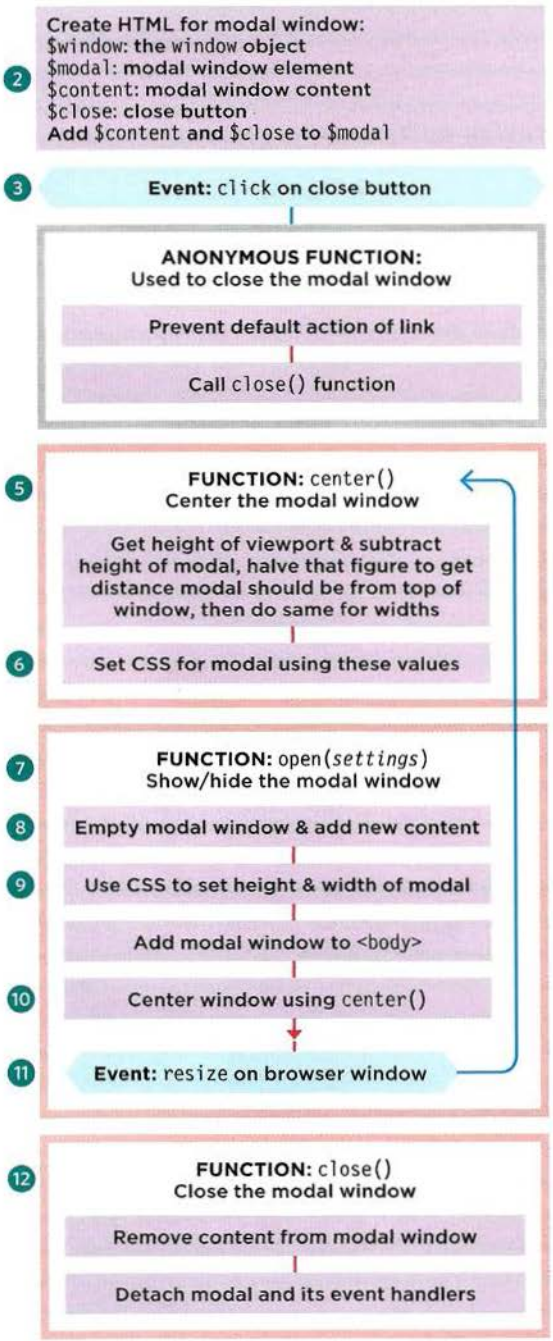
MODAL OBJECT



Below are the steps for creating the `modal` object. Its methods are used to create modal windows.

1. The `modal` object is declared. The methods of this object are created by an Immediately Involved Function Expression or IIFE (see p97). (This step is not shown in the flowchart.)
2. Store the current `window` object in a jQuery selection, then create the three HTML elements needed for the modal window. Assemble the modal window and store it in `$modal`.
3. Add an event handler to the close button which calls the `modal` object's `close()` method.
4. Following the `return` keyword, there is a code block in curly braces. It creates three public methods of the `modal` object. **Please note:** This step is not shown in the flowchart.
5. The `center()` method creates two variables:
i) `top`: takes the height of the browser window and subtracts the height of the modal window. This number is divided by two, giving the distance of the modal from the top of the browser window.
ii) `left`: takes the width of the browser window and subtracts the width of modal window. This number is divided by two, giving the distance of the modal from the left of the browser window.
6. The jQuery `.css()` method uses these variables to position the modal in the center of the page.
7. `open()` takes an object as a parameter; it is referred to as `settings` (the data for this object was shown on the previous page).
8. Any existing content is cleared from the modal, and the `content` property of the `settings` object is added to the HTML created in steps 1 and 2.
9. The width and height of the modal are set using values from the `settings` object. If none were given, `auto` is used. Then the modal is added to the page using the `appendTo()` method.
10. `center()` is used to center the modal window.
11. If the window is resized, call `center()` again.
12. `close()` empties the modal, detaches the HTML from the page, and removes any event handlers.

In the code below, the lines that are highlighted in green are considered **private**. These lines of code are only used within the object. (This code cannot be accessed directly from outside the object.)

When this script has been included in a page, the `center()`, `open()`, and `close()` methods in steps 5-12 are available on the `modal` object for other scripts to use. They are referred to as **public**.

JAVASCRIPT

c11/js/modal-window.js

```
① var modal = (function() {                          // Declare modal object
    var $window = $(window);
    var $modal = $('<div class="modal"/>');          // Create markup for modal
②   var $content = $('<div class="modal-content"/>');
    var $close = $('<button role="button" class="modal-close">close</button>');

    $modal.append($content, $close);                 // Add close button to modal

    $close.on('click', function(e) {                 // If user clicks on close
③     e.preventDefault();                            // Prevent link behavior
      modal.close();                                 // Close the modal window
    });

④   return {                                         // Add code to modal
      center: function() {                           // Define center() method
        // Calculate distance from top and left of window to center the modal
⑤       var top = Math.max($window.height() - $modal.outerHeight(), 0) / 2;
        var left = Math.max($window.width() - $modal.outerWidth(), 0) / 2;
        $modal.css({                                 // Set CSS for the modal
⑥         top: top + $window.scrollTop(),            // Center vertically
          left: left + $window.scrollLeft()          // Center horizontally
        });
      },
⑦     open: function(settings) {                     // Define open() method
⑧       $content.empty().append(settings.content);   // Set new content of modal

        $modal.css({                                 // Set modal dimensions
⑨         width: settings.width || 'auto',           // Set width
          height: settings.height || 'auto'          // Set height
        }).appendTo('body');                         // Add it to the page

⑩       modal.center();                              // Call center() method
⑪       $(window).on('resize', modal.center);        // Call it if window resized
      },
      close: function() {                            // Define close() method
        $content.empty();                            // Remove content from modal
⑫       $modal.detach();                             // Remove modal from page
        $(window).off('resize', modal.center);       // Remove event handler
      }
    };
  }());
```

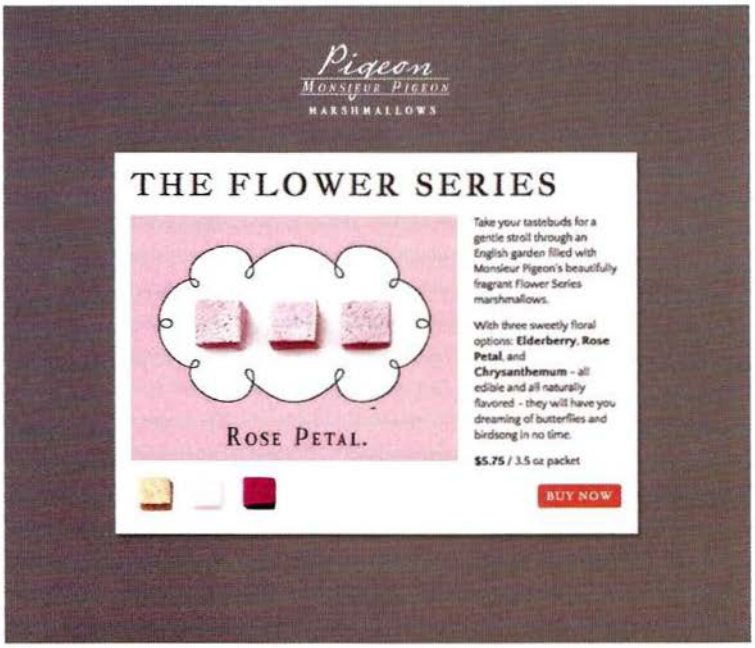
PHOTO VIEWER

The photo viewer is an example of an image gallery. When you click on a thumbnail, the main photograph is replaced with a new image.

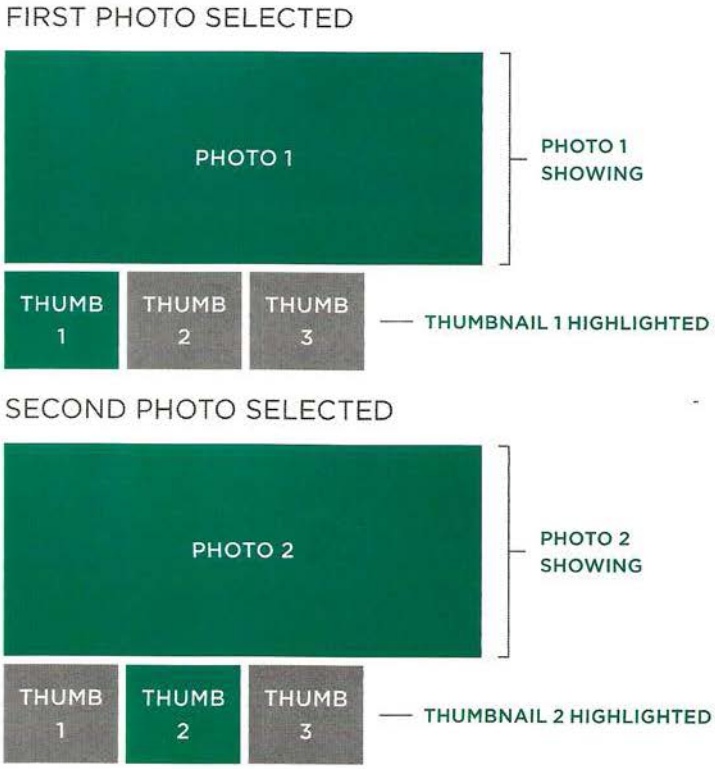
In this example, you can see one main image with three thumbnails underneath it.

The HTML for the photo viewer consists of:

- One large `<div>` element that will hold the main picture. The images that sit in the `<div>` are centered and scaled down if necessary to fit within the allocated area.
- A second `<div>` element that holds a set of thumbnails that show the other images you can view. These thumbnails sit inside links. The `href` attribute on those links point to the larger versions of their images.



Other gallery scripts include Galleria, Gallerific, and TN3Gallery.



When you click on a thumbnail, an event listener triggers an anonymous function that:

1. Looks at the value of the `href` attribute (which points to the large image)
2. Creates a new `<img>` element to hold that image
3. Makes it invisible
4. Adds it to the big `<div>` element

Once the image has loaded, a function called `crossfade()` is used to fade between the existing image and the new one that has been requested.

USING THE PHOTO VIEWER

In order to use the photo viewer, you create a `<div>` element to hold the main image. It is empty, and its `id` attribute has a value of `photo-viewer`.

The thumbnails sit in another `<div>`. Each one is in an `<a>` element with three attributes:

- `href` points to the larger version of the image
- `class` always has a value of `thumb` and the current main image has a value of `active`
- `title` describes the image (it will be used for `alt` text)

c11/photo-viewer.html

HTML

```
<div id="photo-viewer"></div>
<div id="thumbnails">
  <a href="img/photo-1.jpg" class="thumb active" title="Elderberry mallow">
    <img src="img/thumb-1.jpg" alt="Elderberry Marshmallow" /></a>
  <a href="img/photo-2.jpg" title="Rose Marshmallow" class="thumb">
    <img src="img/thumb-2.jpg" alt="Rose Marshmallow" /></a>
  <a href="img/photo-3.jpg" title="Chrysanthemum  Marshmallow" class="thumb">
    <img src="img/thumb-3.jpg" alt="Chrysanthemum Marshmallow" /></a>
</div>
```

The script comes before the closing `</body>` tag. As you will see, it simulates the user clicking on the first thumbnail.

The `<div>` that holds the main picture uses relative positioning. This removes the element from normal flow, so a `height` for the viewer must be specified.

While images are loading, a `class` of `is-loading` is added to them (it displays an animated loading gif). When the image has loaded, `is-loading` is removed.

If the images are larger than the viewer the `max-width` and `max-height` properties will scale them to fit. To center the image within the viewer a mix of CSS and JavaScript will be used. See p511 for detailed explanation.

c11/css/photo-viewer.css

CSS

```
#photo-viewer {
  position: relative;
  height: 300px;
  overflow: hidden;}

#photo-viewer.is-loading:after {
  content: url(images/load.gif);
  position: absolute;
  top: 0;
  right: 0;}

#photo-viewer img {
  position: absolute;
  max-width: 100%;
  max-height: 100%;
  top: 50%;
  left: 50%;}

a.active {
  opacity: 0.3;}
```

ASYNCHRONOUS LOADING & CACHING IMAGES

This script (shown on the next page) shows two interesting techniques:
1. Dealing with asynchronous loading of content
2. Creating a custom `cache` object

SHOWING THE RIGHT IMAGE WHEN LOADING IMAGES ASYNCHRONOUSLY

PROBLEM:
The larger images are only loaded into the page when the user clicks on a thumbnail, and the script waits for the image to fully load before displaying it.

Because larger images take longer to load, if a user clicks on two different images in quick succession:
1. The second image could load faster than the first one and be displayed in the browser.
2. It would be replaced by the *first* image the user clicked on when that image had loaded. This could make users think the wrong image has loaded.

SOLUTION:
When the user clicks on a thumbnail:
- A function-level variable called `src` stores the path to this image.
- A global variable called `request` is also updated with the path to this image.
- An event handler is set to call an anonymous function when *this* image has loaded.

When the image loads, the event handler checks if the `src` variable (which holds the path to *this* image) matches the `request` variable. If the user had clicked on another image since the one that just loaded, the `request` variable would no longer match the `src` variable and the image should not be shown.

CACHING IMAGES THAT HAVE ALREADY LOADED IN THE BROWSER

PROBLEM:
When the user requests a big image (by clicking on the thumbnail), a new `<img>` element is created and added to the frame.

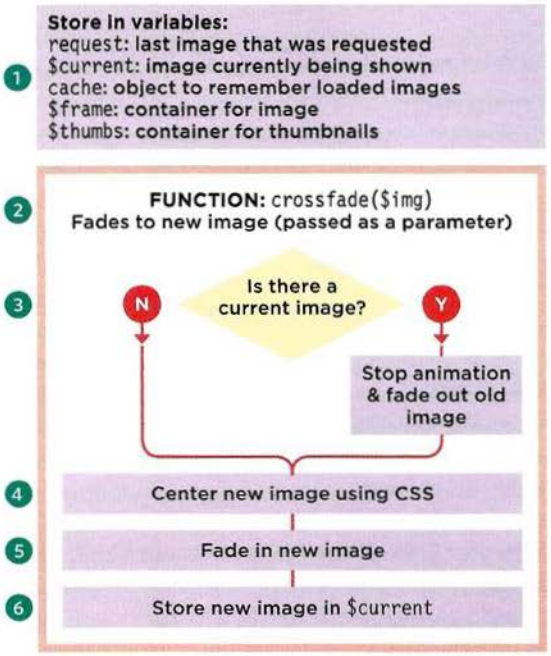
If the user goes back to look at an image they have already selected, you do not want to create a new element and load the image all over again.

SOLUTION:
A simple object is created, and it is called `cache`. Every time a new `<img>` element is created, it will be added to the `cache` object.

That way, each time an image is requested, the code can check if the corresponding `<img>` element is already in the cache (rather than creating it again).

PHOTO VIEWER SCRIPT (1)

This script introduces some new concepts, so it will be spread over four pages. On these two pages you see the global variables and `crossfade()` function.



1. A set of global variables is created. They can be used throughout the script – both in the `crossfade()` function (on this page) and the event handlers (on p512).
2. The `crossfade()` function will be called when the user has clicked on a thumbnail. It is used to fade between the old image and the new one.
3. An `if` statement checks to see if there is an image loaded at the moment. If there is, two things happen: the `.stop()` method will stop any current animation and then `.fadeOut()` will fade the image out.
4. To center the image in the viewer element, you set two CSS properties on the image. Combined with the CSS rules you saw on p508, these CSS properties will center the image in its container. (See the diagrams on the bottom of p511.)
   i) `marginLeft`: gets the width of the image using the `.width()` method, divides it by two, and uses that number as a negative margin.
   ii) `marginTop`: gets the height of the image, using the `.height()` method, divides it by two, and makes that number a negative margin.
5. If the new image is currently being animated, the animation is stopped and the image is faded in.
6. Finally, the new image becomes the current image and is stored in the `$current` variable.

THE CACHE OBJECT

The idea of a cache object might sound complicated, but all objects are just sets of key/value pairs. You can see what the cache object might look like on the right. When an image is requested by clicking on a new thumbnail, a new property is added to the cache object:

- The key added to the cache object is the path to the image (below this is referred to as *src*). Its value is another object with two properties.
- *src*`.$img` holds a reference to a jQuery object that contains the newly created `<img>` element.
- *src*`.isLoading` is a property indicating whether or not it is currently loading (its value is a Boolean).

```
var cache = {
  "c11/img/photo-1.jpg": {
    "$img": jQuery object,
    "isLoading": false
  },
  "c11/img/photo-2.jpg": {
    "$img": jQuery object,
    "isLoading": false
  },
  "c11/img/photo-3.jpg": {
    "$img": jQuery object,
    "isLoading": false
  }.
}
```

JAVASCRIPT

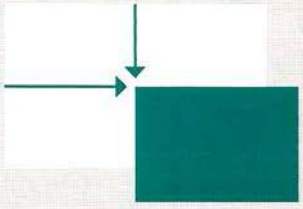
c11/js/photo-viewer.js

```
var request;                          // Latest image to be requested
var $current;                         // Image currently being shown
var cache   = {};                     // Cache object
var $frame  = $('#photo-viewer');     // Container for image
var $thumbs = $('.thumb');            // Container for image

function crossfade($img) {            // Function to fade between images
                                      // Pass in new image as parameter
  if ($current) {                     // If there is currently an image showing
    $current.stop().fadeOut('slow'); // Stop animation and fade it out
  }

  $img.css({                          // Set the CSS margins for the image
    marginLeft: -$img.width() / 2,    // Negative margin of half image's width
    marginTop: -$img.height() / 2     // Negative margin of half image's height
  });

  $img.stop().fadeTo('slow', 1);      // Stop animation on new image & fade in

  $current = $img;                    // New image becomes current image

}
```

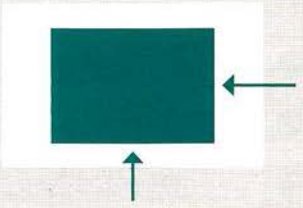
1. Cache object
2. Function to fade between images
3. Stop animation and fade it out
4. Set the CSS margins for the image
5. Stop animation on new image & fade in
6. New image becomes current image

CENTERING THE IMAGE

i) Centering the image involves three steps. In the style sheet, absolute positioning is used to place it in the top-left corner of the containing element.

