The Real Life

First, let's have a look at an example of the *closures-just-doing-their-thing*-type. We will rely on closures here, in order to make our code work, but we won't do anything fancy with them.

Stay With Me

Okay, imagine that you have a bunch of external links on a page, and for some reason you want to give users who click on them some time to reconsider. Maybe your code already knows that they have entered something into a comment field which they haven't submitted yet, and navigating away would mean that their comment would be lost. This situation is often handled by showing a **`confirm`** dialog box when an **`onbeforeunload`** event is triggered, but maybe you think it's more user friendly to change the link to a countdown that follows the link only when it reaches 0 and can be cancelled by clicking.

Here's how you could write that:

RUN ME

```
<!DOCTYPE html>
<html lang="en-US">

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta name="Description"
    content="Give users the chance to cancel the location change after clicking a link" />
  <title>Cancellable Links</title>
  <link rel="stylesheet" type="text/css" href="/wp-content/themes/reallifejs/css/examples.css

  <script type="text/javascript">

    function makeLinkCancellable(link) {

      link.onclick = function (e) {

        var timeout, timer = 3;

        var updateTimer = function () {
          if (timer == 0) {
            link.onclick = null;
            link.innerHTML = 'Here we go ...';
            window.location.href = link.href;
          }
          else {
            link.innerHTML = 'Click again to cancel. Remaining time: ' + (timer--) + ' second
            timeout = setTimeout(updateTimer, 1000);
          }
        };

        var linkText = link.innerHTML;

        link.onclick = function () {
          clearTimeout(timeout);
          link.innerHTML = linkText;
          makeLinkCancellable(link);
          return false;
        };

        updateTimer();
```

Let's have a look what's going on here, closure-wise:

For every link, there's an **`onclick`** handler, which sits inside the function **`makeLinkCancellable`**, and thus forms a closure around it as soon as it's set up to handle clicks (those clicks will happen *after* **`makeLinkCancellable`** has finished). There's not much in there, apart from the click handler, so the only thing it retains access to is the **`link`** parameter. This comes in very handy, though, because the function handling the clicks does need to know which link has actually just been clicked.

Our closure isn't the only way to get to that link, though: inside the click handler, the **`this`** keyword will point to the clicked link, and the event object's **`target`**/**`srcElement`** will also point to it. None of those are as easy to use as that closure that's already in place, though; let's have a look:

1.) **`this`** is always a bit tricky — you have to be aware how a function that uses **`this`** has been called, because that context changes what **`this`** actually is. The anonymous click handler is being called as a result of an **`onclick`** event, so inside it **`this`** will actually refer to the clicked link. Then there's that **`updateTimer`** function, though, which is being called manually the first time, and via **`setTimeout`** every time after that. For the manual call, you can use **`updateTimer.call(this)`** instead of **`updateTimer()`**, so **`this`** will still be the link, but for the recursive timeout calls you can't — inside those, **`this`** will refer to the **`window`** object. One solution to that problem, which you can see quite often, is to put **`this`** into a variable at the beginning of the click handler (**`self`** is often used as variable name for this. It's a keyword itself, but a rather useless one, so there's no harm in overwriting it). That variable will remember the context for you, so it can be used inside the timeout handler. But, basically, that's just another closure doing the work for you, and we already have a perfectly good closure in place that's remembering the link, so let's not mess with **`this`** here.

2.) **`target`**/**`srcElement`** are properties of the event object, and that slash already gives away the problem: there are browser differences, and those are never fun to work around of. First, you have to make sure to even get the event object into the handler, because older versions of Internet Explorer won't pass it as a parameter, but set it as a global variable. Then you have to check out whether the event object provides you with a **`target`** or a **`srcElement`** property, and use the right one accordingly. And if that wouldn't be enough trouble already, there's an additional complication: **`target`**/**`srcElement`** will point to the actual thing that has been clicked, so if there's not just text inside that link but, for instance, an **`<img>`** or a **`<span>`**, then that property will point to that, and not to the actual link.

So, it turns out our closure is the easiest way to get to the clicked link.

After a link has been clicked, two things happen: a) another click handler is set up, so when the link is clicked again, it can be reverted to the way it was originally. Everything that has just been said applies here too. And b), **`updateTimer()`** starts calling itself recursively, using a timeout. This sets up another closure on each run, which is used to access the timing information, so it will know how far the countdown is along, and whether it has reached 0.

Another thing that's noteworthy is that if you click two links, A and B, creating two countdowns, and countdown A reaches 0 before countdown B, it's still possible that you will actually be taken to the URL of link *B*, if site A takes a long time to respond to your request. In this case here, I don't particularly care, but it's imperative that you are aware of possible race conditions like this whenever you are dealing with delayed code.

