`Renderer` to `Renderer2` migration

Migration Overview

The `Renderer` class has been marked as deprecated since Angular version 4. This section provides guidance on migrating from this deprecated API to the newer `Renderer2` API and what it means for your app.

Why should I migrate to Renderer2?

The deprecated `Renderer` class has been removed in version 9 of Angular, so it's necessary to migrate to a supported API. Using `Renderer2` is the recommended strategy because it supports a similar set of functionality to `Renderer`. The API surface is quite large (with 19 methods), but the schematic should simplify this process for your applications.

Is there action required on my end?

No. The schematic should handle most cases with the exception of `Renderer.animate()` and `Renderer.setDebugInfo()`, which already aren't supported.

What are the `__ngRendererX` methods? Why are they necessary?

Some methods either don't have exact equivalents in `Renderer2`, or they correspond to more than one expression. For example, both renderers have a `createElement()` method, but they're not equal because a call such as `renderer.createElement(parentNode, namespaceAndName)` in the `Renderer` corresponds to the following block of code in `Renderer2`:

```
const [namespace, name] = splitNamespace(namespaceAndName);
const el = renderer.createElement(name, namespace);
if (parentNode) {
  renderer.appendChild(parentNode, el);
}
return el;
```

Migration has to guarantee that the return values of functions and types of variables stay the same. To handle the majority of cases safely, the schematic declares helper functions at the bottom of the user's file. These helpers encapsulate your own logic and keep the replacements inside your code down to a single function call. Here's an example of how the `createElement()` migration looks:

Before:

```
public createAndAppendElement() {
  const el = this.renderer.createElement('span');
```

```
  el.textContent = 'hello world';
  return el;
}
```

After:

public createAndAppendElement() { const el = ___ngRendererCreateElement(this.renderer, this.element, 'span'); el.textContent = 'hello world'; return el; } // Generated code at the bottom of the file ___ngRendererCreateElement(renderer: any, parentNode: any, nameAndNamespace: any) { const [namespace, name] = ___ngRendererSplitNamespace(namespaceAndName); const el = renderer.createElement(name, namespace); if (parentNode) { renderer.appendChild(parentNode, el); } return el; } ___ngRendererSplitNamespace(nameAndNamespace: any) { // returns the split name and namespace }

When implementing these helper functions, the schematic ensures that they're only declared once per file and that their names are unique enough that there's a small chance of colliding with pre-existing functions in your code. The schematic also keeps their parameter types as `any` so that it doesn't have to insert extra logic that ensures that their values have the correct type.

I'm a library author. Should I run this migration?

Library authors should definitely use this migration to move away from the `Renderer`. Otherwise, the libraries won't work with applications built with version 9.

Full list of method migrations

The following table shows all methods that the migration maps from `Renderer` to `Renderer2`.

Renderer	Renderer2
`listen(renderElement, name, callback)`	`listen(renderElement, name, callback)`
`setElementProperty(renderElement, propertyName, propertyValue)`	`setProperty(renderElement, propertyName, propertyValue)`
`setText(renderNode, text)`	`setValue(renderNode, text)`
`listenGlobal(target, name, callback)`	`listen(target, name, callback)`
`selectRootElement(selectorOrNode, debugInfo?)`	`selectRootElement(selectorOrNode)`
`createElement(parentElement, name, debugInfo?)`	`appendChild(parentElement, createElement(name))`

Renderer	Renderer2
`setElementStyle(el, style, value?)`	`value == null ? removeStyle(el, style) : setStyle(el, style, value)`
`setElementAttribute(el, name, value?)`	`attributeValue == null ? removeAttribute(el, name) : setAttribute(el, name, value)`
`createText(parentElement, value, debugInfo?)`	`appendChild(parentElement, createText(value))`
`createTemplateAnchor(parentElement)`	`appendChild(parentElement, createComment(''))`
`setElementClass(renderElement, className, isAdd)`	`isAdd ? addClass(renderElement, className) : removeClass(renderElement, className)`
`projectNodes(parentElement, nodes)`	`for (let i = 0; i < nodes.length; i++) { appendChild(parentElement, nodes[i]); }`
`attachViewAfter(node, viewRootNodes)`	`const parentElement = parentNode(node); const nextSibling = nextSibling(node); for (let i = 0; i < viewRootNodes.length; i++) { insertBefore(parentElement, viewRootNodes[i], nextSibling);}`
`detachView(viewRootNodes)`	`for (let i = 0; i < viewRootNodes.length; i++) {const node = viewRootNodes[i]; const parentElement = parentNode(node); removeChild(parentElement, node);}`
`destroyView(hostElement, viewAllNodes)`	`for (let i = 0; i < viewAllNodes.length; i++) { destroyNode(viewAllNodes[i]); }`
`setBindingDebugInfo()`	This function is a noop in `Renderer2`.
`createViewRoot(hostElement)`	Should be replaced with a reference to `hostElement`
`invokeElementMethod(renderElement, methodName, args?)`	`(renderElement as any)[methodName].apply(renderElement, args);`

Renderer	Renderer2
`animate(element, startingStyles, keyframes, duration, delay, easing, previousPlayers?)`	Throws an error (same behavior as `Renderer.animate()`)