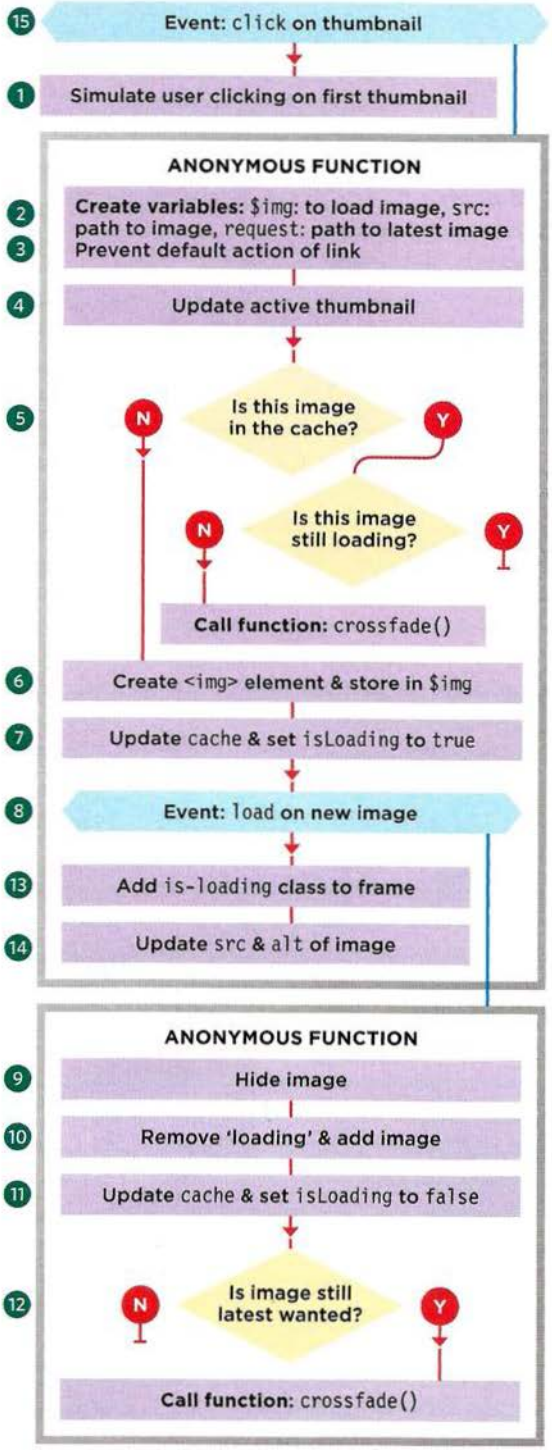


ii) In the style sheet, the image is moved down and right by 50% of the **container's** width and height:
width: 800px ÷ **2** = 400 px
height: 500px ÷ **2** = 250 px



iii) In the script, negative margins move the image up and left by half the **image's** width and height:
width: 500 px ÷ **2** = 250 px
height: 400px ÷ **2** = 200 px

PHOTO VIEWER SCRIPT (2)



1. The thumbnails are wrapped in links. Every time users click on one, the anonymous function will run.

2. Three variables are created:

i) `$img` will be used to create new `<img>` elements that will hold the larger images when they load.

ii) `src` (a function-level variable) holds the path to the new image (it was in the `href` attribute of the link).

iii) `request` (a global variable) holds the same path.

3. The link is prevented from loading the image.

4. The `active` class is removed from *all* the thumbs and is added to the thumb that was clicked on.

5. If the image is in the `cache` object and it has finished loading, the script calls `crossfade()`.

6. If the image has not yet loaded, the script creates a new `<img>` element.

7. It is added to the `cache`. `isLoading` is set to `true`.

8. At this point, the image has not loaded yet (only an empty `<img>` element was created). When the image loads, the `load` event triggers a function (which needs to be written *before* the image loads).

9. First, the function hides the image that just loaded.

10. It then removes the `is-loading` class from the frame and adds the new image to the frame.

11. In the `cache` object, `isLoading` is set to `false` (as it will have loaded when this function runs).

12. An if statement checks if the image that just loaded is the one the user last requested. To see how this is done, look back at step 2 again:

- The `src` variable holds the path to the image that just loaded. It has function-level scope.
- The `request` variable is updated each time the user clicks on an image. It has global scope.

So, if the user has clicked on an image since this one, the `request` and `src` variables will not be the same and nothing should be done. If they do match, then: `crossfade()` is called to show the image.

13. Having set all of this in place, it is time to load the image. The `is-loading` class is added to the frame.

14. Finally, by adding a value to the `src` attribute on the image, the image will start to load. Its alt text is retrieved from the `title` attribute on the link.

15. The last line of the script simulates the user clicking on the first thumbnail. This will load the first image into the viewer when the script first runs.

```
① $(document).on('click', '.thumb', function(e){ // When a thumb is clicked on
    var $img;                                    // Create local variable called $img
②   var src = this.href;                         // Store path to image
    request = src;                               // Store path again in request

③   e.preventDefault();                          // Stop default link behavior

    $thumbs.removeClass('active');               // Remove active from all thumbs
④   $(this).addClass('active');                  // Add active to clicked thumb

    if (cache.hasOwnProperty(src)) {             // If cache contains this image
      if (cache[src].isLoading === false) {      // And if isLoading is false
⑤       crossfade(cache[src].$img);              // Call crossfade() function
      }
    } else {                                     // Otherwise it is not in cache
⑥     $img = $('<img/>');                        // Store empty <img/> element in $img
      cache[src] = {                             // Store this image in cache
        $img: $img,                              // Add the path to the image
⑦       isLoading: true                          // Set isLoading property to true
      };

      // Next few lines will run when image has loaded but are prepared first
⑧     $img.on('load', function() {               // When image has loaded
⑨       $img.hide();                             // Hide it
        // Remove is-loading class from frame & append new image to it
⑩       $frame.removeClass('is-loading').append($img);
⑪       cache[src].isLoading = false;            // Update isLoading in cache
        // If still most recently requested image then
        if (request === src) {
⑫         crossfade($img);                       // Call crossfade() function
        }                                        // Solves asynchronous loading issue
      });

⑬     $frame.addClass('is-loading');             // Add is-loading class to frame

      $img.attr({                                // Set attributes on <img> element
        'src': src,                              // Add src attribute to load image
⑭       'alt': this.title || ''                  // Add title if one was given in link
      });

    }

  });

  // Last line runs once (when rest of script has loaded) to show the first image
⑮ $('.thumb').eq(0).click();                     // Simulate click on first thumbnail
```

RESPONSIVE SLIDER

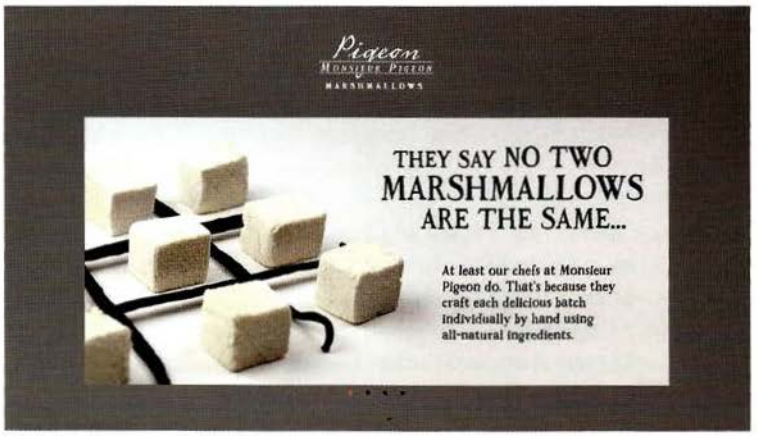
A slider positions a series of items next to each other, but only shows one at a time. The images then slide from one to the next.

This slider loads several panels, but only shows one at a time. It also provides buttons that allow users to navigate between each of the slides and a timer to move them automatically after a set interval.

In the HTML, the entire slider is contained within a `<div>` element whose `class` attribute has value of `slider-viewer`. In turn, the slider needs two further `<div>` elements:

- A container for the slides. Its `class` attribute has a value of `slide-group`. Inside this container, each individual slide is in another `<div>` element.
- A container for the buttons. Its `class` attribute has a value of `slide-buttons`. The buttons are added by the script.

If the HTML contains markup for more than one slider, the script will automatically transform all of them into separate sliders.



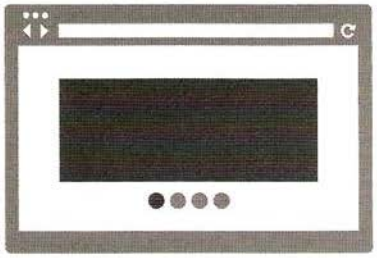
Other slider scripts include Unslider, Anything Slider, Nivo Slider, and WOW Slider. Sliders are also included in jQuery UI and Bootstrap.

When the page first loads, the CSS hides all of the slides, which takes them out of normal flow. The CSS then sets the `display` property of the first slide `block` to make it visible.

The script then goes through each slide and:

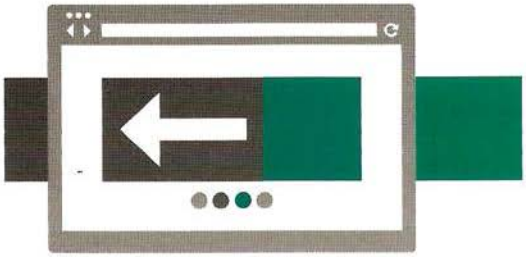
- Assigns an index number to that slide
- Adds a button for it under the slide group

For example, if there are four slides, when the page first loads, the first slide will be shown by default, and four buttons will be added underneath it.

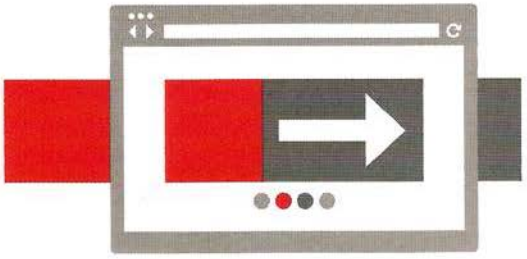


The index numbers allow the script to identify each individual slide. To keep track of which slide is currently being shown, the script uses a variable called `currentIndex` (holding the index number of the current slide). When the page loads, this is 0, so it shows the first slide. It also needs to know which slide it is moving to, which is stored in a variable called `newSlide`.

When it comes to moving between the slides (and creating the sliding effect), if the index number of the new slide is *higher* than the index number of the current slide, then the new slide is placed to the *right* of the group. As the visible slide is animated to the left, the new slide automatically starts to come into view, taking its place.



If the index number of the new slide is *lower* than the current index, then the new slide is placed to the *left* of the current slide, and as it is animated to the right, the new slide starts to come into view.



After the animation, the hidden slides are placed behind the one that is currently active.

USING THE SLIDER

As long as you include the script within your page, any HTML that uses the structure shown here will get transformed into a slider.

There could be several sliders on the page and each one will be transformed using the same script that you see on the next double-page spread.

c11/slider.html

HTML

```
<div class="slide-viewer">
  <div class="slide-group">
    <div class="slide slide-1"><!-- slide content --></div>
    <div class="slide slide-2"><!-- slide content --></div>
    <div class="slide slide-3"><!-- slide content --></div>
    <div class="slide slide-4"><!-- slide content --></div>
  </div>
</div>
<div class="slide-buttons"></div>
```

The width of the `slide-viewer` is not fixed, so it works in a responsive design. But a height does need to be specified because the slides have an absolute position (this removes them from the document flow and without it they could only be 1px tall).

Each slide is shown at the same width and height as the viewer. If the content of a slide is larger than the viewer, the `overflow` property on the `slide-viewer` hides the parts of the slides that extend beyond the frame. If it is smaller it is positioned to the top-left.

c11/css/slider.css

CSS

```
slide-viewer {
  position: relative;
  overflow: hidden;
  height: 300px;}

.slide-group {
  width: 100%;
  height: 100%;
  position: relative;}

.slide {
  width: 100%;
  height: 100%;
  display: none;
  position: absolute;}

.slide:first-child {
  display: block;}
```

SLIDER SCRIPT OVERVIEW

A jQuery selector finds the sliders within the HTML markup.
An anonymous function then runs for each one to create the slider.
There are four key parts to the function.

1: SETUP

Each slider needs some variables, they are in function-level scope so they:

- Can have different values for each slider
- Do not conflict with variables outside of the script

2: CHANGING SLIDE: `move()`

`move()` is used to move from one slide to another, and to update the buttons that indicate which slide is currently being shown. It is called when the user clicks on a button, and by the `advance()` function.

3: A TIMER TO SHOW THE NEXT SLIDE AFTER 4 SECONDS: `advance()`

A timer will call `move()` after 4 seconds.
To create a timer, JavaScript's `window` object has a `setTimeout()` method. It executes a function after a number of milliseconds. The timer is often assigned to a variable, and it uses the following syntax:

```
var timeout = setTimeout(function, delay);
```

- *`timeout`* is a variable name that will be used to identify the timer.
- *`function`* can be a named function or an anonymous function.
- *`delay`* is the number of milliseconds before the function should run.

To stop the timer, call `clearTimeout()`. It takes one parameter: the variable used to identify the timer:

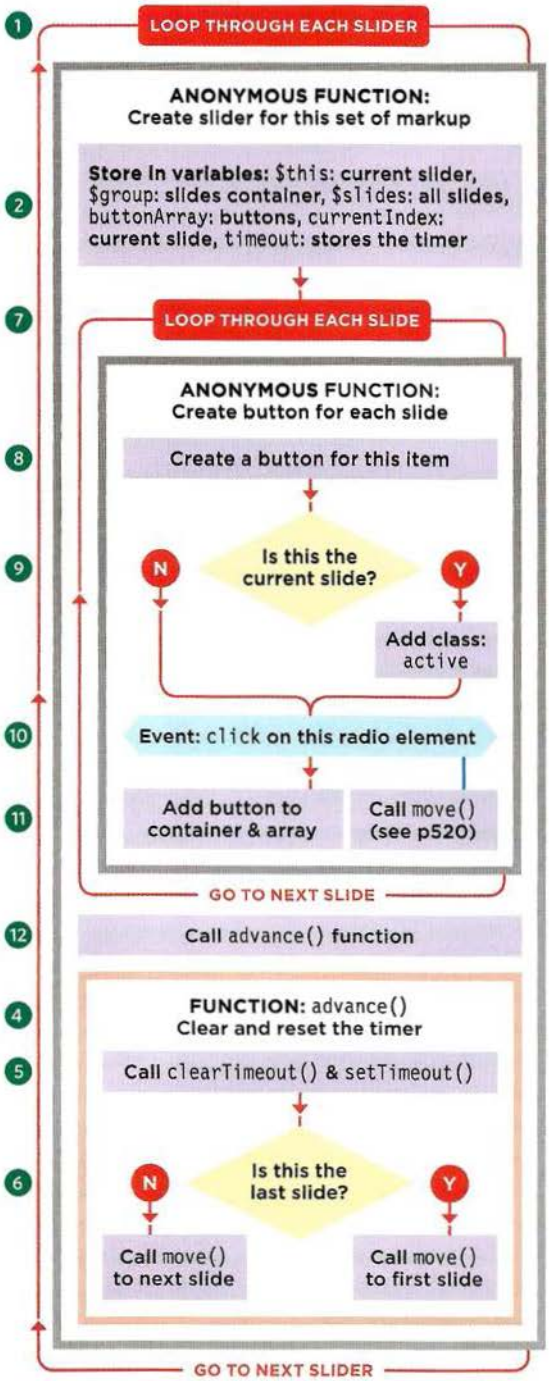
```
clearTimeout(timeout);
```

4: PROCESSING EACH OF THE SLIDES THAT APPEAR WITHIN A SLIDER

The code loops through each of the slides to:

- Create the slider
- Add a button for each slide with an event handler that calls the `move()` function when users clicks it

SLIDER SCRIPT



1. There may be several sliders on a page, so the script starts by looking for every element whose `class` attribute has a value of `slider`. For each one, an anonymous function is run to process that slider.
2. Variables are created to hold:
   i) The current slider
   ii) The element that wraps around the slides
   iii) All of the slides in this slider
   iv) An array of buttons (one for each slide)
   v) The current slide
   vi) The timer
3. The `move()` function appears next; see p520.
   Please note: This is not shown in the flowchart.
4. The `advance()` function creates the timer.
5. It starts by clearing the current timer. A new timer is set and when the time has elapsed it will run an anonymous function.
6. An `if` statement checks whether or not the current slide is the last one.
   If it is not the last slide then it calls `move()` with a parameter that tells it to go to the next slide.
   Otherwise it tells `move()` to go to the first slide.
7. Each slide is processed by an anonymous function.
8. A `<button>` element is created for each slide.
9. If the index number of that slide is the same as the number held in the `currentIndex` variable, then a class of `active` is added to that button.
10. An event handler is added to each button. When clicked it calls the `move()` function. The slide's index number indicates which slide to move to.
11. The buttons are then added to the button container, and to the array of buttons.
    This array is used by the `move()` function to indicate which slide is currently being shown.
12. `advance()` is called to start the timer.

