Using .on()

The most effective way to add event handlers to any dynamically rendered HTML content is to use the **.on()** method. The **.on()** method works a little differently than the jQuery commands you have practiced up to this point. So how does **.on()** work? We must attach the **.on()** function to something within a **selector**. **However**, what you pass into the selector is **not** what you want to attach the event to. What goes into the selector is the element that **contains** the relevant HTML element(s) you want to attach an event listener to. You can think of the element you select as the sphere of **detection** for the event you want to listen for.

Here's an example:

```
$(document).on('click', 'button', function(){alert('you clicked a button!')});
```

So, the 'sphere of detection' of this instance of **.on()** is paired with the selector for the **whole** document. That means that the following parameters of the function will be applied to **everything** contained in the **entire document**. This is the widest scope that you can use on a jQuery function. If you want your scope to be smaller, you can select a different element in the selector.

The **.on()** function takes up to **4** parameters and we will show you how they mesh with our example above:

. The **event**: Are you waiting for a click? A submit? Keydown? In this case, we are looking for a **click.**
. Data you wish to pass to the handler (this is not required and seldom used). We're not using it here!
. The **target**: i.e. the element you are trying to target. Here we are targeting a **button**. So now our search is narrowed down listening for a **button** to be clicked.
. The **function** you want to run: Just the stuff you want to attach! In our case, we are just going to alert to the screen that we clicked a button-- **every time a button is clicked**.

Every time the **event** you specified in the **first** item (1) happens, the function searches the document for the **target(s)** you specified in the **third** item (3). If the action was triggered by the correct element, the code you put in the **fourth** item (along with any information you passed from the optional **second** item) will be executed.

The really neat thing about this **.on()** function is that every time the event happens in the corresponding target, the relevant parts of the document get scanned in real time. Thus, any new element added **after page load** will be detected and the code will run. Pretty great, huh? Using this method, the code we need to accomplish the same goal as the previous tab would be the following:

```
$(document).ready(function(){
    $('button').click(function(){
        $('div').append('<h3>I am a dynamically generated h3 </h3>');
     });
    $(document).on('click', 'h3', function(){
        alert('You clicked me!');
    });
});
```

Scope

Since we first must assign the scope of where our **.on()** function will look, this invites a question: why not just always use **document**? Using **$(document).on()** will assure us that no item can accidentally pass through detection. Well, think about it like this: if we tell you that one of your friends has a message for you, but you must figure out which friend has it -- that might take a while to figure out. If, instead, we say: one of your friends who lives on a **particular street** in Mountain View has a message for you -- that would take a lot less time. Same concept with being more specific in your **selector** for the **.on()** method; the more searching of the document the **.on()** function has to do, the slower your code will run. Selecting **$(document)** for everything in an average website won't slow down your performance at all, but if you have tens of thousands of dynamic HTML elements that you are trying to add handlers to, using a more specific selector can make a huge difference between your website crashing or working. This will apply to anyone interested in **game development**, where thousands of operations per second are the norm.

Also, another great reason to judge the scope of your detection carefully is that you may not want all dynamic HTML elements to have the same functionality. Consider the following code:

```
<script>
 $(document).ready(function(){
  $('button').click(function(){
   $('div').append('<h3>I am a new content</h3>');
  });
  $(document).on('mouseover', 'h3', function(){
   $(this).css('color', 'pink');
  });
 });
</script>
<body>
 <button>Click me for new content!</button>
 <div class='a'>
  <h3>I am old content</h3>
 </div>
 <div class='b'>
  <h3>I am old content</h3>
 </div>
</body>
```

Note: for this function, 'mouseover' takes the place of 'hover'.

If you run the code, you will notice that both *divs* will have dynamic content generated when the button is clicked. When any *h3* on the page is hovered over, it will turn pink. Perhaps you want to change the functionality a little bit so that only the *divs* in the *class 'a'* div have this functionality. We can make this change quite easily using the selector correctly. Observe:

```
<script>
  $(document).ready(function(){
   $('button').click(function(){
    $('div').append('<h3>I am a new content</h3>');
   });
   $('div.a').on('mouseover', 'h3', function(){
    $(this).css('color', 'pink');
   });
  });
</script>
<body>
  <button>Click me</button>
  <div class="a">
   <h3>I am old content</h3>
  </div>
  <div class="b">
   <h3>I am old content</h3>
  </div>
</body>
```

Now you can see the usefulness of this technique!