Alright, so, all in all, nothing spectacular happening here, but you'd be hard-pressed to write this sort of functionality without using any closures.

Next, let's have a look at the module pattern, which, as we have already seen, can be used to create objects that provide a clean public interface, while hiding away the implementation details inside a closure.

Switchboard

I already mentioned how one might go about writing an animated page navigation (like the one on realLifeJS), so let's see how this might actually look like. I've used jQuery here, to make the whole thing a bit less verbose.

Make sure to click into the iframe before trying the cursor key navigation:

RUN ME

```
<!DOCTYPE html>
<html lang="en-US">

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <meta name="Description"
    content="Move through pages by clicking or left/right cursor key action" />
  <title>Animated Navigation</title>
  <link rel="stylesheet" type="text/css" href="/wp-content/themes/reallifejs/css/examples.css
  <script type="text/javascript"
    src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.4/jquery.min.js"></script>

  <script type="text/javascript">

  $(function () {

    // The navigation object. Provides an interface to change which page
    // is currently active, and triggers an event when the active page changes.
    var Navigation = (function () {

      var $pages = $('#pages>div');
      var totalPages = $pages.length;
      var activePage = $pages.length - 1;

      var goTo = function (i) {
        if (i < 0 || i >= totalPages) return;
        activePage = i;
        $pages.eq(i).trigger('onNavigate');
      };

      return {
        goTo: goTo,
        next: function () {
          goTo(activePage + 1);
        },
        previous: function () {
          goTo(activePage - 1);
        },
        getActivePage: function () {
          return activePage;
```

I've already shown you in the last part how closures are fundamental to the module pattern, so I'll concentrate now on explaining how to actually split up different responsibilities into different modules, and how to let those modules communicate, if they have to, without getting completely entagled with each other.

First, what actually needs to be done to make this animation thing work? We will have to set up mouse and keyboard events to enable user interaction, and on each of those interactions several things have to happen: We have to check whether the interaction actually results in a page change (e.g. if you're on the last page, you can't go any further), and if it does, the pages have to be animated, and somewhere the information which page is currently active needs to be stored, so going to the next/previous page using the cursor keys will work.

Second, and that's the interesting part, how do you figure out how those responsibilities should be distributed? Let's start at the beginning: Setting up the mouse and keyboard events is just a piece of code that runs only once and will never be called again, so let's keep that out of any module, and just put it at the end of the code. If you're writing the script from scratch, that's where you should start, because it gives you a clear idea what public methods you will need:

```
// Set up cursor key interaction
$(document).keydown(function (e) {
  if (e.keyCode == 39) {
    next();
  }
  else if (e.keyCode == 37) {
    previous();
  }
});

// Set up click-on-pages interaction
$('#pages>div').click(function () {
  goTo($(this).index());
});
```

So, obviously, you will need a **`next`** and a **`previous`** method for keyboard navigation, and a **`goTo`** method for clicking on a page, which accepts an integer denoting the page index. So far, so good. Let's write a module that provides those public methods and keeps track of the active page for us. There's no use jumping right on the animation part, since animating the pages is moot as long as you don't keep track of which page is active at any time.

```
var Navigation = (function () {

  // In order to check whether next/previous navigation is possible,
  // we need to know how many pages there are, so fetch them from the DOM
  var $pages = $('#pages>div');
  // Page numbers go from left to right, and the way the pages are stacked,
  // the rightmost page is active in the beginning
  var activePage = $pages.length - 1;

  // Check if the page actually exists, and if so, set it as active
  var goTo = function (i) {
    if (i < 0 || i >= $pages.length) return;
    activePage = i;
  };

  // Provide the public interface
  return {
    goTo: goTo,
    next: function () {
      goTo(activePage + 1);
    },
    previous: function () {
      goTo(activePage - 1);
    }
  };

}());
```

Great, so the user interactions are set up and result in the variable **`activePage`** being set to a number representing the index of the active page. As for the animation: We could just let the **`goTo`** function do the animating, since it already has everything it needs — it's being called on every user interaction, and it knows which page has just been requested. The thing is, though, that animating those pages isn't a one-liner — there's all sorts of stuff the animating routine needs to know, like the widths of those pages and the duration of the animation. Sure, we could just add those as local variables to the module, but we would be mixing things that have nothing to do with each other. If you find yourself defining the duration of an animation right next to defining how many pages your site has (— given that you're not doing that in some dedicated configuration object, of course), you should get suspicious — those two don't correlate in any way.

