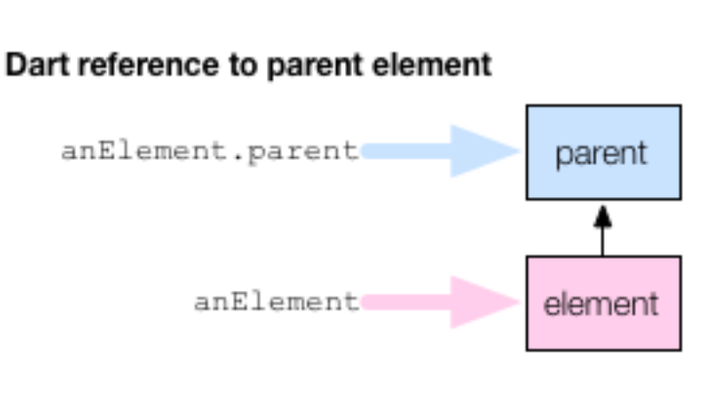
The Node class implements the basic treeing behavior for nodes in the Dart DOM. The `Element` class is a subclass of Node that implements the behavior specific to page element nodes. For example, an element knows the width and height of its enclosing rectangle on the page and it can receive events.

You can manipulate the DOM tree by adding and deleting nodes. However, many Dart apps are concerned only with page elements. So for convenience and code simplicity, the `Element` class implements API for interacting with a subset of the DOM that includes only the nodes that are elements. You can work with a virtual tree of elements rather than the more complex tree of Node objects. This tutorial shows you how to manipulate the DOM through the `Element` class.

An `Element` object has a parent `Element` and maintains references to its children elements in a list.



An `Element` object has at most one parent `Element`. An `Element` object's parent is final and cannot be changed. So you cannot move an `Element` by changing its parent. Get an `Element`'s parent with the getter `parent`. For example, if you have an `Element` with the name `anElement` you would refer to its parent element with `anElement.parent`.

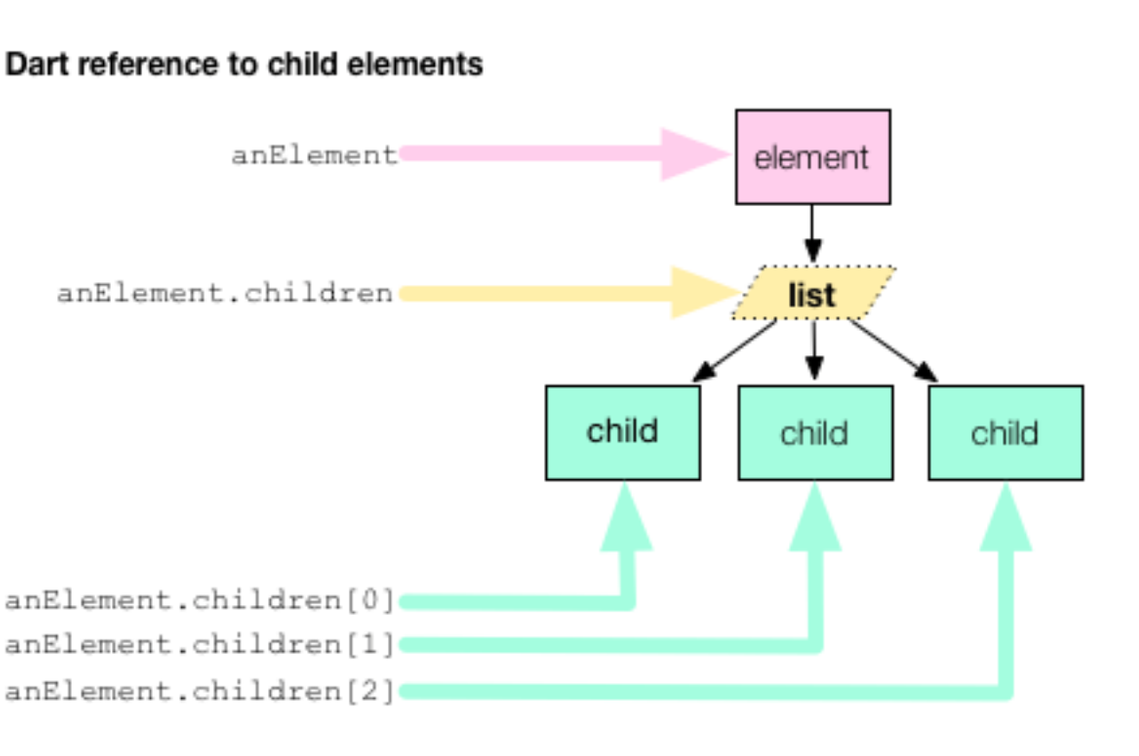


An `Element` object maintains references to its child elements in a list. `List` is a class in the dart:core library that implements an indexable collection with a length. A list can be of fixed size or extendable.