JAVASCRIPT

c11/js/slider.js

```
① $('.slider').each(function(){                    // For every slider
    var $this    = $(this),                        // Get the current slider
    var $group   = $this.find('.slide-group'),     // Get the slide-group (container)
②   var $slides = $this.find('.slide'),           // jQuery object to hold all slides
    var buttonArray  = [],                         // Create array to hold nav buttons
    var currentIndex = 0,                          // Index number of current slide
    var timeout;                                   // Used to store the timer

③   // move() - The function to move the slides goes here (see next page)

④   function advance() {                           // Sets a timer between slides
      clearTimeout(timeout);                       // Clear timer stored in timeout
⑤     // Start timer to run an anonymous function every 4 seconds
      timeout = setTimeout(function(){             //
        if (currentIndex < ($slides.length - 1)) {  // If not the last slide
          move(currentIndex + 1);                  // Move to next slide
⑥       } else {                                   // Otherwise
          move(0);                                 // Move to the first slide
        }
      }, 4000);                                    // Milliseconds timer will wait
    }

⑦   $.each($slides, function(index){
      // Create a button element for the button
⑧     var $button = $('<button type="button" class="slide-btn">&bull;</button>');
      if (index === currentIndex) {                // If index is the current item
⑨       $button.addClass('active');                // Add the active class
      }
      $button.on('click', function(){              // Create event handler for the button
⑩       move(index);                               // It calls the move() function
      }).appendTo('.slide-buttons');               // Add to the buttons holder
⑪     buttonArray.push($button);                   // Add it to the button array
    });

⑫   advance();

  });
```

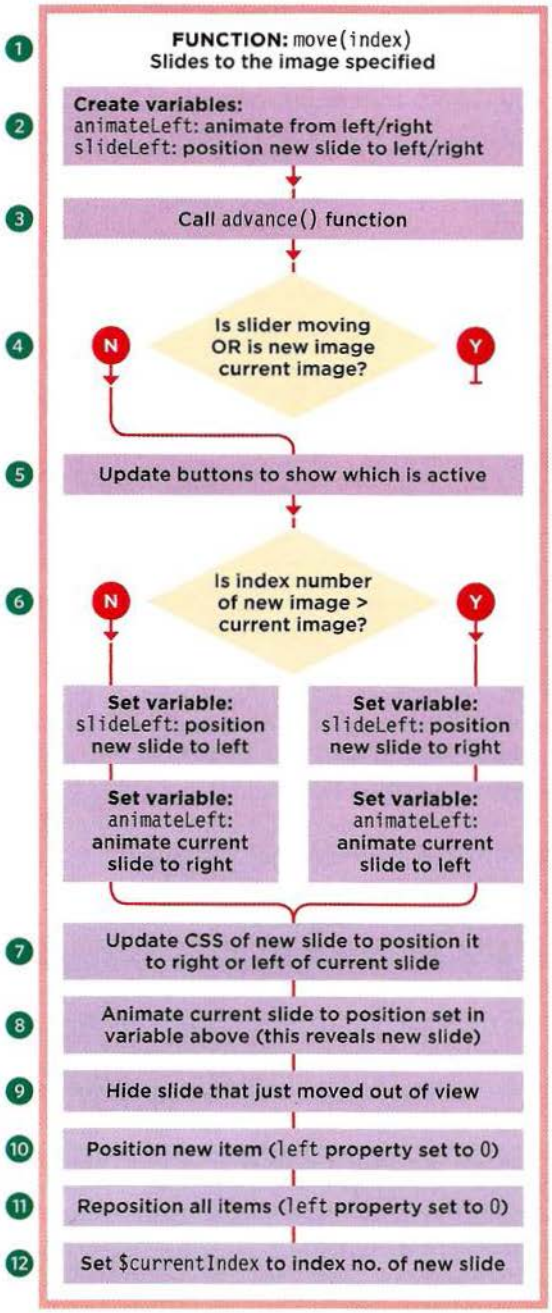
PROBLEM: GETTING THE RIGHT GAP BETWEEN SLIDES USING A TIMER

Each slide should show for four seconds (before the timer moves it on to the next slide). But if the user clicks a button after two seconds, then the new slide might not show for four seconds because the timer is already counting down.

SOLUTION: RESET THE TIMER WHENEVER A BUTTON IS CLICKED

The `advance()` function clears the timer before setting it off again. Every time the user clicks on a button the `move()` function (shown on the next two pages) it calls `advance()` to ensure the new slide is shown for four seconds.

SLIDER MOVE() FUNCTION



1. The `move()` function will create the animated sliding movement between two slides. When it is called, it needs to be told which slide to move to.

2. Two variables are created that are used to control whether the slider is moving to the left or right.

3. `advance()` is called to reset the timer.

4. The script checks if the slider is currently animating or if the user selected the current slide. In either case, nothing should be done, and the `return` statement stops the rest of the code from running.

5. References to each of the buttons were stored in an array in step 11 of the script on the previous page. The array is used to update which button is active.

6. If the new item has a higher index number, then the slider will need to move from right to left. If the item has a lower index number, the slider will need to move from left to right. These variable values are set first and are then used in step 7.

`slideLeft` positions the new slide in relation to the current slide. (100% sits the new slide to the right of it and -100% sits the new slide to the left of it.)

`animateLeft` indicates whether the current slide should move to the left or the right, letting the new slide take its place. (-100% moves the current slide to the left, 100% moves the current slide to the right.)

7. The new slide is positioned to the right or the left of the current slide using the value in the `slideLeft` variable and its `display` property is set to `block` so that it becomes visible. That new slide is identified using `newIndex`, which was passed into the function.

8. The current slide is then moved to the left or right using the value stored in the `animateLeft` variable. That slide is selected using the `currentIndex` variable, which was defined at the start of the script.

JAVASCRIPT

c11/js/slider.js

```
// Setup of the script shown on the previous page

① function move(newIndex) {           // Creates the slide from old to new one
②   var animateLeft, slideLeft;       // Declare variables

③   advance();                        // When slide moves, call advance() again

    // If current slide is showing or a slide is animating, then do nothing
    if ($group.is(':animated') || currentIndex === newIndex) {
④     return;
    }

    buttonArray[currentIndex].removeClass('active'); // Remove class from item
⑤   buttonArray[newIndex].addClass('active');        // Add class to new item

    if (newIndex > currentIndex) {    // If new item > current
      slideLeft = '100%';             // Sit the new slide to the right
      animateLeft = '-100%';          // Animate the current group to the left
⑥   } else {                          // Otherwise
      slideLeft = '-100%';            // Sit the new slide to the left
      animateLeft = '100%';           // Animate the current group to the right
    }
    // Position new slide to left (if less) or right (if more) of current
⑦   $slides.eq(newIndex).css( {left: slideLeft, display: 'block'} );
⑧   $group.animate( {left: animateLeft} , function() {   // Animate slides and
⑨     $slides.eq(currentIndex).css( {display: 'none'} ); // Hide previous slide
⑩     $slides.eq(newIndex).css( {left: 0} ); // Set position of the new item
⑪     $group.css( {left: 0} );               // Set position of group of slides
⑫     currentIndex = newIndex;               // Set currentIndex to new image
    });
  }

// Handling the slides shown on p519
```

Once the slide has finished animating, an anonymous function performs housekeeping tasks:

9. The slide that was the `currentIndex` is hidden.

10. The position of the left-hand side of the new slide is set to 0 (left-aligning it).

11. The position of all of the other slides is set so the left-hand side is 0 (left-aligning them).

12. At this point, the new slide will be visible, and the transition is complete, so it is time to update the `currentIndex` variable to hold the index number of the slide that has just been shown. This is easily done by giving it the value that was stored in the `newIndex` variable.

Now that this function has been defined, as you saw on the p519, the code creates a timer and goes through each slide adding a button and an event handler for it. (Steps 4-12 on the page p519.)

CREATING A JQUERY PLUGIN

jQuery plugins allow you to add new methods to jQuery without customizing the library itself.

jQuery plugins have benefits over plain scripts:

- You can perform the same task on any elements that match jQuery's flexible selector syntax
- Once the plugin has done its job, you can chain other methods after it (on the same selection)
- They facilitate re-use of code (either within one project or across multiple projects)
- They are commonly shared within the JavaScript and jQuery community
- Namespace collisions (problems when two scripts use the same variable name) are prevented by placing the script is placed in an IIFE (immediately invoked function expression, which you met on p97)

You can turn any function into a plugin if it:

- Manipulates a jQuery selection
- Can return a jQuery selection

The basic concept is that you:

- Pass it a set of DOM elements in a jQuery selection
- Manipulate the DOM elements using the jQuery plugin code
- Return the jQuery object so that other functions can be chained off it

This final example shows you how to create a jQuery plugin. It takes the accordion example you saw at the start of the chapter and turns it into a plugin.

The earlier version applied to all matching markup on the page; the plugin version requires that users call the `accordion()` method on a jQuery selection.

Here a jQuery selection is made collecting elements with a class of `menu`. The `.accordion()` method is called; once that has run, `.fadeIn()` is called.

```
$('.menu').accordion(500).fadeIn();
```

① ② ③

1. A jQuery selection is made containing any elements which have the class of `menu`.

2. The `.accordion()` method is called on those elements. It has one parameter; the speed of animation (in milliseconds).

3. The `.fadeIn()` method is applied to the same selection of elements once `.accordion()` has done its job.

BASIC PLUGIN STRUCTURE

1) ADDING A METHOD TO JQUERY

jQuery has an object called `.fn` which helps you extend the functionality of jQuery.

Plugins are written as methods that are added to the `.fn` object.

Parameters that can be passed to the function are placed inside the parentheses on the first line:

```
$.fn.accordion = function(speed) {
  // Plugin will go here
}
```

2) RETURNING THE JQUERY SELECTION TO CHAIN METHODS

jQuery works by collecting a set of elements and storing them in a `jQuery` object. The jQuery object's methods can be used to alter the selected elements.

Because jQuery lets you chain multiple methods to the same selection, once the plugin has done its job it should return the selection for the next method.

The selection is returned using:

1. The `return` keyword (sends a value back from a function)
2. `this` (refers to the selection that was passed in)

```
$.fn.accordion = function(speed) {
  // Plugin will go here
  return this;
}
```

3) PROTECTING THE NAMESPACE

jQuery is not the only JavaScript library to use $ as a shorthand, so the plugin code lives in an IIFE, which creates function-level scope for the code in the plugin.

On the first line below, the IIFE has one named parameter: $. On the last line, you can see that the `jQuery` selection is passed into the function.

Inside the plugin, $ acts like a variable name. It references the `jQuery` object containing the set of elements that the plugin is supposed to be working with.

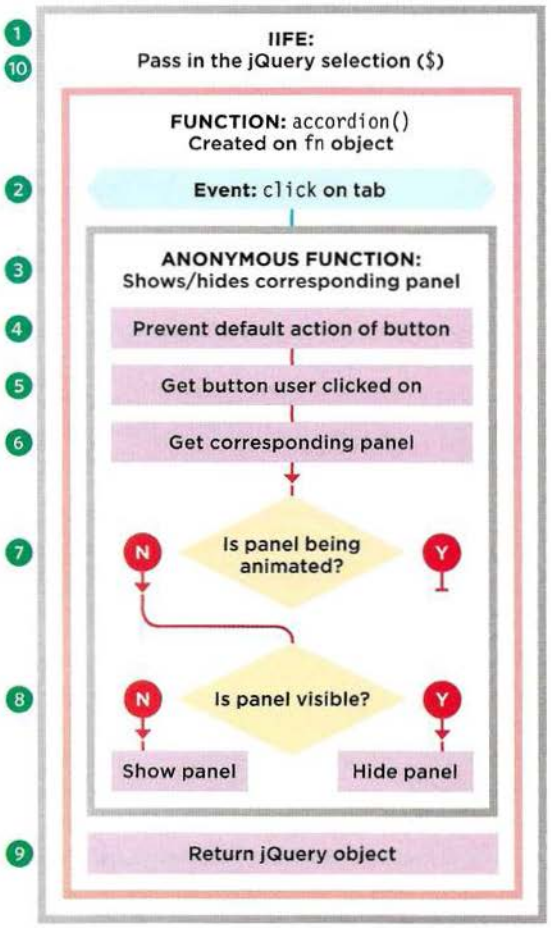
```
(function($){
  $.fn.accordion = function(speed) {
    // Plugin code will go here
  }
})(jQuery);
```

If you want to pass in more values, it is typically done using a single parameter called `options`.

When the function is called, the `options` parameter contains an object literal.

The object can contain a set of key/value pairs for the different options.

THE ACCORDION PLUGIN



To use the plugin, you create a jQuery selection that contains any `<ul>` elements that hold an accordion. In the example on the right, the accordion is in a `<ul>` element that has a class name of menu (but you could use any name you wish). You then call the `.accordion()` method on that selection, like so:

```
$('.menu').accordion(500);
```

This code could be placed in the HTML document (as shown on the right-hand page), but it would be better placed in a separate JavaScript file that runs when the page loads (to keep the JavaScript separate from the HTML).

You can see the full code for the accordion plugin on the right. The parts in orange are identical to the accordion script at the start of the chapter.

1. The plugin is wrapped in an IIFE to create function-level scope. On the first line, the function is given one named parameter: $ (which means you can use the $ shortcut for jQuery in the function).
10. On the last line of code, the jQuery object is passed into the function (using its full name `jQuery` rather than its shortcut $). This jQuery object contains the selection of elements that the plugin is working with. Together, points 1 and 10 mean that in the IIFE, $ refers to the `jQuery` object and it will not be affected if other scripts use $ as a shorthand, too.

2. Inside the IIFE, the new `.accordion()` method is created by extending the `fn` object. It takes the one parameter of `speed`.

3. The `this` keyword refers to the jQuery selection that was passed into the plugin. It is used to create an event handler that will listen for when the user clicks on an element with a `class` attribute whose value is `accordion-control`. When the user does, the anonymous function runs to animate the corresponding panel into or out of view.

4. The default action of the link is prevented.
5. In the anonymous function, `$(this)` refers to a jQuery object containing the element that the user clicked upon.
6. 7. 8. The only difference between this anonymous function and the one used in the example at the start of the chapter is that the `.slideToggle()` method takes a parameter of `speed` to indicate how fast the panel should be shown or hidden. (It is specified when the `.accordion()` method is called.)

9. When the anonymous function has done its work, the `jQuery` object containing the selected elements is returned from the function, allowing the same set of elements to be passed to another jQuery method.

JAVASCRIPT

c11/js/accordion-plugin.js

```
① (function($){                                    // Use $ as variable name
②   $.fn.accordion = function(speed) {             // Return the jQuery selection
③     this.on('click', '.accordion-control', function(e){
④         e.preventDefault();
⑤         $(this)
⑥             .next('.accordion-panel')
⑦             .not(':animated')
⑧             .slideToggle(speed);
      });
⑨     return this;                                 // Return the jQuery selection
    }
⑩ })(jQuery);                                      // Pass in jQuery object
```

Note how the filename for the jQuery plugin starts with `jquery.` to indicate that this script relies upon jQuery.

After the accordion plugin script has been included, the `accordion()` method can be used on any jQuery selection.

Below you can see the HTML for the accordion. This time it includes both the jQuery script and the jQuery accordion script.

HTML

c11/accordion-plugin.html

```
<ul class="menu">
  <li>
    <a href="#" class="accordion-control"><h3>Classics</h3></a>
    <div class="accordion-panel">If you like your flavors traditional...</div>
  </li>
  <li>
    <a href="#" class="accordion-control"><h3>The Flower Series</h3></a>
    <div class="accordion-panel">Take your tastebuds for a gentle...</div>
  </li>
  <li>
    <a href="#" class='accordion-control"><h3>Salt o' the Sea</h3></a>
    <div class="accordion-panel">Ahoy! If you long for a taste of...</div>
  </li>
</ul>
<script src="js/jquery.js"></script>
<script src="js/jquery.accordion.js"></script>
<script>
  $('.menu').accordion(500);
</script>
```

SUMMARY

CONTENT PANELS

- Content panels offer ways to show more content within a limited area.
- Popular types of content panels include accordions, tabs, photo viewers, modal windows, and sliders.
- As with all website code, it is advisable to separate content (HTML), presentation (CSS), and behavior (JavaScript) into different files.
- You can create objects to represent the functionality you want (as with the modal window).
- You can turn functions into jQuery plugins that allow you to re-use code and share it with others.
- Immediately invoked function expressions (IIFEs) are used to contain scope and prevent naming collisions.

12

FILTERING, SEARCHING & SORTING

If your pages contain a lot of data, there are tree techniques that you can use to help your users to find the content they are looking for.

FILTERING

Filtering lets you reduce a set of values, by selecting the ones that meet stated criteria.

SEARCH

Search lets you show the items that match one or more words the user specifies.

SORTING

Sorting lets you reorder a set of items on the page based on criteria (for example, alphabetically).

Before you get to see how to deal with filtering, searching, and sorting, it is important to consider how you are going to store the data that you are working with. In this chapter many of the examples will use arrays to hold data stored in objects using literal notation.


Feel the difference.
fall fruits
Aqua

JAVASCRIPT ARRAY METHODS

An array is a kind of object. All arrays have the methods listed below; their property names are index numbers. You will often see arrays used to store complex data (including other objects).

Each item in an array is sometimes called an **element**. It does not mean that the array holds HTML elements; element is just the name given to the pieces of information in the array. *Note some methods only work in IE9+.

