**Exercise 7.03: Identifying Memory Leaks**

Let's take a simple example and see how Chrome's developer tools can help us identify that we have a memory leak. Let's get started.

1. Open a new tab in Chrome, open **Menu** > **More Tools** > **Developer tools**, and go to the **Sources** tab:

Figure 7.17: An empty code snippet


Figure 7.17: An empty code snippet

1. Click **'+ New snippet'** at the top left of this window and add the following code to set up our memory leak: First of all, we are creating a new DOM element of the **div** type and assigning it to a variable called **imageWrappe**r:

*Note*

*Every modern browser has its own set of developer tools. However, for dealing with memory leaks, Chrome's developer tools are the most useful.*

1. Next, we declare three functions that simulate some user interaction with our page. The first function, **loadImages()**, creates 50 new image elements and adds a data property to them of a string containing **1,024 b** characters. This is analogous to loading images and adding them to **imageWrapper**:
2. The next function, **add()**, simply adds the **imageWrapper** element to the end of the document body, and our third function **remove()** will be used to remove that image wrapper.
3. Now, let's write one last function to tie these three together:
4. This last function simulates adding and removing the image wrapper multiple times, thus loading 50 new images to it each time. A real-world example may be that we have a gallery of images and the user is clicking the '**next**" button to load the next set of images (in our scenario, they'd be clicking it 1,000 times!). Clearly, this would be a rather poor implementation of such a feature, but our aim here is to demonstrate how memory leaks can occur in a simple way. Your final code snippet should look something like this:

Figure 7.18: Code snippet with all the process() code


Figure 7.18: Code snippet with all the process() code

1. Now, click the **run snippet** button to execute the code.
2. Next, we will go to the performance tab of the dev tools and have a look at what's happening in the memory heap as we add and remove our images. From the performance tab, click the record button to start recording a performance profile. While it's recording, in the console, call the **process()** function, say, three times, and then hit the **stop** button. You should now see a screen that looks something like this:

Figure 7.19: Memory heap of the performance tab of the developer tools


Figure 7.19: Memory heap of the performance tab of the developer tools

This is the performance profile we have just recorded. It can tell us lots of information about the system resources that are being used by our application over the duration of the recording. The blue line here shows us how much of JavaScript's memory heap is being used over time. In an application without memory leaks, we would expect the memory usage to repeatedly go up with memory allocation and back down again to a base level with memory deallocation, giving us a **sawtooth graph**. In our case, though, it's only going in one direction – up. This is a sure sign that we have a memory leak in our code. The garbage collector is unable to determine that the **images** we've loaded are no longer needed after each call of **process()**, and so they stay in the memory heap. If you notice a real-life application becoming less performant over time, or using higher than expected system resources, then this is a good place to come to check for a memory leak.

**Exercise 7.04: Fixing the Memory Leak**

Now that we've established that we have a memory leak – as shown by the JavaScript memory heap graph in the preceding screenshot – our next task is to fix our code so that it no longer contains the leak. Given what you know of the garbage collector and the reason for our leak, try to implement a fix for the previous code so that the garbage collector can see when our objects are no longer needed. After each function call, record a performance profile to see whether the garbage collector is able to free up the memory. What you're looking for is that the blue heap-memory allocation line goes up with memory allocation, but then comes back down again at regular intervals, showing that the memory is being deallocated. This is a sign of a garbage collector being able to deallocate the memory after each execution of the **process()** function. Let's get started:

1. Write a function and add it to the existing three functions that are called in the **process**:

Here, we've added one more function that is called **resetImageWrapper()**, which resets the **imageWrapper** object to an empty **div** element and added the function to our **process()** function's **for…loop**. Now, each time a set of images is removed from the DOM, its reference in JavaScript is also removed, and it can be marked by the garbage collector for deletion.

1. The next step is to call this new function each time we process our images, so we will add it to our main **process()** function:
2. Once again, we'll run the performance profiler. Go to the dev tools, then to the **Performance** tab, and click the record button to begin a performance profile recording. Then, call the **process()** function a few times and take a look at the memory heap usage:

Figure 7.20: Memory heap usage


Figure 7.20: Memory heap usage

*Note*

*As you probably noticed, the problem with our code was that we were storing references to each image element, and, therefore, the contents of its data attribute, inside the***imageWrapper***variable. The simple solution to this problem is to reassign the***imageWrapper***variable each time we remove it from the DOM.*

This is a much healthier memory heap profile. After each of the three **process()** function calls, the garbage collector could see that the images were no longer referenced by either the DOM or by JavaScript, and the space they were allocated in memory was freed up and given back to the memory pool.