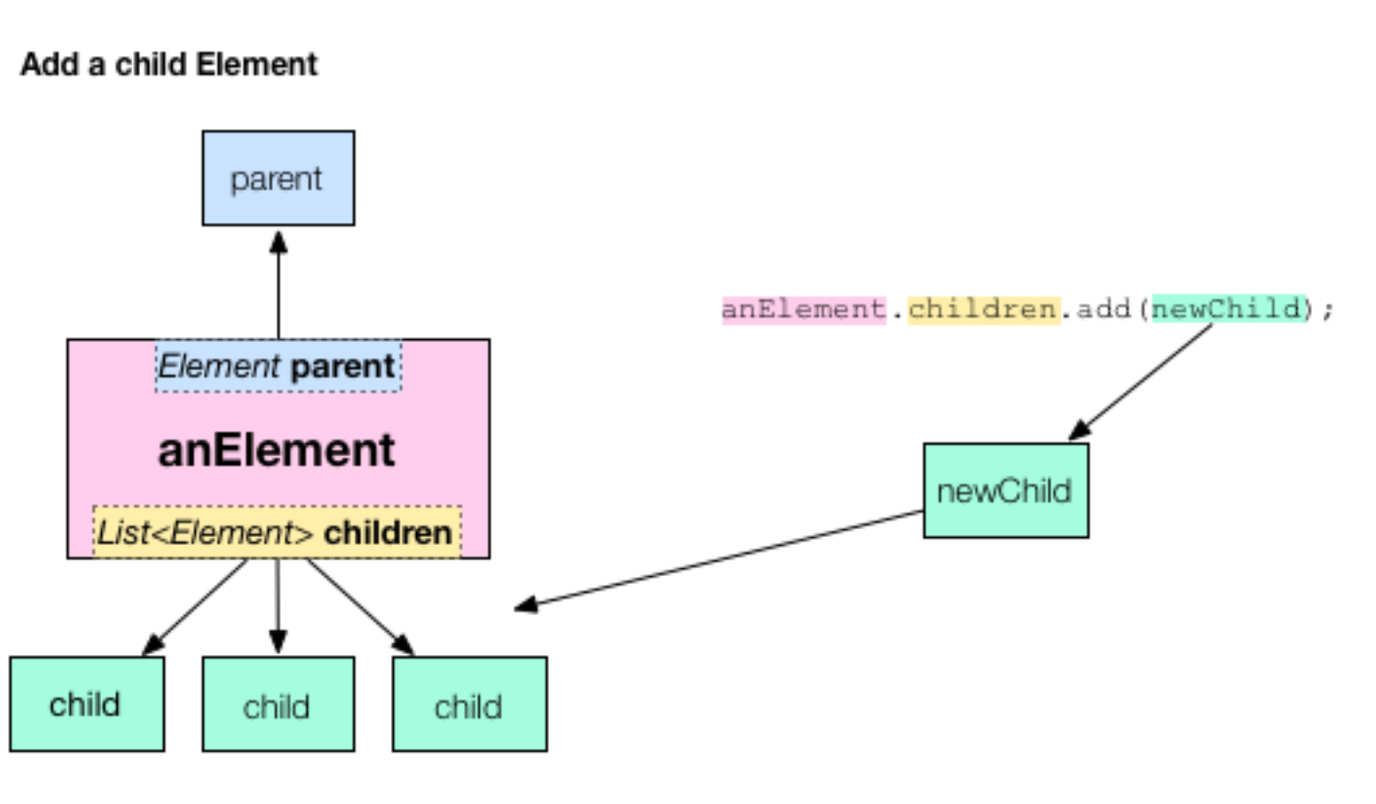
`List` is an example of a *generic* (or *parameterized*) type—a type that can declare formal type parameters. This means that a list can be declared to contain only objects of a particular type. For example:

List declaration	List description
`List<String>`	List of strings
`List<int>`	List of integers
`List<Element>`	List of elements

An `Element` maintains references to its child element in a `List<Element>`, which your Dart code can refer to with the getter `children`. The `List` class has various methods and operators whereby you can refer to each child element individually, iterate over the list, and add and remove elements.



You can change the tree structure by adding children to and removing children from an element's list of children.