ADDING ITEMS	`push()`	Adds one or more items to end of array and returns number of items in it
	`unshift()`	Adds one or more items to start of array and returns new length of it
REMOVING ITEMS	`pop()`	Removes last element from array (and returns the element)
	`shift()`	Removes first element from array (and returns the element)
ITERATING	`forEach()`	Executes a function once for each element in array*
	`some()`	Checks if some elements in array pass a test specified by a function*
	`every()`	Checks if all elements in array pass a test specified by a function*
COMBINING	`concat()`	Creates new array containing this array and other arrays/values
FILTERING	`filter()`	Creates new array with elements that pass a test specified by a function*
REORDERING	`sort()`	Reorders items in array using a function (called a compare function)
	`reverse()`	Reverses order of items in array
MODIFYING	`map()`	Calls a function on each element in array & creates new array with results

JQUERY METHODS FOR FILTERING & SORTING

jQuery collections are array-like objects representing DOM elements. They have similar methods to an array for modifying the elements. You can use other jQuery methods on the selection once they have run.

In addition to the jQuery methods shown below, you may see animation methods chained after filtering and sorting methods to create animated transitions as the user makes a selection.

ADDING OR COMBINING ITEMS	`.add()`	Adds elements to a set of matched elements
REMOVING ITEMS	`.not()`	Removes elements from a set of matched elements
ITERATING	`.each()`	Applies same function to each element in matched set
FILTERING	`.filter()`	Reduces number of elements in matched set to those that either match a selector or pass a test specified by a function
CONVERTING	`.toArray()`	Converts a jQuery collection to an array of DOM elements, enabling the use of the array methods shown on the left-hand page

SUPPORTING OLDER BROWSERS

Older browsers do not support the latest methods of the `Array` object. But a script called the ECMAScript 5 Shim can reproduce these methods. ECMAScript is the standard that modern JavaScript is based upon.

A BRIEF HISTORY OF JAVASCRIPT

1996	Jan	
	Feb	
	Mar	····· Netscape Navigator 2 contains the first version of JavaScript written by Brendan Eich
	Apr	
	May	
	Jun	
	Jul	
	Aug	····· Microsoft created a compatible scripting language called JScript
	Sep	
	Oct	
	Nov	····· Netscape gave JavaScript to the ECMA standards body so that its development could be standardized
	Dec	
1997	Jan	
	Feb	
	Mar	
	Apr	
	May	
	Jun	····· ECMAScript 1 was released
	Jul	
	Aug	
	Sep	
	Nov	
	Dec	
2014	**May**	····· Time of writing: ECMAScript 6 is close to being finalized

ECMAScript is the official name for the standardized version of JavaScript, although most people still call it JavaScript unless they are discussing new features.

ECMA International is a standards body that looks after the language, just like the W3C looks after HTML and CSS. And, browser manufacturers often add features beyond the ECMA specs (just as they do with HTML & CSS).

In the same way that the latest features from the HTML and CSS specifications are only supported in the most recent browsers, so the latest features of ECMAScript are only found in recent browsers. This will not affect much of what you have learned in this book (and jQuery helps iron out issues with backwards compatibility), but it is worth noting for the techniques you meet in this chapter.

The following methods of the `Array` object were all introduced in ECMAScript version 5, and they are not supported by Internet Explorer 8 (or older): `forEach()`, `some()`, `every()`, `filter()`, `map()`.

For these methods to work in older browsers you include the ECMAScript 5 Shim, a script that reproduces their functionality for legacy browsers: `https://github.com/es-shims/es5-shim`

ARRAYS VS. OBJECTS CHOOSING THE BEST DATA STRUCTURE

In order to represent complex data you might need several objects. Groups of objects can be stored in arrays or as properties of other objects. When deciding which approach to use, consider how you will use the data.

OBJECTS IN AN ARRAY

When the order of the objects is important, they should be stored in an array because each item in an array is given an index number. (Key-value pairs in objects are not ordered.) But note that the index number can change if objects are added/removed. Arrays also have properties and methods that help when working with a sequence of items, e.g.,

- The `sort()` method reorders items in an array.
- The `length` property counts the number of items.

```
var people = [
  {name: 'Casey', rate: 70, active: true},
  {name: 'Camille', rate: 80, active: true},
  {name: 'Gordon', rate: 75, active: false},
  {name: 'Nigel', rate: 120, active: true}
]
```

To retrieve data from an array of objects, you can use the index number for the object:

```
// This retrieves Camille's name and rate
person[1].name;
person[1].rate;
```

To add/remove objects in an array you use array methods.

To iterate over the items in an array you can use `forEach()`.

OBJECTS AS PROPERTIES

When you want to access objects using their name, they work well as properties of another object (because you would not need to iterate through all objects to find that object as you would in an array).

But note that each property must have a unique name. For example, you could not have two properties both called `Casey` or `Camille` within the same object in the following code.

```
var people = {
  Casey = {rate: 70, active: true},
  Camille = {rate: 80, active: true},
  Gordon = {rate: 75, active: false},
  Nigel = {rate: 120, active: true}
}
```

To retrieve data from an object stored as a property of another object, you can the object's name:

```
// This retrieves Casey's rate
people.Casey.rate;
```

To add/remove objects to an object you can use the `delete` keyword or set it to a blank string.

To iterate over child objects you can use `Object.keys`.

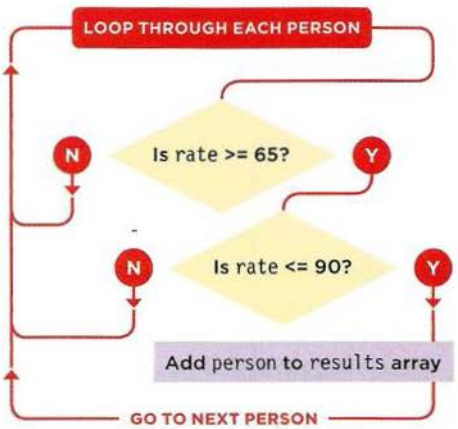
FILTERING

Filtering lets you reduce a set of values.
It allows you to create a subset of data that meets certain criteria.

To look at filtering, we will start with data about freelancers and their hourly rate. Each person is represented by an object literal (in curly braces). The group of objects is held in an array:

```
var people = [
  {
    name: 'Casey',
    rate: 60
  },
  {
    name: 'Camille',
    rate: 80
  },
  {
    name: 'Gordon',
    rate: 75
  },
  {
    name: 'Nigel',
    rate: 120
  }
];
```

The data will be filtered before it is displayed. To do this we will loop through the objects that represent each person. If their rate is more than $65 and less than $90, they are put in a new array called `results`.



NAME	HOURLY RATE ($)
Camille	80
Gordon	75

DISPLAYING THE ARRAY

On the next two pages, you will see two different approaches to filtering the data in the `people` array, both of which involve using methods of the `Array` object: `.forEach()` and `.filter()`.

Both methods will be used to go through the data in the `people` array, find the ones who charge between $65 and $90 per hour and then add those people to a new array called `results`.

Once the new `results` array has been created, a `for` loop will go through it adding the people to an HTML table (the result is shown on the left-hand page).

Below, you can see the code that displays the data about the people who end up in the `results` array:

1. The entire example runs when the DOM is ready.
2. The data about people and their rates is included in the page (this data is shown on left-hand page).
3. A function will filter the data in the `people` array and create a new array called `results` (next page).
4. A `<tbody>` element is created.
5. A `for` loop goes through the array and uses jQuery to create a new table row for each person and their hourly rate.
6. The new content is added to the page after the table heading.

JAVASCRIPT

c12/js/filter-foreach.js + c12/js/filter-filter.js

```
① $(function() {
②   // DATA ABOUT PEOPLE GOES HERE (shown on left-hand page)

③   // FILTERING CODE (see p537) GOES HERE - CREATES A NEW ARRAY CALLED results

    // LOOP THROUGH NEW ARRAY AND ADD MATCHING PEOPLE TO THE RESULTS TABLE
④   var $tableBody = $('<tbody></tbody>');            // New content jQuery
    for (var i = 0; i < results.length; i++) {       // Loop through matches
      var person = results[i];                       // Store current person
      var $row = $('<tr></tr>');                     // Create a row for them
⑤     $row.append($('<td></td>').text(person.name)); // Add their name
      $row.append($('<td></td>').text(person.rate)); // Add their rate
      $tableBody.append( $row );                     // Add row to new content
    }

    // Add the new content after the body of the page
⑥   $('thead').after($tableBody);                    // Add tbody after thead
  });
```

USING ARRAY METHODS TO FILTER DATA

The array object has two methods that are very useful for filtering data. Here you can see both used to filter the same set of data. As they filter the data, the items that pass a test are added to a new array.

The two examples on the right both start with an array of objects (shown on p534) and use a filter to create a new array containing a subset of those objects. The code then loops through the new array to show the results (as you saw on the previous page).

- The first example uses the `forEach()` method.
- The second example uses the `filter()` method.

Note how `person` is used as a parameter name and acts as a variable inside the functions:

- In the `forEach()` example it is used as a parameter of the anonymous function.
- In the `filter()` example it is used as a parameter of the `priceRange()` function.

It corresponds to the current object from the `people` array and is used to access that object's properties.

forEach()

The `forEach()` method loops through the array and applies the same function to every item in it. `forEach()` is very flexible because the function can perform any kind of processing with the items in an array (not just filtering as shown in this example). The anonymous function acts as a filter because it checks if a person's rates are within a specified range and, if so, adds them to a new array.

1. A new array is created to hold matching results.
2. The `people` array uses the `forEach()` method to run the same anonymous function on each object (that represents a person) in the `people` array.
3. If they match the criteria, they are added to the `results` array using the `push()` method.

filter()

The `filter()` method also applies the same function to each item in the array, but that function only returns `true` or `false`. If it returns `true`, the `filter()` method adds that item to a new array.

The syntax is slightly simpler than `forEach()`, but is only meant to be used to filter data.

1. A function called `priceRange()` is declared; it will return `true` if the person's wages are within the specified range.
2. A new array is created to hold matching results.
3. The `filter()` method applies the `priceRange()` function to each item in the array. If `priceRange()` returns `true`, that item is added to the `results` array.

STATIC FILTERING OF DATA

JAVASCRIPT c12/js/filter-foreach.js

```
$(function() {
  // DATA ABOUT PEOPLE GOES HERE (shown on p534)

  // CHECKS EACH PERSON AND ADDS THOSE IN RANGE TO ARRAY
① var results = [];                                    // Array for people in range
② people.forEach(function(person) {                    // For each person
    if (person.rate >= 65 && person.rate <= 90) {      // Is rate in range
③     results.push(person);                            // If yes add to array
    }
  });

  // LOOP THROUGH RESULTS ARRAY AND ADD MATCHING PEOPLE TO THE RESULTS TABLE
});
```

JAVASCRIPT c12/js/filter-filter.js

```
$(function() {
  // DATA ABOUT PEOPLE GOES HERE (shown on p534)

  // THE FUNCTION ACTS AS A FILTER
  function priceRange(person) {                          // Declare priceRange()
①   return (person.rate >= 65) && (person.rate <= 90); // In range returns true
  };
  // FILTER THE PEOPLE ARRAY & ADD MATCHES TO THE RESULTS ARRAY
② var results = [];                                    // Array for matching people
③ results = people.filter(priceRange);                 // filter() calls priceRange()

  // LOOP THROUGH RESULTS ARRAY AND ADD MATCHING PEOPLE TO THE RESULTS TABLE
});
```

The code that you saw on the p535 to show the table results could live in the `.forEach()` method, but it is separated out here to illustrate the different approaches to filtering and how they can create new arrays.

DYNAMIC FILTERING

If you let users filter the contents of a page, you can build all of the HTML content, and then show and hide the relevant parts as the user interacts with the filters.

Imagine that you were going to provide the user with a slider so that they could update the price that they were prepared to pay per hour. That slider would automatically update the contents of the table based upon the price range the user had specified.

If you built a new table every time the user interacts with the slider (like the previous two examples that showed filtering), it would involve creating and deleting a lot of elements. Too much of this type of DOM manipulation can slow down your scripts.

A far more efficient solution would be to:

1. Create a table row for *every* person.
2. Show the rows for the people that are within the specified range, and hide the rows that are outside the specified bounds.

Below, the range slider used is a jQuery plugin called noUiSlider (written by Léon Gerson).
`http://refreshless.com/nouislider/`

Before you see the code for this example, take a moment to think about how to approach this script... Here are the tasks that the script needs to perform:

i) It needs to go through each object in the array and create a row for that person.

ii) Once the rows have been created, they need to be added to the table.

iii) Each row needs to be shown / hidden depending on whether that person is within the price range shown on the slider. (This task happens each time the slider is updated.)

In order to decide which rows to show / hide, the code needs to cross-reference between:

- The `person` object in the `people` array (to check how much that person charges)
- The row that corresponds to that person in the table (which needs to be made visible or hidden)

To build this cross-reference we can create a new array called `rows`. It will hold a series of objects with two properties:

- `person`: a reference to the object for this person in the `people` array
- `$element`: a jQuery collection containing the corresponding row in the table

In the code, we create a function to represent each of the tasks identified on the left. The new cross-reference array will be created in the first function:

`makeRows()` will create a row in the table for each person *and* add the new object into the `rows` array

`appendRows()` loops through the `rows` array and adds each of the rows to the table

`update()` will determine which rows are shown or hidden based on data taken from the slider

In addition, we will add a fourth function: `init()` This function contains all of the information that needs to run when the page first loads (including creating the slider using the plugin).

`init` is short for **initialize**; you will often see programmers using this name for functions or scripts that run when the page first loads.

Before looking at the script in detail, the next two pages are going to explain a little more about the `rows` array and how it creates the cross-reference between the objects and the rows that represent each person.

STORING REFERENCES TO OBJECTS & DOM NODES

The **rows** array contains objects with two properties, which associate:
1: References to the objects that represent people in the **people** array
2: References to the row for those people in the table (jQuery collections)

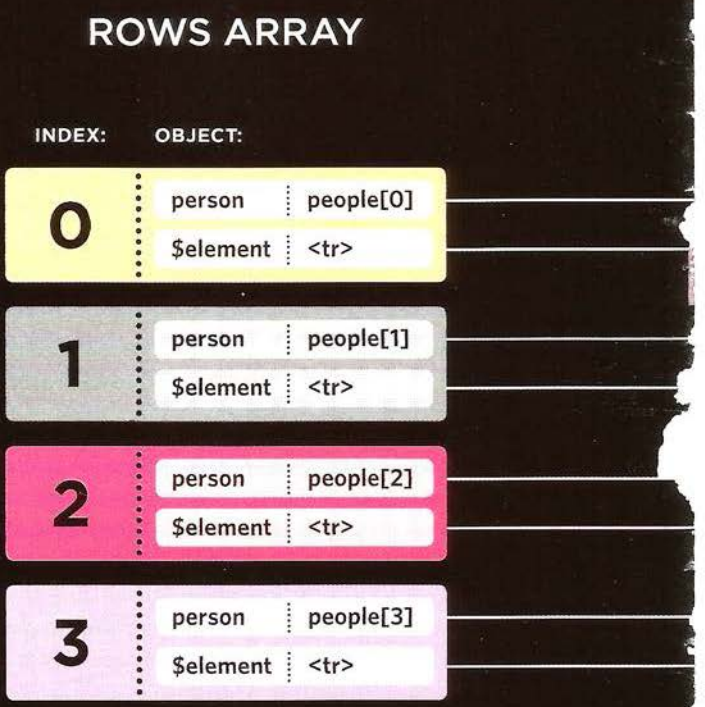
You have seen examples in this book where variables were used to store a reference to a DOM node or jQuery selection (rather than making the same selection twice). This is known as **caching**.

This example takes that idea further: as the code loops through each object in the **people** array creating a row in the table for that person, it also creates a new object for that person and adds it to an array called **rows**. Its purpose is to create an association between:

- The object for that person in the source data
- The row for that person in the table

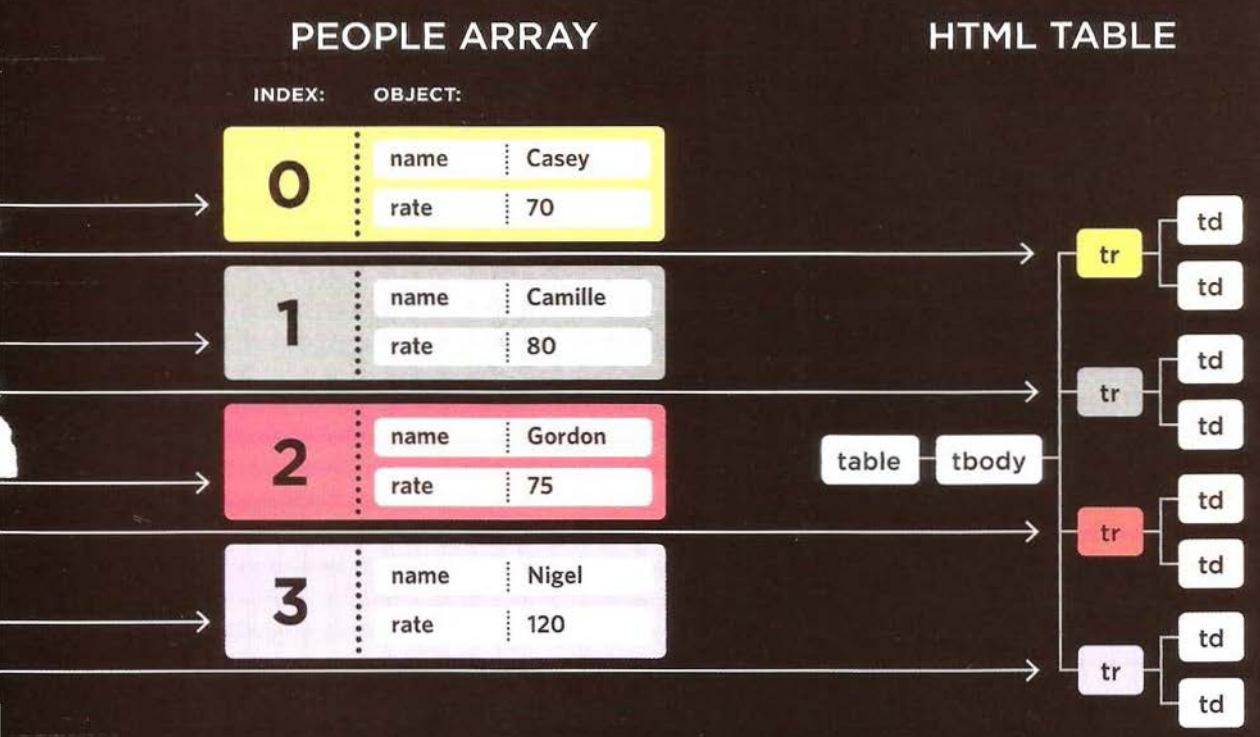
When deciding which rows to show, the code can then loop through this new array checking the person's rate. If they are affordable, it can show the row. If not, it can hide the row.