You might be asking yourself what the actual harm in mixing this little animation thing right into the **`Navigation`** object is. That's easily answered: The minor point is that your code will be harder to read if you're writing long objects that do lots of different things. That's not just for other people reading your code, but you yourself will have a hard time figuring out what's what if you come back to make a change after a month. The major point is a bit more abstract: All the things inside a module have unlimited access to each other, so if you change one of them, every other part of the module could potentially change how it works (or even break). That's a scary thought, isn't it — changing the duration of an animation and completely fucking up the active page tracking in the process.

Of course, you know that's not what's going to happen — the whole code is simple, and you have just written the damn thing, so you know full well that the page tracking part doesn't care about whatever that **`animationDuration`** variable holds. The thing is, though, that correlation is *possible*, as long as those two things sit in the same module, and I promise you that sooner or later, as your code becomes more complex, you *will* make a change to something that completely fucks up another thing which you thought was totally uncorrelated.

So, those are good reasons to put everything that has to do with the actual page animations into its own module:

```
var Animation = (function () {

  // Fetch the pages from the DOM
  var $pages = $('#pages>div');
  // The distance between two pages
  var spacing = 10;
  // The duration of the page animations
  var animationDuration = 200;
  // The width of the pages, which will be needed to figure out how far
  // the overlaying pages have to be slided away to make the requested one visible
  var width = $pages.width();

  // Rearrange the pages, so the requested one will be visible
  var moveTo = function (active) {
    $pages.each(function (i) {
      var left = (i <= active) ? i * spacing : width + (i - 2) * spacing;
      $(this).stop().animate({left: left}, animationDuration);
    });
  };

  // Do some initial setup
  $pages.each(function (i) {
    $(this).css({top: i * spacing, left: i * spacing});
  });

  // The moveTo function needs to be public
  return {
    moveTo: moveTo
  };

}());
```

Do you see that "initial setup" thing in there? There's only one CSS rule for all the pages, so in the beginning they will all be stacked right on top of each other; this piece of code moves the pages apart a bit, so you can actually see (and click on) all of them. Earlier I said we won't put the code that sets up the event handlers into any module, because it will only run once — this piece of code here runs only once, too, but it fits perfectly to what that module is about — moving around pages. There's already that **`spacing`** variable in the module that's needed for that initial setup, so it's pretty clear that that piece of code belongs in there.

Alright, we have just about everything we need now: User interactions will cause the active page index to be tracked, and we have all of the animation code — the only thing missing is for the animation code to be actually called. There are several ways to do this, but you have to be very careful here not to introduce dependencies that can bite you in the ass later:

This is very important: You have a bunch of things that cause page changes, like clicks or cursor key hits. Later on, you might decide to add some more causes, like hash changes in the URL or timing events (think automatic page slideshow). On the other hand, you have several effects of a page change. Actually, you have only one now, and that's the animation of those page elements, but later on, you might add others, like changing the document title, or putting the page change into the browse history, so the browser's back button will work for those page changes. That's a lot of stuff that needs to communicate with each other somehow; let's see how we can get this done:

The Bad Thing to do is call each of the effects from each of the causes. Beginners often do this. This would mean putting a call to the animating routine into all the interaction handlers. It should be clear why this is bad: if you add a cause and an effect, you have to put a call to the new effect into all the causes you already got, and you have to put a call to all the effects you got into the new cause. That's tedious and stupid. We already have a component in place that's being called by all of the causes (**`Navigation.goTo`**), so …

The Better But Still Bad Thing would be to put the calls to all the effects there. This would mean putting a call to the animating routine into **`Navigation.goTo`** (let's call this the central component, CC). If you add a cause and an effect, the new cause just has to make one function call (to the CC), which is independent of the actual effects — but you will still have to add a call to the new effect to the CC. That's not that much work, but it isn't great that the CC has to care about all of those effects. Adding a module shouldn't force you to change the contents of another module, and especially the CC should be allowed complete ignorance of what's going on around it. So, how can we accomplish this? Here we go:

The Good Thing to do is to make the CC trigger a custom event, which all of the effects can listen to and act accordingly. If you add a cause and an effect, the new cause just has to call the CC, and the new effect just has to listen to the CC. No other parts of the code need to be changed. You can see in the code example that implementing custom events in jQuery is very easy, and they are a great way to give two independent modules the chance to communicate with each other.

And that's the gist of it: Use some common sense on how to distribute responsibilities among your modules, and keep them as separate from each other as possible. If you notice that you're hesitating to add new functionality, because you are afraid that the changes you would have to make to your existing code might fuck something up, that probably means that you have lost control over what's responsible for what. Try not to let it get that far.

Alright, so much for closures. You should have a pretty good idea by now why they are a cool thing, and how to use them to your advantage. If you have any feedback regarding this article, just shoot me an email at feedback@reallifejs.com.