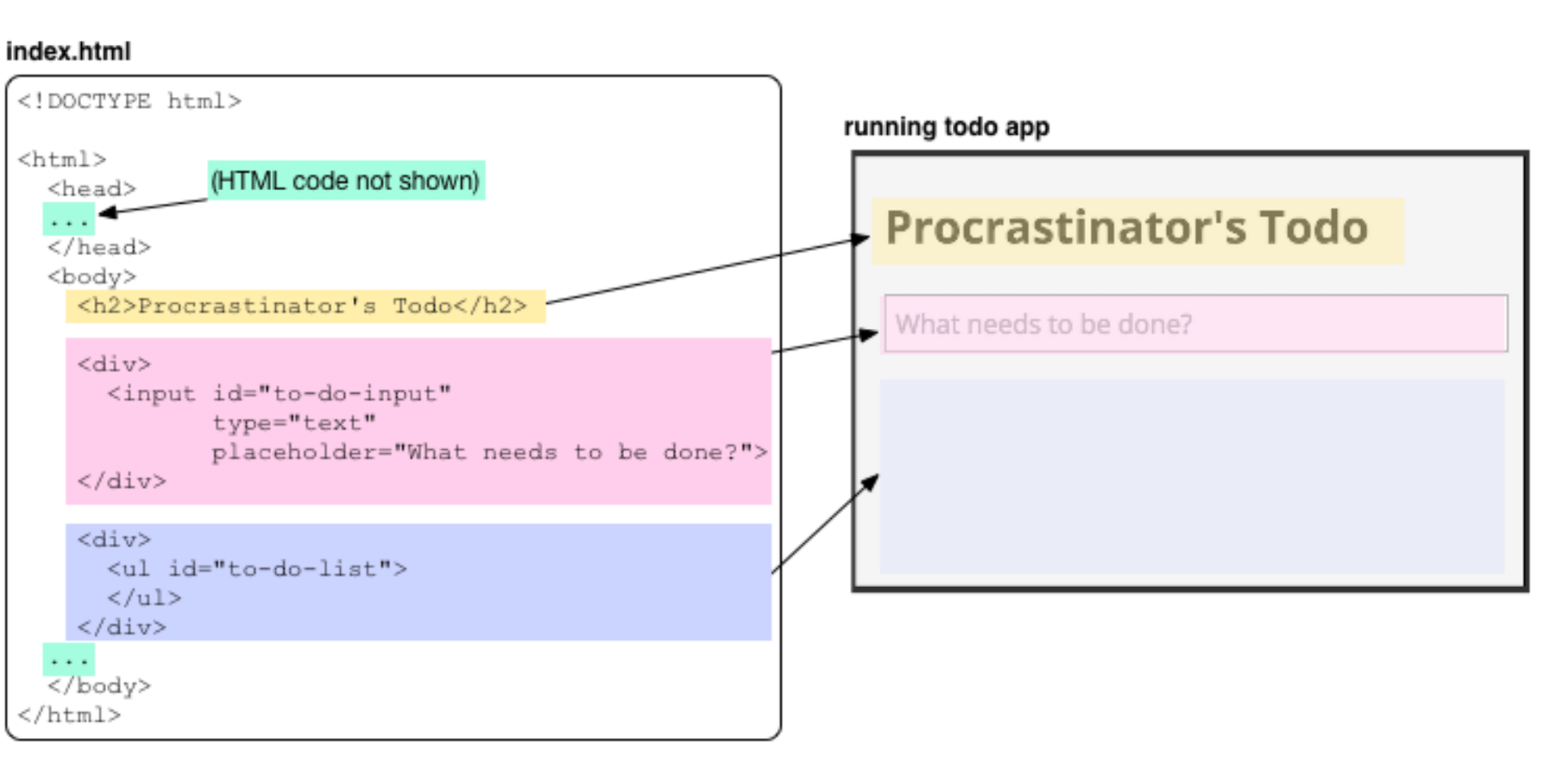


When you change an `Element` or its child elements in your Dart program, you change the DOM and therefore the web page. The browser re-renders the page automatically.