This takes less resources than recreating the contents of the table when the user changes the rate they are willing to pay.



On the right, you can see the **Array** object's **push()** method creates a new entry in the **rows** array. The entry is an object literal, and it stores the **person** object and the row being created for it in the table.

```
rows.push({
  person: this,     // person object
  $element: $row    // jQuery collection
});
```



The **people** array already holds information about each person and the rates that they charge, so the object in the **rows** array only needs to point to the original object for that person (it does not copy it).

A jQuery object was used to create each row of the table. The objects in the **rows** array store a reference to each individual row of the table. There is no need to select or create the row again.

DYNAMIC FILTERING

1. Place the script in an IIFE (not shown in flowchart). The IIFE starts with the `people` array.
2. Next, four global variables are created as they are used throughout the script:
`rows` holds the cross-referencing array .
`$min` holds the input to show the minimum rate.
`$max` holds the input to show the maximum rate.
`$table` holds the table for the results.

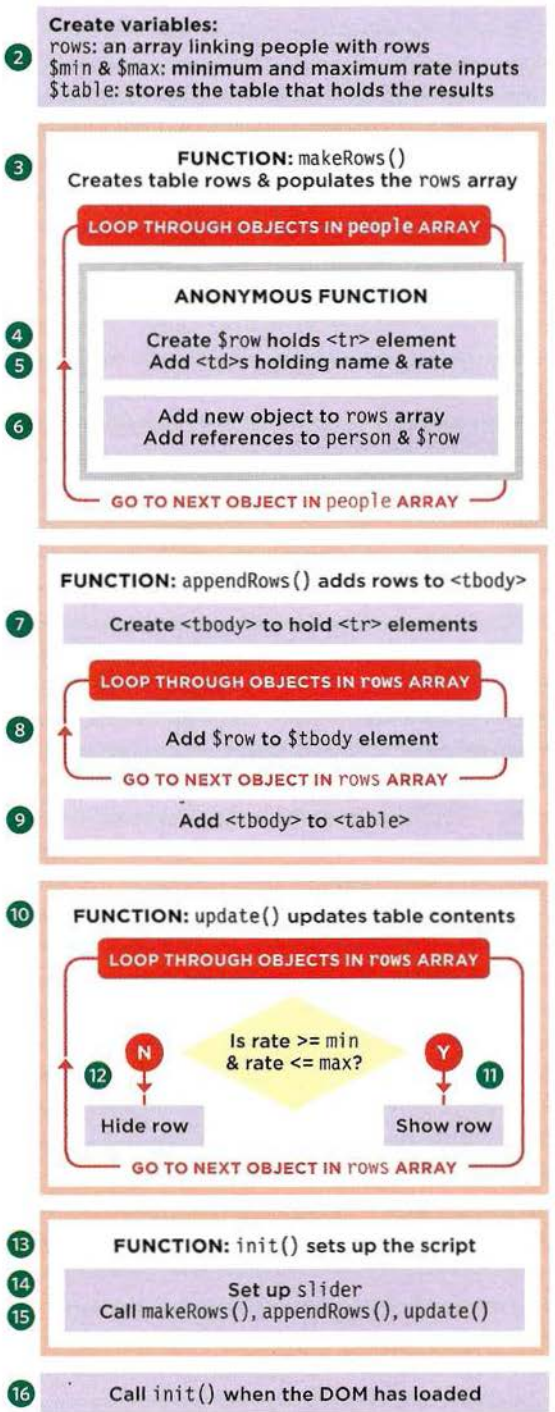
3. `makeRows()` loops through each person in the `people` array calling an anonymous function for each object in the array. Note how `person` is used as a parameter name. This means that within the function, `person` refers to the current object in the array.
4. For each person, a new jQuery object called `$row` is created containing a `<tr>` element.
5. The person's name and rate are added in `<td>`s.
6. A new object with two properties is added to the `rows` array: `person` stores a reference to their object, `$element` stores a reference to their `<tr>` element.

7. `appendRows()` creates a new jQuery object called `$tbody` containing a `<tbody>` element.
8. It then loops through all of the objects in the `rows` array and adds their `<tr>` element to `$tbody`.
9. The new `$tbody` selection is added to the `<table>`.

10. `update()` goes through each of the objects in the `rows` array and checks if the rate that the person charges is more than the minimum and less than the maximum rate shown on the slider.
11. If it is, jQuery's `show()` method shows the row.
12. If not, jQuery's `hide()` method hides the row.

13. `init()` starts by creating the slide control.
14. Every time the slider is changed, the `update()` function is called again.
15. Once the slider has been set up, the `makeRows()`, `appendRows()`, `update()` functions are called.

16. The `init()` function is called (which will in turn call the other code).



FILTERING AN ARRAY

JAVASCRIPT c12/js/dynamic-filter.js

```
① (function(){                              // PEOPLE ARRAY GOES HERE
    var rows = [],                          // rows array
②       $min = $('#value-min'),             // Minimum text input
        $max = $('#value-max'),             // Maximum text input
        $table = $('#rates');               // The table that shows results
③   function makeRows() {                   // Create table rows and the array
④     people.forEach(function(person) {     // For each person object in people
        var $row = $('<tr></tr>');          // Create a row for them
⑤       $row.append( $('<td></td>').text(person.name) ); // Add their name
        $row.append( $('<td></td>').text(person.rate) ); // Add their rate
        rows.push({ // Add object to cross-references between people and rows
⑥         person: person,                   // Reference to the person object
          $element: $row                    // Reference to row as jQuery selection
        });
      });
    }
⑦   function appendRows() {                 // Adds rows to the table
      var $tbody = $('<tbody></tbody>');    // Create <tbody> element
⑧     rows.forEach(function(row) {          // For each object in the rows array
        $tbody.append(row.$element);        // Add the HTML for the row
      });
⑨     $table.append($tbody);                // Add the rows to the table
    }
⑩   function update(min, max) {            // Update the table content
      rows.forEach(function(row) {          // For each row in the rows array
⑪       if (row.person.rate >= min && row.person.rate <= max) { // If in range
          row.$element.show();              // Show the row
⑫       } else {                          // Otherwise
          row.$element.hide();              // Hide the row
        }
      });
    }
⑬   function init() {                      // Tasks when script first runs
      $('#slider').noUiSlider({             // Set up the slide control
        range: [0, 150], start: [65, 90], handles: 2, margin: 20, connect: true,
        serialization: { to: [$min,$max], resolution: 1 }
⑭     }).change(function() { update($min.val(), $max.val()); });
      makeRows();                             // Create table rows and rows array
⑮     appendRows();                           // Add the rows to the table
      update($min.val(), $max.val());         // Update table to show matches
    }
⑯   $(init);                               // Call init() when DOM is ready
  }());
```

FILTERED IMAGE GALLERY

In this example, a gallery of images are tagged. Users click on filters to show matching images.

IMAGES ARE TAGGED

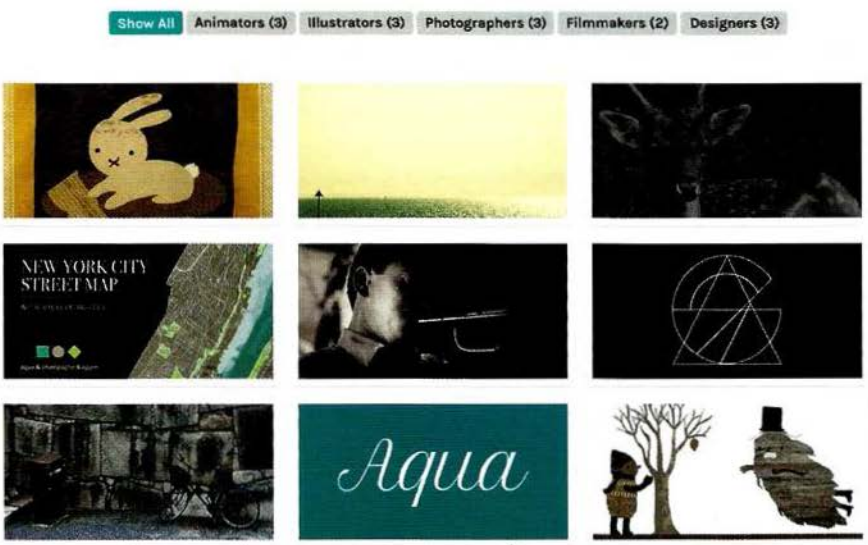
In this example, a series of photos are tagged. The tags are stored in an HTML attribute called `data-tags` on each of the `<img>` elements. HTML5 allows you to store any data with an element using an attribute that starts with the word `data-`. The tags are comma-separated.
(See right-hand page)

TAGGED OBJECT

The script creates an object called `tagged`. The script then goes through each of the images looking at its tags. Each tag is added as a property of the `tagged` object. The value of that property is an array holding a reference to each `<img>` element that uses that tag.
(See p546-p547)

FILTER BUTTONS

By looping through each of the keys on the `tagged` object, the buttons can automatically be generated. The tag counts come from the `length` of the array. Each button is given an event handler. When clicked, it filters the images and only shows those with the tag the user selected.
(See p548-p549)



TAGGED IMAGES

HTML

c12/filter-tags.html

```
<body>
  <header>
    <h1>CreativeFolk</h1>
  </header>
  <div id="buttons"></div>
  <div id="gallery">
    <img src="img/p1.jpg" data-tags="Animators, Illustrators" alt="Rabbit" />
    <img src="img/p2.jpg" data-tags="Photographers, Filmmakers" alt="Sea" />
    <img src="img/p3.jpg" data-tags="Photographers, Filmmakers" alt="Deer" />
    <img src="img/p4.jpg" data-tags="Designers" alt="New York Street Map" />
    <img src="img/p5.jpg" data-tags="Filmmakers" alt="Trumpet Player" />
    <img src="img/p6.jpg" data-tags="Designers, Animators" alt="Logo Ident" />
    <img src="img/p7.jpg" data-tags="Photographers" alt="Bicycle Japan" />
    <img src="img/p8.jpg" data-tags="Designers" alt="Aqua Logo" />
    <img src="img/p9.jpg" data-tags="Animators, Illustrators" alt="Ghost" />
  </div>
  <script src="js/jquery.js"></script>
  <script src="js/filter-tags.js"></script>
</body>
```

On the right, you can see the `tagged` object for the HTML sample used in this example. For each new tag in the images' `data-tags` attribute, a property is created on the `tagged` object. Here it has five properties: `animators`, `designers`, `filmmakers`, `illustrators`, and `photographers`. The value is an array of images that use that tag.

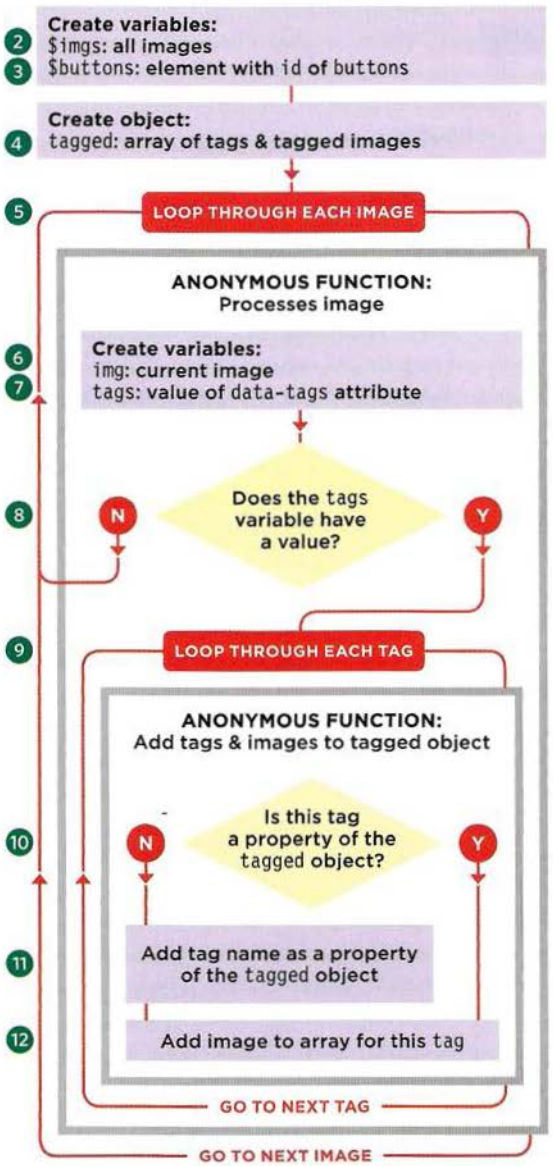
```
tagged = {
  animators: [p1.jpg, p6.jpg, p9.jpg],
  designers: [p4.jpg, p6.jpg, p8.jpg]
  filmmakers: [p2.jpg, p3.jpg, p5.jpg]
  illustrators: [p1.jpg, p9.jpg]
  photographers: [p2.jpg, p3.jpg, p8.jpg]
}
```

PROCESSING THE TAGS

Here you can see how the script is set up. It loops through the images and the `tagged` object is given a new property for each tag. The value of each property is an array holding the images with that tag.

1. Place the script in an IIFE (not shown in flowchart).
2. The `$imgs` variable holds a jQuery selection containing the images.
3. The `$buttons` variable holds a jQuery selection holding the container for the buttons.
4. The `tagged` object is created.
5. Loop through each of the images stored in `$imgs` using jQuery's `.each()` method. For each one, run the same anonymous function:
6. Store the current image in a variable called `img`.
7. Store the tags from the current image in a variable called `tags`. (The tags are found in the image's `data-tags` attribute.)
8. If the `tags` variable for this image has a value:
9. Use the `String` object's `split()` method to create an array of tags (splitting them at the comma). Chaining the `.forEach()` method off the `split()` method lets you run an anonymous function for each of the elements in the array (in this case, each of the tags on the current image). For each tag:
10. Check if the tag is already a property of the `tagged` object.
11. If not, add it as a new property whose value is an empty array.
12. Then get the property of the `tagged` object that matches this tag and add the image to the array that is stored as the value of that property.

Then move onto the next tag (go back to step **10**). When all of the tags for that image have been processed, move to the next image (step **5**).



THE TAGGED OBJECT

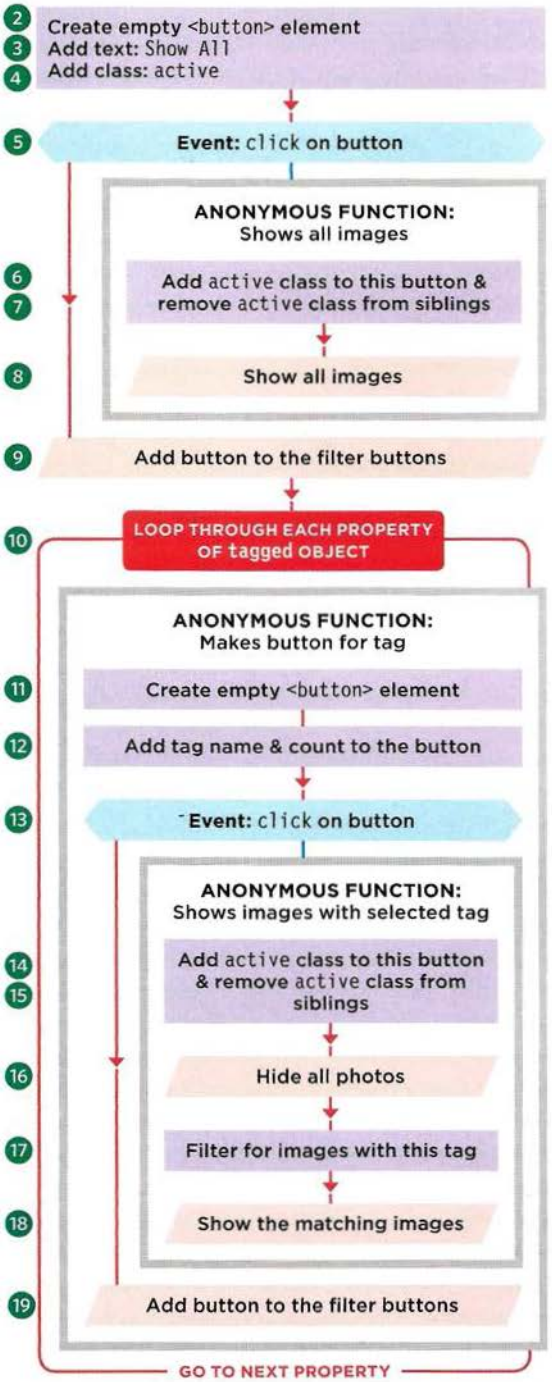
JAVASCRIPT c12/js/filter-tags.js

```
① (function() {

②   var $imgs = $('#gallery img');                    // Store all images
③   var $buttons = $('#buttons');                     // Store buttons element
④   var tagged = {};                                  // Create tagged object

⑤   $imgs.each(function() {                           // Loop through images and
⑥     var img = this;                                 // Store img in variable
⑦     var tags = $(this).data('tags');                // Get this element's tags

⑧     if (tags) {                                     // If the element had tags
⑨       tags.split(',').forEach(function(tagName) { // Split at comma and
⑩         if (tagged[tagName] == null) {            // If object doesn't have tag
⑪           tagged[tagName] = [];                   // Add empty array to object
          }
⑫         tagged[tagName].push(img);                // Add the image to the array
        });
      }
    });

    // Buttons, event handlers, and filters go here (see p549)

  }());
```

FILTERING THE GALLERY

The filter buttons are created and added by the script. When a button is clicked, it triggers an anonymous function, which will hide and show the appropriate images for that tag.

1. The script lives in an IIFE (not shown in flowchart).
2. Create the button to show all images. The second parameter is an object literal that sets its properties:
3. The text on the button is set to say 'Show All'.
4. A value of `active` is added to the `class` attribute.
5. When the user clicks on the button, an anonymous function runs. When that happens:
6. This button is stored in a jQuery object and is given a `class` of `active`.
7. Its siblings are selected, and the `class` of `active` is removed from them.
8. The `.show()` method is called on all images.
9. The button is then appended to the button container using the `.appendTo()` method. This is chained off the jQuery object that was just created.
10. Next, the other filter buttons are created. jQuery's `$.each()` method is used to loop through each property (or each tag) in the `tagged` object. The same anonymous function runs for each tag:
11. A button is created for the tag using the same technique you saw for the 'Show All' button.
12. The text for the button is set to the tag name, followed by the length of the array (which is the number of images that have that tag).
13. The `click` event on that button triggers an anonymous function:
14. This button is given a `class` of `active`.
15. `active` is removed from all of its siblings.
16. Then all of the images are hidden.
17. The jQuery `.filter()` method is used to select the images that have the specified tag. It does a similar job to the `Array` object's `.filter()` method, but it returns a jQuery collection. It can also work with an object or an element array (as shown here).
18. The `.show()` method is used to show the images returned by the `.filter()` method.
19. The new button is added to the other filter buttons using the `.appendTo()` method.



THE FILTER BUTTONS

JAVASCRIPT

c12/js/filter-tags.js

```
(function() {

  // Create variables (see p547)
  // Create tagged object (see p547)

  $('<button/>', {                                  // Create empty button
    text: 'Show All',                               // Add text 'show all'
    class: 'active',                                // Make it active
    click: function() {                             // Add onclick handler to it
      $(this)                                       // Get the clicked on button
        .addClass('active')                         // Add the class of active
        .siblings()                                 // Get its siblings
        .removeClass('active');                     // Remove active from them
      $imgs.show();                                 // Show all images
    }
  }).appendTo($buttons);                            // Add to buttons

  $.each(tagged, function(tagName){                 // For each tag name
    $('<button/>', {                                // Create empty button
      text: tagName + ' (' + tagged[tagName].length + ')', // Add tag name
      click: function() {                           // Add click handler
        $(this)                                     // The button clicked on
          .addClass('active')                       // Make clicked item active
          .siblings()                               // Get its siblings
          .removeClass('active');                   // Remove active from them
        $imgs                                       // With all of the images
          .hide()                                   // Hide them
          .filter(tagged[tagName])                  // Find ones with this tag
          .show();                                  // Show just those images
      }
    }).appendTo($buttons);                          // Add to the buttons
  });
}());
```

SEARCH

Search is like filtering but you show only results that match a search term. In this example, you will see a technique known as livesearch. The `alt` text for the image is used for the search instead of tags.

SEARCH LOOKS IN ALT TEXT OF IMAGES

This example will use the same set of photos that you saw in the last example, but will implement a livesearch feature. As you type, the images are narrowed down to match the search criteria.

The search looks at the `alt` text on each image and shows only `<img>` elements whose `alt` text contains the search term.

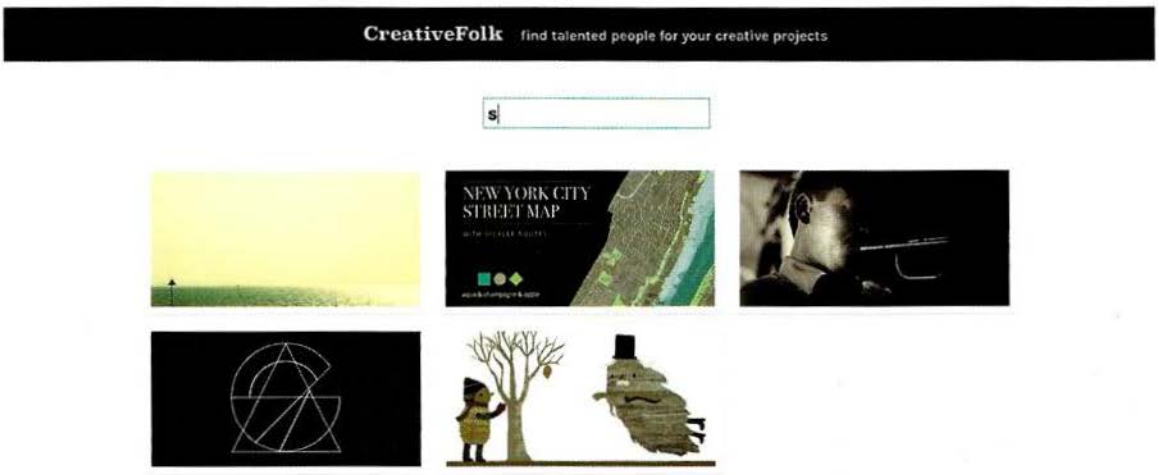
IT USES INDEXOF() TO FIND A MATCH

The `indexOf()` method of the `String` object is used to check for the search term. If it is not found, `indexOf()` returns `-1`. Since `indexOf()` is case-sensitive, it is important to convert all text (both the `alt` text and the search term) to lowercase (which is done using the `String` object's `toLowerCase()` function).

SEARCH A CUSTOM CACHE OBJECT

We do not want to do the case conversion for each image every time the search terms change, so an object called `cache` is created to store the text along with the image that uses that text.

When the user enters something into the search box, this object is checked rather than looking through each of the images.



SEARCHABLE IMAGES

HTML c12/filter-search.html

```
<body>
  <header>
    <h1>CreativeFolk</h1>
  </header>
  <div id="search">
    <input type="text" placeholder="filter by search" id="filter-search" />
  </div>
  <div id="gallery">
    <img src="img/p1.jpg" data-tags="Animators, Illustrators" alt="Rabbit" />
    <img src="img/p2.jpg" data-tags="Photographers, Filmmakers" alt="Sea" />
    <img src="img/p3.jpg" data-tags="Photographers, Filmmakers" alt="Deer" />
    <img src="img/p4.jpg" data-tags="Designers" alt="New York Street Map" />
    <img src="img/p5.jpg" data-tags="Filmmakers" alt="Trumpet Player" />
    <img src="img/p6.jpg" data-tags="Designers, Animators" alt="Logo Ident" />
    <img src="img/p7.jpg" data-tags="Photographers" alt="Bicycle Japan" />
    <img src="img/p8.jpg" data-tags="Designers" alt="Aqua Logo" />
    <img src="img/p9.jpg" data-tags="Animators, Illustrators" alt="Ghost" />
  </div>
  <script src="js/jquery.js"></script>
  <script src="js/filter-search.js"></script>
</body>
```

For each of the images, the cache array is given a new object. The array for the HTML above would look like the one shown on the right (except where it says `img`, it stores a reference to the corresponding `<img>` element).

When the user types in the search box, the code will look in the `text` property of each object, and if it finds a match, it will show the corresponding image.

```
cache = [
  {element: img, text: 'rabbit'},
  {element: img, text: 'sea'},
  {element: img, text: 'deer'},
  {element: img, text: 'new york street map'},
  {element: img, text: 'trumpet player'},
  {element: img, text: 'logo ident'},
  {element: img, text: 'bicycle japan'},
  {element: img, text: 'aqua logo'},
  {element: img, text: 'ghost'}
]
```

SEARCH TEXT

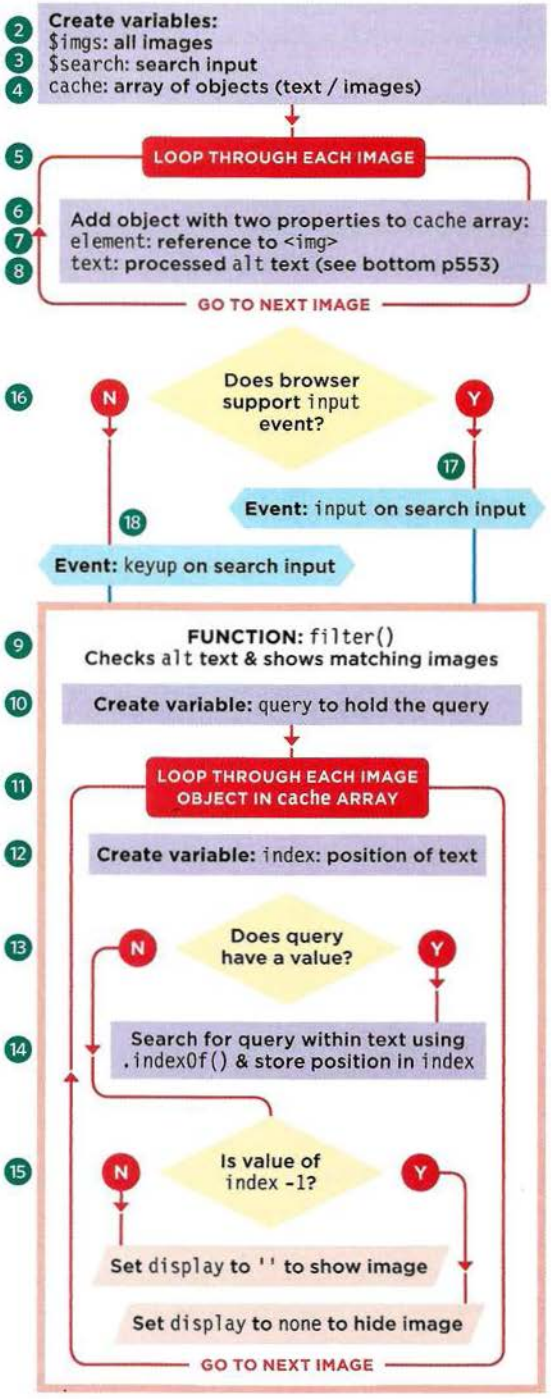
This script can be divided into two key parts:

SETTING UP THE CACHE OBJECT

1. Place the script in an IIFE (not shown in flowchart).
2. The `$imgs` variable holds a jQuery selection containing the images.
3. `$search` holds search input.
4. The cache array is created.
5. Loop through each image in `$imgs` using `.each()`, and run an anonymous function on each one:
6. Use `push()` to add an object to the cache array representing that image.
7. The object's `element` property holds a reference to the `<img>` element.
8. Its `text` property holds the `alt` text. Note that two methods process the text:
`.trim()` removes spaces from the start and end.
`.toLowerCase()` converts it all to lowercase.

FILTERING IMAGES WHEN USER TYPES IN SEARCH BOX

9. Declare a function called `filter()`.
10. Store the search text in a variable called `query`. Use `.trim()` and `.toLowerCase()` to clean the text.
11. Loop through each object in the cache array and call the same anonymous function on each:
12. A variable called `index` is created and set to 0.
13. If `query` has a value:
14. Use `indexOf()` to check if the search term is in the `text` property of this object.
The result is stored in the `index` variable. If found, it will be a positive number. If not, it will be -1.
15. If the value of `index` is `-1`, set the `display` property of the image to `none`. Otherwise, set `display` to a blank string (showing the image). Move onto the next image (step 11).
16. Check if the browser supports the `input` event. (It works well in modern browsers, but is not supported in IE8 or earlier.)
17. If so, when it fires on the search box, call the `filter()` function.
18. Otherwise, use the `input` event to trigger it.



LIVESEARCH

JAVASCRIPT

c12/js/filter-search.js

```
(function() {                                  // Lives in an IIFE
  var $imgs = $('#gallery img');               // Get the images
  var $search = $('#filter-search');           // Get the input element
  var cache = [];                              // Create an array called cache

  $imgs.each(function() {                      // For each image
    cache.push({                               // Add an object to the cache array
      element: this,                           // This image
      text: this.alt.trim().toLowerCase()      // Its alt text (lowercase trimmed)
    });
  });

  function filter() {                          // Declare filter() function
    var query = this.value.trim().toLowerCase();  // Get the query

    cache.forEach(function(img) {              // For each entry in cache pass image
      var index = 0;                           // Set index to 0
      if (query) {                             // If there is some query text
        index = img.text.indexOf(query);       // Find if query text is in there
      }

      img.element.style.display = index === -1 ? 'none' : '';  // Show / hide
    });
  }

  if ('oninput' in $search[0]) {               // If browser supports input event
    $search.on('input', filter);               // Use input event to call filter()
  } else {                                     // Otherwise
    $search.on('keyup', filter);               // Use keyup event to call filter()
  }
}());
```

The `alt` text of every image and the text that the user enters into the search input are cleaned using two jQuery methods. Both are used on the same selection and are chained after each other.

METHOD	USE
`trim()`	Removes whitespace from start or end of string
`toLowerCase()`	Converts string to lowercase letters because `indexOf()` is case-sensitive

SORTING

Sorting involves taking a set of values and reordering them. Computers often need detailed instructions about in order to sort data. In this section, you meet the `Array` object's `sort()` method.

When you sort an array using the `sort()` method, you change the order of the items it holds.

Remember that the elements in an array have an index number, so sorting can be compared to changing the index numbers of the items in the array.

By default, the `sort()` method orders items **lexicographically**. It is the same order dictionaries use, and it can lead to interesting results (see the numbers below).

To sort items in a different way, you can write a compare function (see right-hand page).

Lexicographic order is as follows:
1. Look at the first letter, and order words by the first letter.
2. If two words share the same first letter, order those words by the second letter.
3. If two words share the same first two letters, order those words by the third letter, etc.

SORTING STRINGS

Take a look at the array on the right, which contains names. When the `sort()` method is used upon the array, it changes the order of the names.

```
var names = ['Alice', 'Ann', 'Andrew', 'Abe'];
names.sort();
```

The array is now ordered as follows:

```
['Abe', 'Alice', 'Andrew', 'Ann'];
```

SORTING NUMBERS

By default, numbers are also sorted lexicographically, and you can get some unexpected results. To get around this you would need to create a compare function (see next page).

```
var prices = [1, 2, 125, 19, 14, 156];
prices.sort();
```

The array is now ordered as follows:

```
[1, 125, 14, 156, 19, 2]
```

CHANGING ORDER USING COMPARE FUNCTIONS

If you want to change the order of the sort, you write a compare function. It compares two values at a time and returns a number. The number it returns is then used to rearrange the items in the array.

The `sort()` method only ever compares two values at a time (you will see these referred to as *a* and *b*), and it determines whether value *a* should appear before or after value *b*.

Because only two values are compared at a time, the `sort()` method may need to compare each value in the array with several other values in the array (see diagram on the next page).

`sort()` can have an anonymous or a named function as a parameter. This function is called a **compare function** and it lets you create rules to determine whether value *a* should come before or after value *b*.

COMPARE FUNCTIONS MUST RETURN NUMBERS

A compare function should return a number. That number indicates which of the two items should come first.

The `sort()` method will determine which values it needs to compare to ensure the array is ordered correctly.

You just write the compare function so that it returns a number that reflects the order in which you want items to appear.