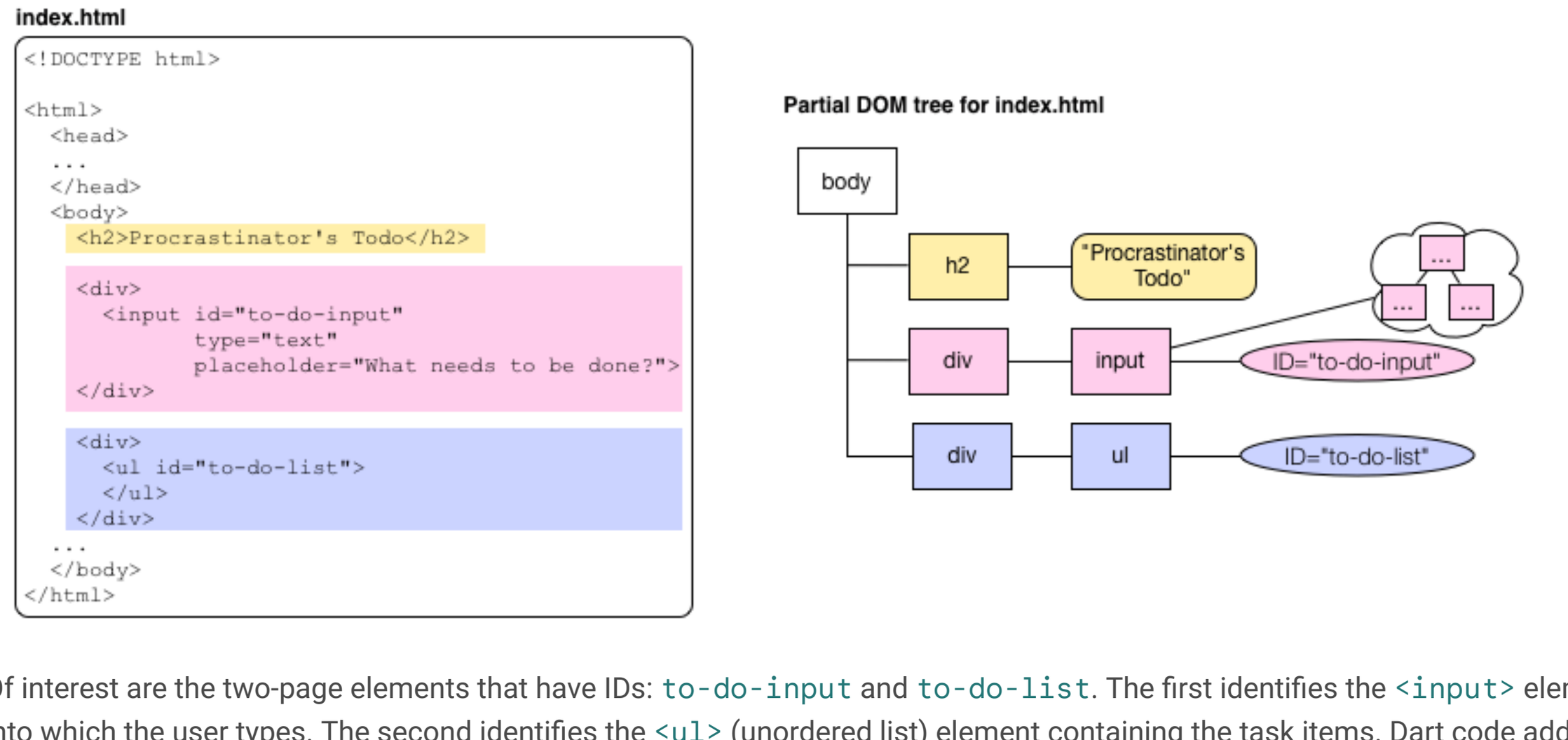
Setting up the page in HTML

Let's take a look at the todo app to see how it dynamically adds an element to the DOM and displays a new item in the list of things to do.

The HTML code for the todo app sets up the initial HTML page, and thereby the initial DOM tree. You could get the same results using Dart code, but it's usually better to define the primary page elements in HTML code (easier to read, quicker to load).



The following diagram shows a partial DOM tree for the todo app.



Of interest are the two-page elements that have IDs: `to-do-input` and `to-do-list`. The first identifies the `<input>` element into which the user types. The second identifies the `<ul>` (unordered list) element containing the task items. Dart code adds elements to this list whenever the user enters text into the input element.

Getting an element from the DOM

The following diagram shows the Dart code for the todo app.

The top-level variables are initialized using the `dart:html` library's top-level `querySelector()` function to get the interesting elements from the DOM. Because calling `querySelector()` isn't free, if a program refers to an element more than once it should stash a reference to the element if possible.

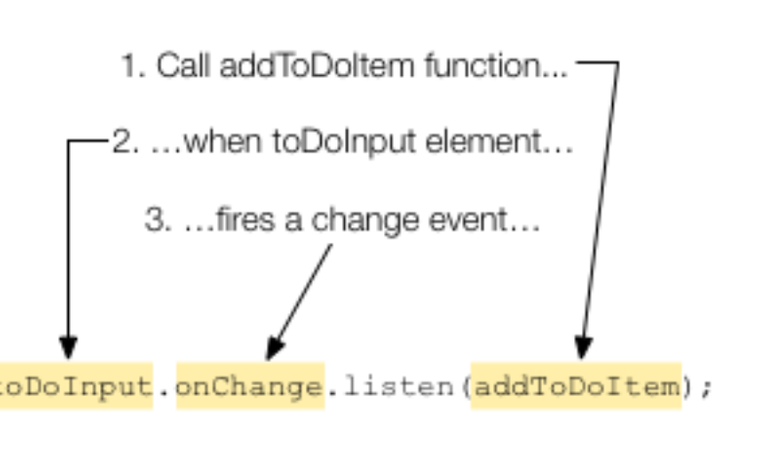
This program stashes a reference to the input element in a top-level variable called `toDoInput`. The unordered list is in the top-level variable `toDoList`.

Note the types of these variables: `InputElement` and `UListElement`. These are both subclasses of `Element`. The `dart:html` library has dozens of `Element` subclasses, many of which correspond to certain HTML tags. This program uses three:

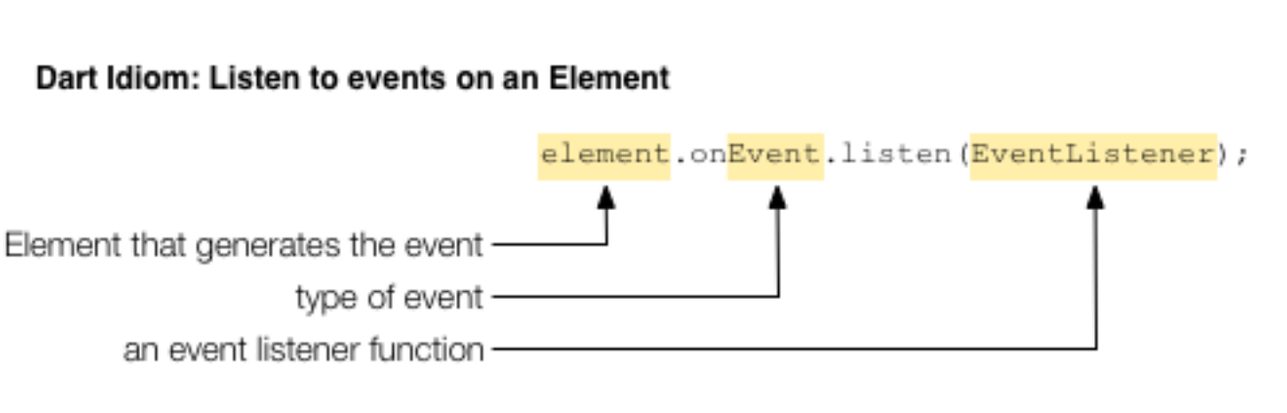
HTML tag	Dart class
`<input>`	`InputElement`
`<ul>`	`UListElement`
`<li>`	`LIElement`

Registering an event handler

When a user enters text into the input field, a *change* event fires, indicating that the value in the input field has just changed. The todo app defines a function, `addToDoItem()`, that can handle these change events. The following code connects `addToDoItem()` to the input field:



Rather than dissect this busy line of code, think of it as a Dart idiom for adding an event handler to an `Element` object.



A change event is just one of many types of events that an input element can generate. For example, you can use `click` to handle mouse clicks, or `keyDown` for when the user presses a key on the keyboard.

About EventListener functions

The argument passed to the `listen()` method is a *callback function* of type `EventListener`. `EventListener` is a typedef defined in the `dart:html` library as follows:

```
typedef void EventListener(Event event)
```

As you can see, an EventListener returns no value (`void`) and takes an `Event` object as an argument. Any function with this signature is an `EventListener`. Based on its signature, the `addToDoItem()` function is an `EventListener`.

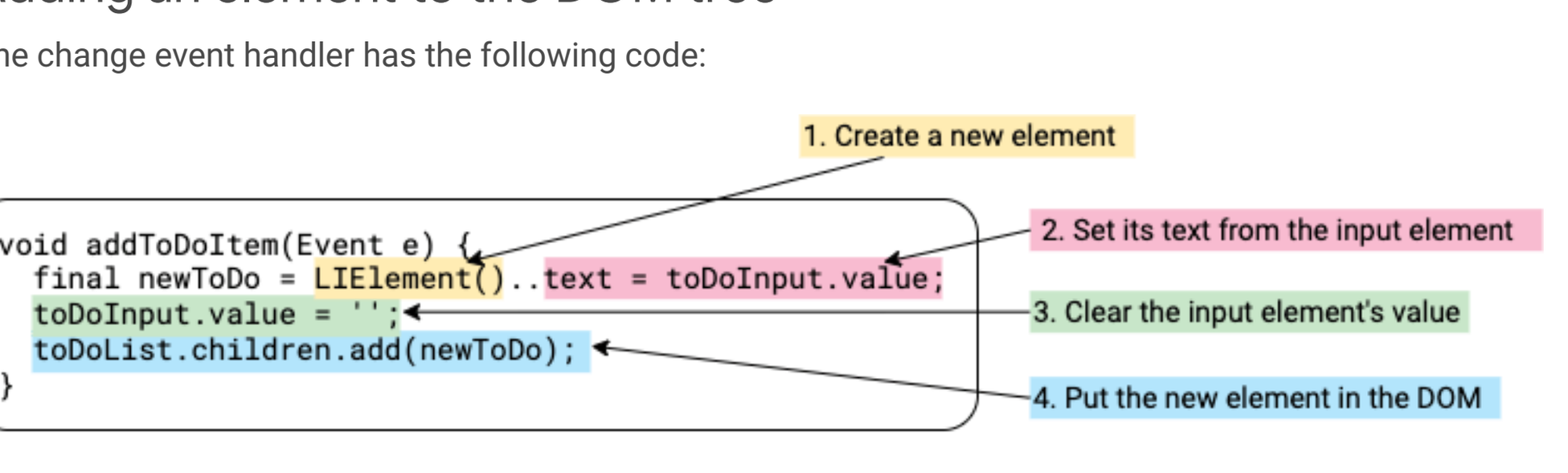
```
void addToDoItem(Event e) { ... }
```

The `Event` object passed into an `EventListener` function carries information about the `Event` that occurred. For example, the `Event` object knows which `Element` fired the event, and when. For location-specific events such as mouse clicks, the `Event` object also knows where the event occurred.