<0

Indicates that it should show *a* before *b*

0

Indicates that the items should remain in the same order

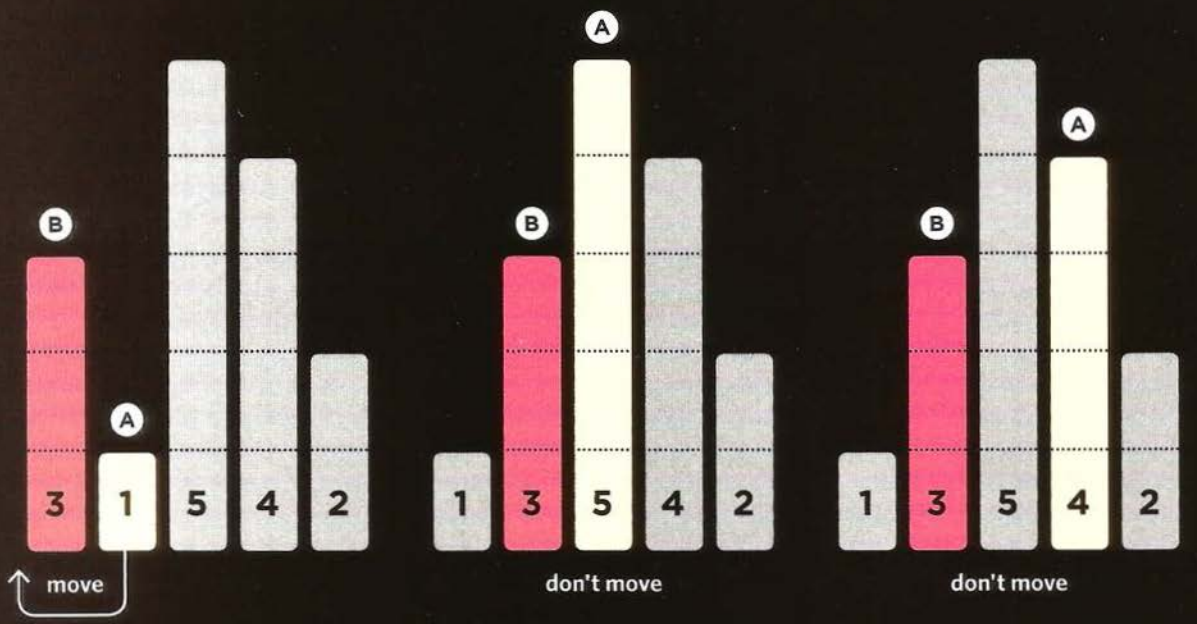
>0

Indicates that it should show *b* before *a*

To see the order in which the values are being compared, you can add the `console.log()` method to the compare function. For example: `console.log(a + ' - ' + b + ' = ' + (b - a));`

HOW SORTING WORKS

Here an array holds 5 numbers that will be sorted in ascending order.
You can see how two values (**a** and **b**) are compared against each other.
The compare function has rules to decide which of the two goes first.



a should go *before* **b**

1 - 3 = -2

a - b = < 0

a should go *after* **b**

5 - 3 = 2

a - b = > 0

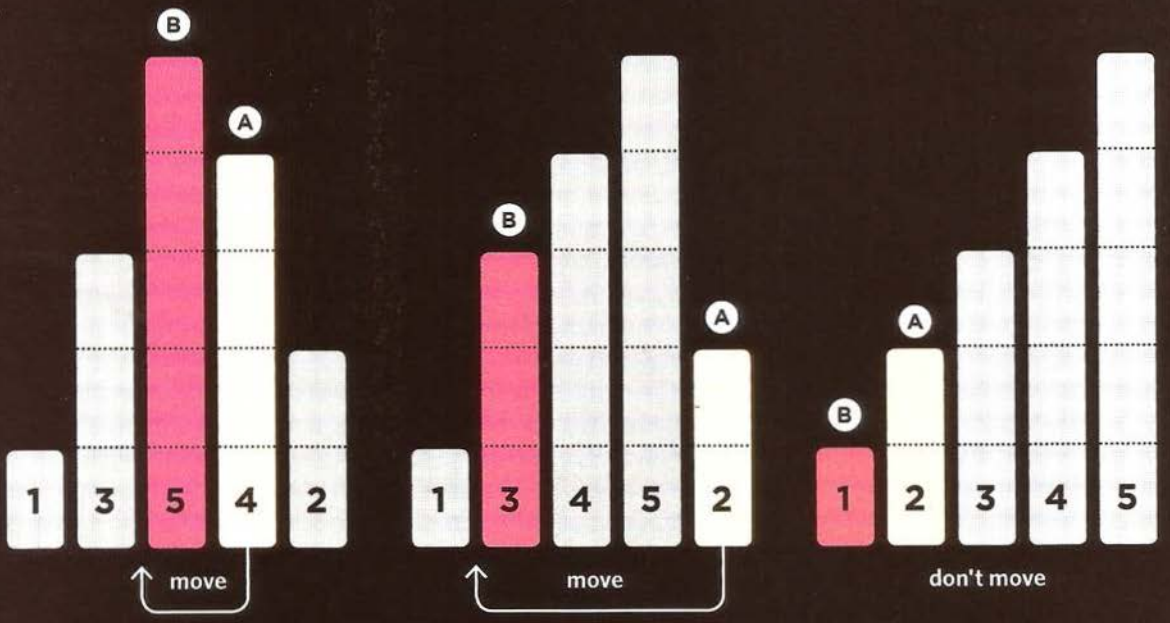
a should go *after* **b**

4 - 3 = 1

a - b = > 0

It is up to the browser to decide which order to sort items in .
This illustrates the order used by Safari. Other browsers sort items in a different order.

```
var prices = [3, 1, 5, 4, 2];  // Numbers stored in an array
prices.sort(function(a, b) {   // Two values are compared
  return a - b;                // Decides which goes first
});
```



a should go *before* **b**

4 - 5 = -1

a - b = < 0

a should go *before* **b**

2 - 3 = -1

a - b = < 0

a should go *after* **b**

2 - 1 = 1

a - b = > 0

Chrome compares this array in the following order: 3 - 4, 5 - 2, 4 - 2, 3 - 2, 1 - 2.
Firefox compares this array in the following order: 3 - 1, 3 - 5, 4 - 2, 5 - 2, 1 - 2, 3 - 2, 3 - 4, 5 - 4.

SORTING NUMBERS

Here are some examples of compare functions that can be used as a parameter of the `sort()` method.

ASCENDING NUMERICAL ORDER

To sort numbers in an ascending order, you subtract the value of the second number *b* from the first number *a*. In the table on the right, you can see examples of how two values from the array are compared.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function(a, b) {
  return a - b;
});
```

a	OPERATOR	*b*	RESULT	ORDER
1	-	2	-1	*a* comes before *b*
2	-	2	0	leave in same order
2	-	1	1	*b* comes before *a*

DESCENDING NUMERICAL ORDER

To order numbers in a descending order, you subtract the value of the first number *a* from the second number *b*.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function(a, b) {
  return b - a;
});
```

b	OPERATOR	*a*	RESULT	ORDER
2	-	1	1	*b* comes before *a*
2	-	2	0	leave in same order
1	-	2	-1	*a* comes before *b*

RANDOM ORDER

This will randomly return a value between -1 and 1 creating a random order for the items.

```
var prices = [1, 2, 125, 2, 19, 14];
prices.sort(function() {
  return 0.5 - Math.random();
});
```

SORTING DATES

Dates need to be converted into a `Date` object so that they can then be compared using < and > operators.

```
var holidays = [
  '2014-12-25',
  '2014-01-01',
  '2014-07-04',
  '2014-11-28'
];

holidays.sort(function(a, b){
    var dateA = new Date(a);
    var dateB = new Date(b);

    return dateA - dateB
});
```

The array is now ordered as follows:

```
holidays = [
  '2014-01-01',
  '2014-07-04',
  '2014-11-28',
  '2014-12-25'
]
```

DATES IN ASCENDING ORDER

If the dates are held as strings, as they are in the array shown on the left, the compare function needs to create a `Date` object from the string so that the two dates can be compared.

Once they have been converted into a `Date` object, JavaScript stores the date as the number of milliseconds since the 1st January 1970.

With the date stored as a number, two dates can be compared in the same way that numbers are compared on the left-hand page.

SORTING A TABLE

In this example, the contents of a table can be reordered.
Each row of the table is stored in an array.
The array is then sorted when the user clicks on a header.

SORT BY HEADER

When users click on a heading, it triggers an anonymous function to sort the contents of the array (which contains the table rows). The rows are sorted in ascending order using data in that column.

Clicking the same header again will show the same column sorted in descending order.

DATA TYPES

Each column can contain one of the following types of data:

- Strings
- Time durations (mins/secs)
- Dates

If you look at the `<th>` elements, the type of data used is specified in an attribute called `data-sort`.

COMPARE FUNCTIONS

Each type of data needs a different compare function. The compare functions will be stored as three methods of an object called `compare`, which you create on p563:

- `name()` sorts strings
- `duration()` sorts mins/secs
- `date()` sorts dates

HTML TABLE STRUCTURE

1. The `<table>` element needs to carry a `class` attribute whose value contains `sortable`.

2. Table headers have an attribute called `data-sort`. It reflects the type data in that column.

The value of the `data-sort` attribute corresponds with the methods of the `compare` object.

HTML

c12/sort-table.html

```
<body>
  <table class="sortable">
    <thead>
      <tr>
        <th data-sort="name">Genre</th>
        <th data-sort="name">Title</th>
        <th data-sort="duration">Duration</th>
        <th data-sort="date">Date</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Animation</td>
        <td>Wildfood</td>
        <td>3:47</td>
        <td>2014-07-16</td>
      </tr>
      <tr>
        <td>Film</td>
        <td>The Deer</td>
        <td>6:24</td>
        <td>2012-02-28</td>
      </tr>
      <tr>
        <td>Animation</td>
        <td>The Ghost</td>
        <td>11:40</td>
        <td>2013-04-10</td>
      </tr>...
    </tbody>
  </table>
  <script src="js/jquery.js"></script>
  <script src="js/sort-table.js"></script>
</body>
```

COMPARE FUNCTIONS

1. Declare the `compare` object. It has three methods used to sort names, time durations, and dates.

THE `name()` METHOD

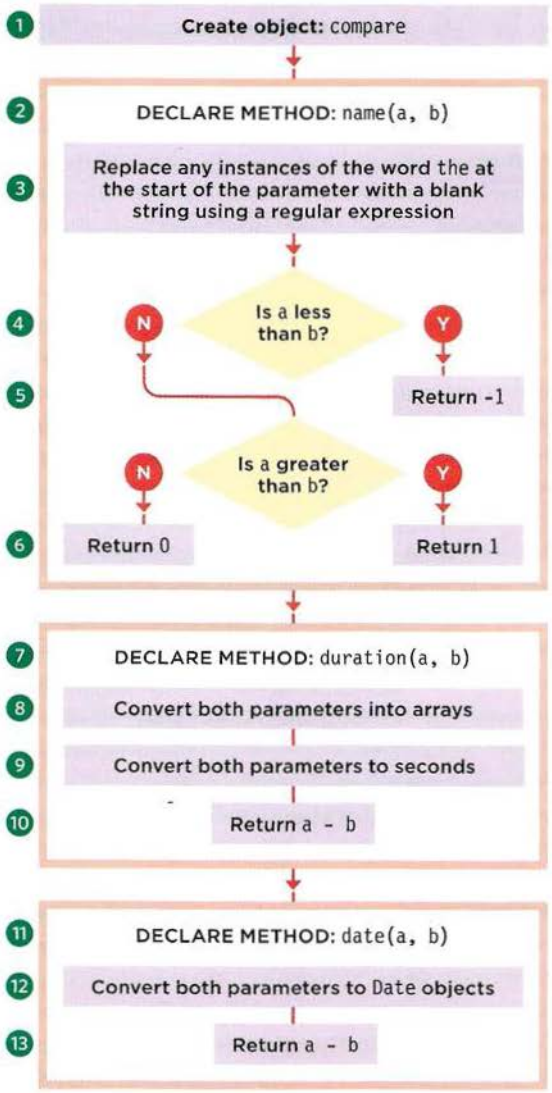
2. Add a method called `name()`. Like all compare functions, it should take two parameters: a and b.
3. Use a regular expression to remove the word 'the' from the beginning of both of the arguments that have been passed into the function (for more on this technique, see the bottom of the right-hand page).
4. If the value of a is lower than that of b:
5. Return -1 (indicating that a should come before b).
6. Otherwise, if a is greater than b, return 1. Or, if they are the same, return 0. (See bottom of page.)

THE `duration()` METHOD

7. Add a method called `duration()`. Like all compare functions, it should take two parameters: a and b.
8. Duration is stored in minutes and seconds: `mm:ss`. The `String` object's `split()` method splits the string at the colon, and creates an array with minutes and seconds as separate entries.
9. To get the total duration in seconds, `Number()` converts the strings in the arrays to numbers. The minutes are multiplied by 60 and added to the number of seconds.
10. The value of a - b is returned.

THE `date()` METHOD

11. Add a method called `date()`. Like all compare functions, it should take two parameters: a and b.
12. Create a new `Date` object to represent each of the arguments passed into the method.
13. Return the value of a minus b.



`return a > b ? 1 : 0`

A shorthand for a conditional operator is the **ternary operator**. It evaluates a condition and returns one of two values. The condition is shown to the left of the question mark.

The two options are shown to the right separated by a colon. If the condition returns a truthy value, the first value is returned. If the value is falsy, the value after the colon is returned.

THE COMPARE OBJECT

JAVASCRIPT

c12/js/sort-table.js

```
① var compare = {                                  // Declare compare object
②   name: function(a, b) {                         // Add a method called name
③     a = a.replace(/^the /i, '');                 // Remove The from start of parameter
      b = b.replace(/^the /i, '');                 // Remove The from start of parameter

④     if (a < b) {                                 // If value a is less than value b
⑤       return -1;                                 // Return -1
      } else {                                     // Otherwise
⑥       return a > b ? 1 : 0;                      // If a is greater than b return 1 OR
      }                                            // if they are the same return 0
    },
⑦   duration: function(a, b) {                     // Add a method called duration
⑧     a = a.split(':');                            // Split the time at the colon
      b = b.split(':');                            // Split the time at the colon

⑨     a = Number(a[0]) * 60 + Number(a[1]);        // Convert the time to seconds
      b = Number(b[0]) * 60 + Number(b[1]);        // Convert the time to seconds

⑩     return a - b;                                // Return a minus b
    },
⑪   date: function(a, b) {                         // Add a method called date
⑫     a = new Date(a);                             // New Date object to hold the date
      b = new Date(b);                             // New Date object to hold the date

⑬     return a - b;                                // Return a minus b
    }
  };
```

a.replace(/^the /i, '');

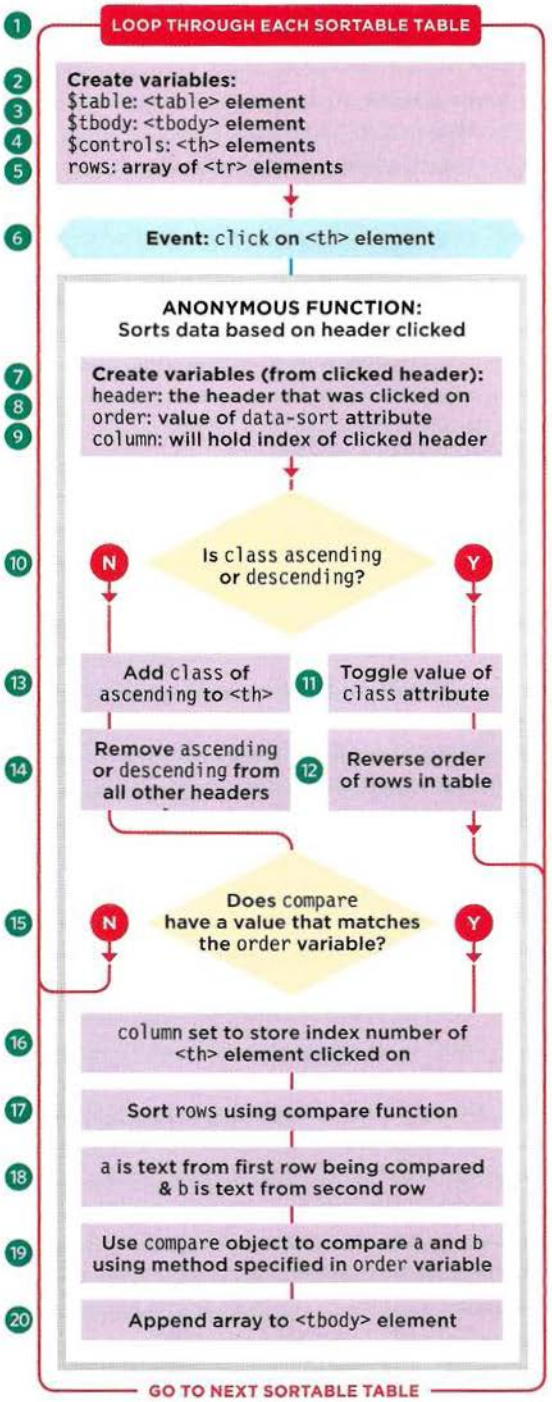
The `replace()` method is used to remove any instances of *The* from the start of a string. `replace()` works on any string and it takes one argument: a regular expression (see p612). It is helpful when *The* is not always used in a name, e.g., for band names or film titles. The regular expression is the first parameter of `replace()` method.

- The string you are looking for is shown between the forward slash characters.
- The caret ^ indicates that *the* must be at the start of the string.
- The **space** after *the* indicates there must be a space after it.
- The `i` indicates that the test is case insensitive.

When a match for the regular expression is found, the second parameter specifies what should take its place. In this case it is an empty string.

SORTING COLUMNS

1. For each element that has a `class` attribute with a value of `sortable`, run the anonymous function.
2. Store the current `<table>` in `$table`.
3. Store the table body in `$tbody`.
4. Store the `<th>` elements in `$controls`.
5. Put each row in `$tbody` into an array called `rows`.
6. Add an event handler for when users click on a header. It should call an anonymous function.
7. `$header` stores that element in a jQuery object.
8. Store the value of that heading's `data-sort` attribute in an variable called `order`.
9. Declare a variable called `column`.
10. In the header the user clicked upon, if the `class` attribute has a value of `ascending` or `descending`, then it is already sorted by this column.
11. Toggle the value of that `class` attribute (so that it shows the other value `ascending`/`descending`).
12. Reverse the rows (stored in the `rows` array) using the `reverse()` method of the array.
13. Otherwise, if the row the user clicked on was not selected, add a `class` of `ascending` to the header.
14. Remove the class of `ascending` or `descending` from all other `<th>` elements on this table.
15. If the `compare` object has a method that matches the value of the `data-type` attribute for this column:
16. Get the column number using the `index()` method (it returns the index number of the element within a jQuery matched set). That value is stored in the `column` variable.
17. The `sort()` method is applied to the array of rows and will compare two rows at a time. As it compares these values:
18. The values a and b are stored in variables:
`.find()` gets the `<td>` elements for that row.
`.eq()` looks for the cell in the row whose index number matches the `column` variable.
`.text()` gets the text from that cell.
19. The `compare` object is used to compare a and b. It will use the method specified in the `type` variable (which was collected from the `data-sort` attribute in step **6**).
20. Append the rows (stored in the `rows` array) to the table body.