The `addToDoItem()` function ignores the Event object passed to it.

Adding an element to the DOM tree

The change event handler has the following code:

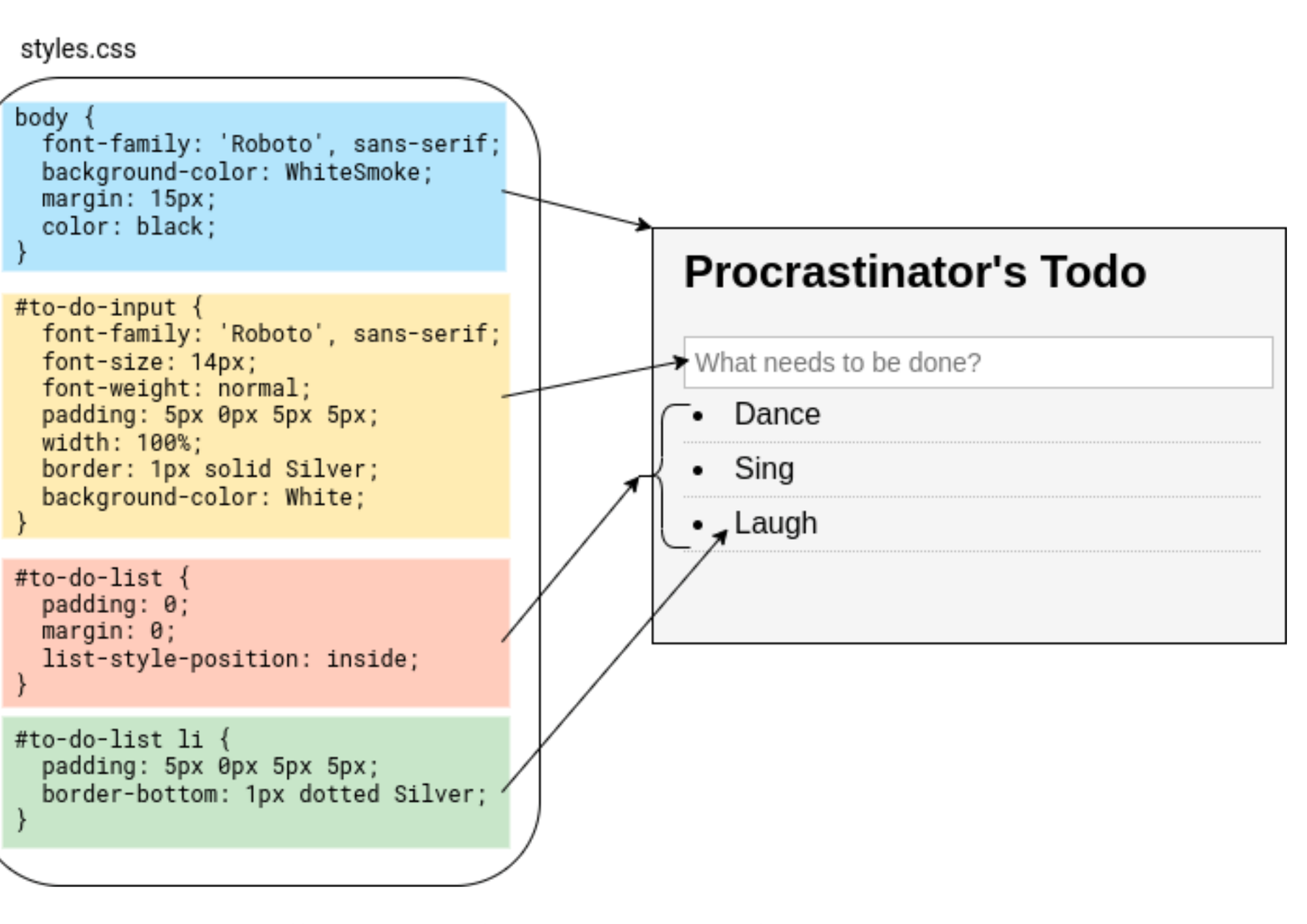


The final line of code is where the DOM gets changed.

An `Element` object keeps references to all of its children in a list called `children`. By adding and removing elements to and from this list, the code changes the DOM. When the DOM changes, the browser re-renders the browser page. The effect, in our todo app, is that a new bullet item appears in the to do list.

Styling the page elements

Let's take a look at the CSS file for this app.



This code uses three different kinds of CSS selectors. The first is an HTML element selector that matches the `<body>` element and sets some basic style attributes, such as the background color, for the entire page. Next in the file are two ID selectors: `#to-do-input` controls the appearance of the input field and `#to-do-list` sets the appearance of the unordered list element in general. The elements in the list are controlled by the final rule, which uses both an ID selector and an HTML selector. This rule matches all `<li>` elements in the element with the ID `to-do-list`, thus styling each item in the to do list.

Moving elements within the DOM tree

The Anagram app shows how to move an element within the DOM.

Try it! Click **Run** to start the web app. Then form a word by clicking the app's letter tiles.

Dart | HTML | CSS — Install SDK | Format | Reset | Run

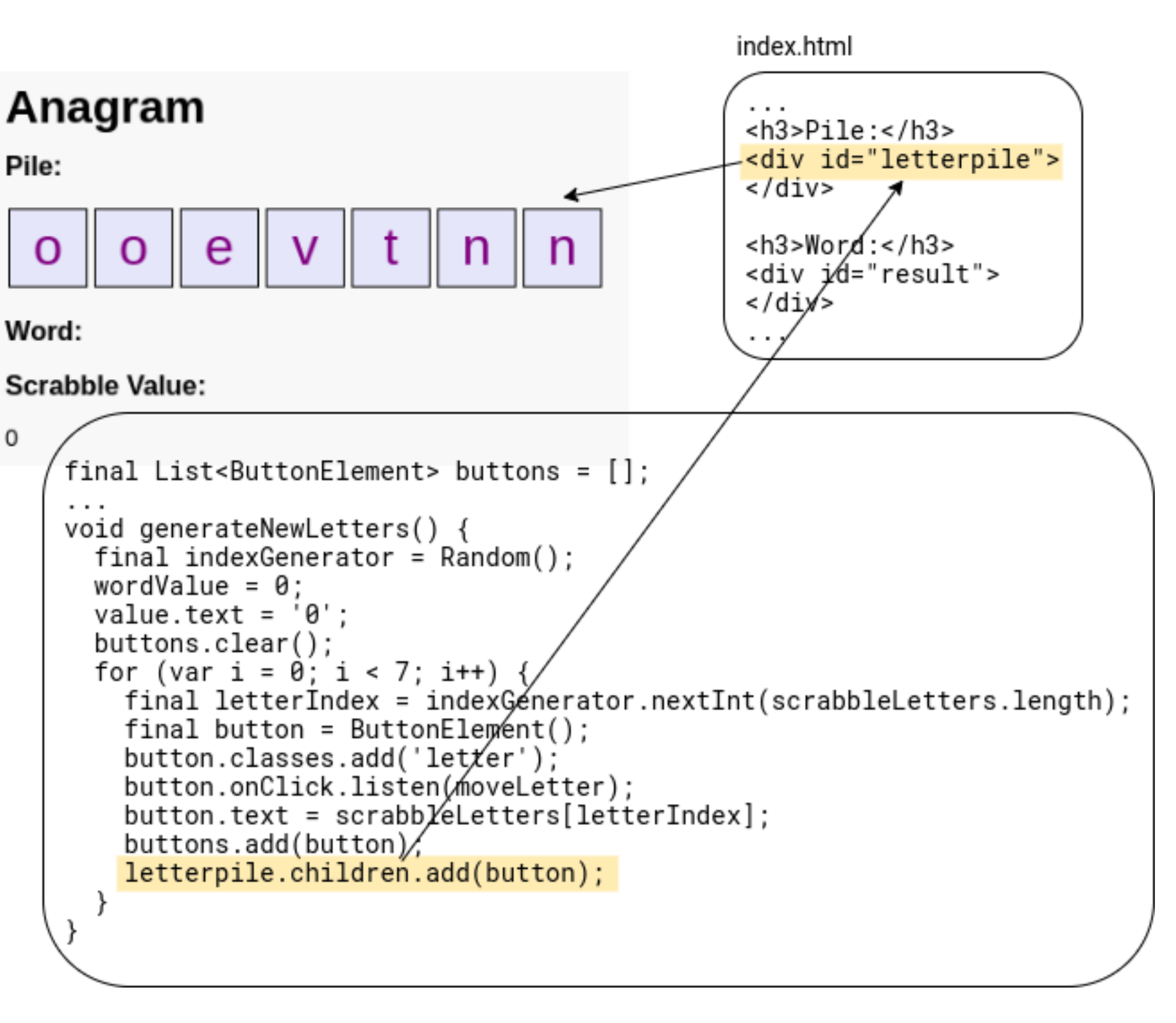
```
import 'dart:html';
import 'dart:math';

// Should remove tiles from here when they are selected
// otherwise the ratio is off.
const String scrabbleLetters =
    'aaaaaaaaabbccddddeeeeeeeeeeeeffggghhiiiiiiiiijkllllmmnnnnnnoooooooo
const Map<String, int> scrabbleValues = {
  'a': 1, 'e': 1, 'i': 1, 'l': 1, 'n': 1,
  'o': 1, 'r': 1, 's': 1, 't': 1, 'u': 1,
  'd': 2, 'g': 2, 'b': 3, 'c': 3, 'm': 3,
  'p': 3, 'f': 4, 'h': 4, 'v': 4, 'w': 4,
  'y': 4, 'k': 5, 'j': 8, 'x': 8, 'q': 10,
  'z': 10, '*': 0
};
```

UI Output

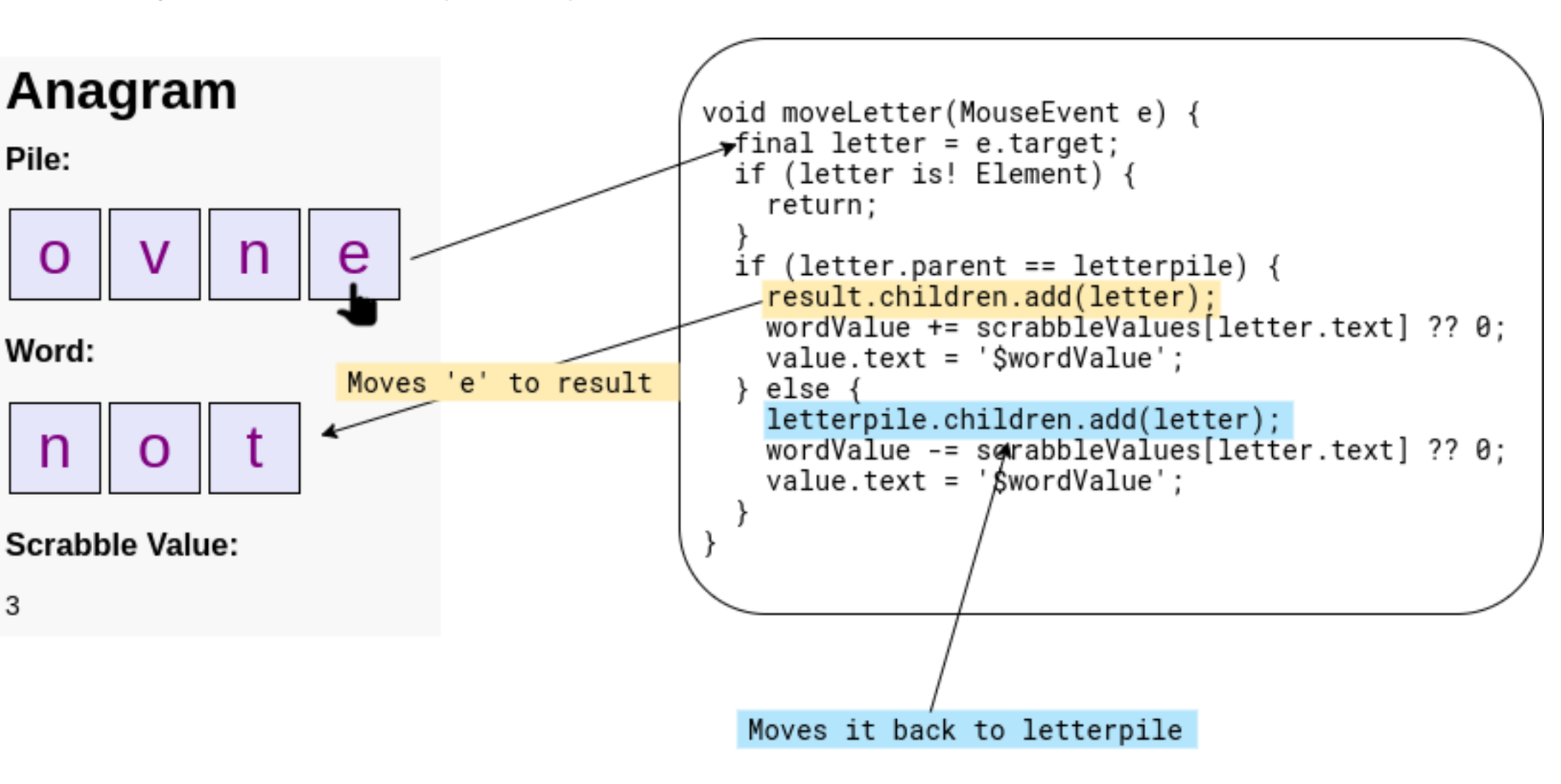
Console — no issues

When the program starts, it creates one button element for each of seven randomly selected letters. The program adds each button to a DOM element—a simple `<div>` element identified by the CSS selector `letterpile`—with a call to `letterpile.children.add()`.



Each button element in the letter pile has a mouse click handler called `moveLetter()`. If the button is in the letter pile, the mouse click handler moves the button to the end of the word. If the button is in the word, the mouse click handler moves the button back to the letter pile.

To move the button from the letter pile to the word or back, the code simply adds the button to a DOM element that is different from the button's current parent. Because an element can have only one parent, adding the button to a different parent automatically removes it from its previous parent.



The `+=` operator is a compound assignment operator, which combines an operation (`+`) with an assignment.

The `scrabbleValues` variable is a Map—a data structure that contains key/value pairs. Use the square bracket syntax to retrieve a value by its key and the `length` property to get the number of pairs it contains.

What next?

The next tutorial, Remove DOM Elements, describes how to remove elements from the DOM and items off your todo list.

‹ Connect Dart and HTML | Remove DOM elements ›