SORTABLE TABLE SCRIPT

JAVASCRIPT

c12/js/sort-table.js

```
① $('.sortable').each(function() {
②   var $table = $(this);                        // This sortable table
③   var $tbody = $table.find('tbody');           // Store table body
④   var $controls = $table.find('th');           // Store table headers
⑤   var rows = $tbody.find('tr').toArray();     // Store array containing rows

⑥   $controls.on('click', function() {           // When user clicks on a header
⑦     var $header = $(this);                     // Get the header
⑧     var order = $header.data('sort');          // Get value of data-sort attribute
⑨     var column;                                // Declare variable called column

      // If selected item has ascending or descending class, reverse contents
⑩     if ($header.is('.ascending') || $header.is('.descending')) {
⑪       $header.toggleClass('ascending descending');  // Toggle to other class
⑫       $tbody.append(rows.reverse());                // Reverse the array
      } else {                                        // Otherwise perform a sort
⑬       $header.addClass('ascending');                // Add class to header
        // Remove asc or desc from all other headers
⑭       $header.siblings().removeClass('ascending descending');
⑮       if (compare.hasOwnProperty(order)) {      // If compare object has method
⑯         column = $controls.index(this);         // Search for column's index no

⑰         rows.sort(function(a, b) {              // Call sort() on rows array
⑱           a = $(a).find('td').eq(column).text();// Get text of column in row a
            b = $(b).find('td').eq(column).text();// Get text of column in row b
⑲           return compare[order](a, b);          // Call compare method
          });

⑳         $tbody.append(rows);
        }
      }
    });
});
```

SUMMARY

FILTERING, SEARCHING & SORTING

- Arrays are commonly used to store a set of objects.
- Arrays have helpful methods that allow you to add, remove, filter, and sort the items they contain.
- Filtering lets you remove items and only show a subset of them based on selected criteria.
- Filters often rely on custom functions to check whether items match your criteria.
- Search lets you filter based upon data the user enters.
- Sorting allows you to reorder the items in an array.
- If you want to control the order in which items are sorted, you can use a compare function.
- To support older browsers, you can use a shim script.

13

FORM ENHANCEMENT & VALIDATION

Forms allow you to collect information from visitors, and JavaScript can help you get the right information from them.

Since JavaScript was created, it has been used to enhance and validate forms. Enhancements make forms easier to use. Validation checks whether the user has provided the right information before submitting the form (if not, it provides feedback to the user). This chapter is divided into the following three sections:

FORM ENHANCEMENT

This section features many examples of form enhancement. Each one introduces the different properties and methods you can use when working with form elements.

HTML5 FORM ELEMENTS

HTML5 contains validation features that do not use JavaScript. This section addresses ways in which you can offer validation to old and new browsers in a consistent way.

FORM VALIDATION

The final, and longest, example in the book shows a script that validates (and enhances) the registration form that you can see on the right-hand page. It has over 250 lines of code.

The first section of this chapter also drops jQuery in favor of plain JavaScript, because you should not always rely upon jQuery (especially for scripts that use little of its functionality).


Rizzoli
présente
8 1/2
de FEDERICO FELLINI
LA VOIX DU NORD
RAVITA
TOUT POUR
SUPER 8
FILM SOCIETY
Become a Member
Settings
Profile
FILM SPLICER
OXFORD 600

HELPER FUNCTIONS

The first section of this chapter uses plain JavaScript, no jQuery. We will create our own JavaScript file to handle cross-browser issues, it will contain one helper function to create events.

Forms use a lot of event handlers and (as you saw in Chapter 6) IE5-8 used a different event model than other browsers. You can use jQuery to deal with cross-browser event handling. But, if you do not want to include the entire jQuery script just to handle events in older version of IE, then you need to write your own fallback code to handle the events.

Instead of writing the same fallback code every time you need an event handler, you can write the fallback code once in a **helper function**, and then call that function every time you need to add an event handler to a page.

On the right-hand page you can see a function called `addEvent()`. It lives in a file called `utilities.js`. Once that file has been included in the HTML page, any scripts included *after* it will be able to use this function to create cross-browser event handler:

```
addEvent(el, event, callback);
         ⓘ    ⓘⓘ     ⓘⓘⓘ
```

The function takes three parameters:
i) *el* is a DOM node representing the element that the event will be added to or removed from.
ii) *event* is the type of event being listened for.
iii) *callback* is the function that is to be run when the event is triggered on that element.

The `utilities.js` file on the website also has a method to remove events.

If you look inside the `addEvent()` method on the right-hand page, a conditional statement checks whether the browser supports `addEventListener()`. If it does, a standard event listener will be added. If not, then the IE fallback will be created.

The fallback addresses three points:

- It uses IE's the `attachEvent()` method.
- In IE5-8, the `event` object is not automatically passed into the event-handling function (and is not available via the `this` keyword) see p264. Instead it is available on the `window` object. So the code must pass the `event` object into the event handler as a parameter.
- When you pass parameters to an event-handling function, the call must be wrapped in an anonymous function see p256.

To achieve this, the fallback adds two methods to the element the event handler will be placed upon (see steps 5 and 6 on the right-hand page). It then uses IE's `attachEvent()` method to add the event handler code to the element.

The functions demonstrate two new techniques:

- **Adding new methods to DOM nodes:** You can add methods to DOM nodes because they are just objects (that represent elements).
- **Creating method names using a variable:** Square brackets can be used to set a property/ method, their content is evaluated into a string.

UTILITIES FILE

Here, you can see the `addEvent()` function that will be used to create all of the event handlers in this chapter. It lives in a file called `utilities.js`.

These reusable functions are often referred to as **helper functions**. As you write more JavaScript, you are increasingly likely to create this type of function.

JAVASCRIPT

c13/js/utilities.js

```
// Helper function to add an event listener
function addEvent(el, event, callback) {
  if ('addEventListener' in el) {                  // If addEventListener works
    el.addEventListener(event, callback, false);// Use it
  } else {                                         // Otherwise
    el['e' + event + callback] = callback;         // Make callback a method of el
    el[event + callback] = function() {            // Add second method
      el['e' + event + callback](window.event); // Use it to call prev func
    };
    el.attachEvent('on' + event, el[event + callback]); // Use attachEvent()
  }           // to call the second function, which then calls the first one
}
```

1. The `addEvent()` function is declared with three parameters: element, event type, callback function.
2. A conditional statement checks if the element supports the `addEventListener()` method.
3. If it does, then `addEventListener()` is used.
4. If not, the fallback code will run instead.

The fallback must add two methods to the element the event handler will be placed upon. It then uses Internet Explorer's `attachEvent()` method to call them when the event occurs on that element.

5. The first method added to the element is the code that should run when the event occurs on this element (it was the third parameter of the function).
6. The second method calls the method from the previous step. It is needed in order to pass the `event` object to the function in step 5.
7. The `attachEvent()` method is used to listen for the specified event, on the specified element. When the event fires, it calls the method that it added in step 6, which in turn calls the method in step 5 using the correct reference to the `event` object.

In steps 5 and 6, square bracket notation is used to add a method name to an element:

```
el['e' + event + callback]
 i          ii
```

i) The DOM node is stored in `el`. The square brackets add the method name to that node. That method name must be unique to that element, so it is created using three pieces of information.

ii) The method names are made up of:

- The letter e (used for the first method in step 5 but not used in step 6)
- The event type (e.g., `click`, `blur`, `mouseover`)
- The code from the callback function

In the code on the right-hand page, the value of this method is the callback function. (This could lead to a long method name, but it serves the purpose.) This function is based on one by John Resig, who created jQuery (`http://ejohn.org/projects/flexible-javascript-events/`).

THE FORM ELEMENT

DOM nodes for form controls have different properties, methods, and events than some of the other elements you have met so far. Here are some you should note for the `<form>` element.

PROPERTY	DESCRIPTION
`action`	The URL the form is submitted to
`method`	If it is to be sent via GET or POST
`name`	Rarely used, more common to select a form by the value of its `id` attribute
`elements`	A collection of the elements in the form that users can interact with. They can be accessed via index numbers or the values of their `name` attributes.

METHOD	DESCRIPTION
`submit()`	This has the same effect as clicking the submit button on a form
`reset()`	Resets the form to the initial values it had when the page loaded

EVENT	DESCRIPTION
`submit`	Fires when the form is submitted
`reset`	Fires when the form is reset

The DOM methods you saw in Chapter 5, such as `getElementById()`, `getElementsByTagName()`, and `querySelector()`, are the most popular techniques for accessing both the `<form>` element and the form controls within any form. However, the `document` object also has something called the **forms collection**. The forms collection holds a reference to each of the `<form>` elements that appear on a page.

Each item in a collection is given an index number (a number starting at 0, like an array). This would access the second form using its index number:

```
document.forms[1];
```

You can also access a form using the value of its `name` attribute. The following would select a form whose `name` attribute has a value of `login`:

```
document.forms.login
```

Each `<form>` element in the page also has an **`elements` collection**. It holds all of the form controls within that form. Each item in the `elements` collection can also be accessed by index number or by the value of its `name` attribute.

The following would access the *second* form on the page and then select the *first* form control within it:

```
document.forms[1].elements[0];
```

The following would access the *second* form on the page, then select the element whose `name` attribute had a value of `password` from that form:

```
document.forms[1].elements.password;
```

Note: index numbers in a collection of elements can change if the markup of a page is altered. So, use of index numbers ties a script to the HTML markup (- it does not achieve a separation of concerns).

FORM CONTROLS

Each type of form control uses a different combination of the properties, methods, and events shown below. Note that the methods can be used to simulate how a user would interact with the form controls.

PROPERTY	DESCRIPTION
`value`	In a text input, it is the text the user entered; otherwise, it is the value of the `value` attribute
`type`	When a form control has been created using the `<input>` element, this defines the `type` of the form element (e.g., `text`, `password`, `radio`, `checkbox`)
`name`	Gets or sets the value of the `name` attribute
`defaultValue`	The initial value of a text box or text area when the page is rendered
`form`	The form that the control belongs to
`disabled`	Disables the `<form>` element
`checked`	Indicates which checkbox or radio buttons have been checked. This property is a Boolean; in JavaScript it will have a value of `true` if checked
`defaultChecked`	Whether the checkbox or radio button was checked or not when the page loaded (Boolean)
`selected`	Indicates that an item from a select box has been selected (Boolean - `true` if selected)

METHOD	DESCRIPTION
`focus()`	Gives an element focus
`blur()`	Removes focus from an element
`select()`	Selects and highlights text content of an element, (e.g., text inputs, text areas, and passwords)
`click()`	Triggers a `click` event upon buttons, checkboxes, and file upload Also triggers a `submit` event on a submit button, and the `reset` event on a reset button

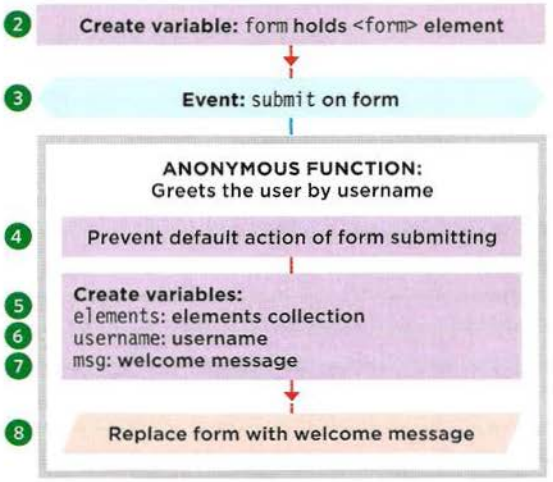
EVENT	DESCRIPTION
`blur`	When the user leaves a field
`focus`	When the user enters a field
`click`	When the user clicks on an element
`change`	When the value of an element changes
`input`	When the value of an `<input>` or `<textarea>` element changes
`keydown, keyup, keypress`	When the user interacts with a keyboard

SUBMITTING FORMS

In this example, a basic login form lets users enter a username and password. When the user submits the form, a welcome message will replace the form. On the right-hand page you can see both the HTML and the JavaScript for this example.

1. Place the script in an Immediately Invoked Function Expression (IIFE see p97). (This is not shown in the flowchart.)
2. A variable called `form` is created and it is set to hold the `<form>` element. It is used in the event listener in the next line of code.
3. An event listener triggers an anonymous function when the form is submitted. Note how this is set using the `addEvent()` function that was created in the `utilities.js` file that you saw on p571.
4. To prevent the form being sent (and to allow this example to show a message to the user) the `preventDefault()` method is used on the form.
5. The collection of elements in this form is stored in a variable called `elements`.
6. To get the username, first select the username input from the `elements` collection using the value of its `name` attribute. Then, to get the text the user entered, the `value` property of that element is used.
7. A welcome message is created and stored in a variable called `msg`; this message will incorporate the username that the visitor entered.
8. The message replaces the form within the HTML.

In the HTML page, the `utilities.js` file you saw on p571 is included before the `submit-event.js` script because its `addEvent()` function is used to create the event handlers for this example. `utilities.js` is included for all examples in this section.



The event listener waits for the `submit` event on the form (rather than a `click` on the submit button) because the form can be submitted in other ways than clicking on the submit button. For example, the user might press the Enter key.

THE SUBMIT EVENT & GETTING FORM VALUES

HTML

c13/submit-event.html

```
<form id="login" action="/login" method="post">...
  <div class="two-thirds column" id="main">
    <fieldset>
      <legend>Login</legend>
      <label for="username">Username:</label>
      <input type="text" id="username" name="username" />
      <label for="pwd">Password:</label>
      <input type="password" id="pwd" name="pwd" />
      <input type="submit" value="Login" />
    </fieldset>
  </div> <!-- .two-thirds -->
</form> ...
<script src="js/utilities.js"></script>
<script src="js/submit-event.js"></script>
```

JAVASCRIPT

c13/js/submit-event.js

```
① (function(){
②   var form = document.getElementById('login');        // Get form element

③   addEvent(form, 'submit', function(e) {              // When user submits form
④     e.preventDefault();                               // Stop it being sent
⑤     var elements = this.elements;                     // Get all form elements
⑥     var username = elements.username.value;           // Select username entered
⑦     var msg      = 'Welcome ' + username;             // Create welcome message
⑧     document.getElementById('main').textContent = msg; // Write welcome message
  });
}());
```

When selecting a DOM node, if you are likely to use it again, it should be cached. On the right, you can see a variation of the above code, where the username and the `main` element have both been stored in variables outside of the event listener. If the user had to resubmit the form, the browser would not have to make the same selections again.

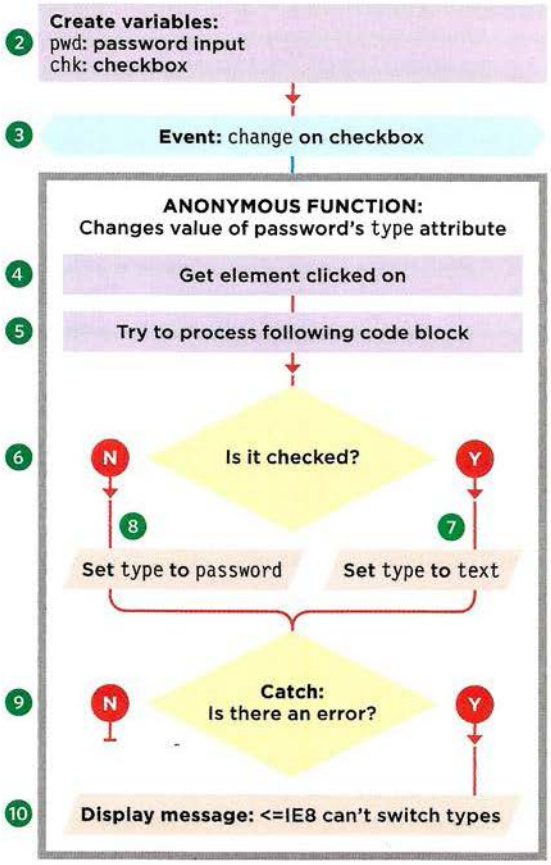
```
var form = document.getElementById('login');
var elements = form.elements;
var elUsername = elements.username;
var elMain = document.getElementById('main');
addEvent(form, 'submit', function(e) {
  e.preventDefault();
  var msg = 'Welcome ' + elUsername.value;
  elMain.textContent = msg;
});
```

CHANGING TYPE OF INPUT

This example adds a checkbox under the password input. If the user checks that box, their password will become visible. It works by using JavaScript to change the `type` property of the input from `password` to `text`. (The `type` property in the DOM corresponds to `type` attribute in the HTML.)

Changing the `type` property causes an error in IE8 (and earlier), so this code is placed in a `try... catch` statement. If the browser detects an error, the script continues to run the second code block.

1. Place the script in an IIFE (not shown in flowchart).
2. Put password input and checkbox in variables.
3. An event listener triggers an anonymous function when the show password checkbox is changed.
4. The target of the event (the checkbox) is stored in a variable called target. As you saw in Chapter 6, `e.target` will retrieve this for most browsers. `e.srcElement` is only used for old versions of IE.
5. A `try... catch` statement checks if an error is caused when the `type` attribute is updated.
6. If the checkbox is selected:
7. The value of the password input's `type` attribute is set to `text`.
8. Otherwise, it is set to `password`.
9. If trying to change the type causes an error, the `catch` clause runs another code block instead.
10. It shows a message to tell the user.



As you saw in Chapter 10, an error can stop a script from running. If you know something may cause an error for some browsers, placing that code in a `try... catch` statement lets the interpreter continue with an alternative set of code.

SHOWING A PASSWORD

HTML

c13/input-type.html

```
<fieldset>
  <legend>Login</legend>
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" />
  <label for="pwd">Password:</label>
  <input type="password" id="pwd" name="pwd" />
  <input type="checkbox" id="showPwd">
  <label for="showPwd">show password</label>
  <input type="submit" value="Login" />
</fieldset> ...
<script src="js/utilities.js"></script>
<script src="js/input-type.js"></script>
```

JAVASCRIPT

c13/js/input-type.js

```
① (function(){

②   var pwd = document.getElementById('pwd');      // Get password input
    var chk = document.getElementById('showPwd'); // Get checkbox

③   addEvent(chk, 'change', function(e) {          // When user clicks on checkbox
④     var target = e.target || e.srcElement;       // Get that element
⑤     try {                                        // Try the following code block
⑥       if (target.checked) {                      // If the checkbox is checked
⑦         pwd.type = 'text';                       // Set pwd type to text
        } else {                                   // Otherwise
⑧         pwd.type = 'password';                   // Set pwd type to password
        }
⑨     } catch(error) {                             // If this causes an error
⑩       alert('This browser cannot switch type'); // Say 'cannot switch type'
      }
    });

  }());
```

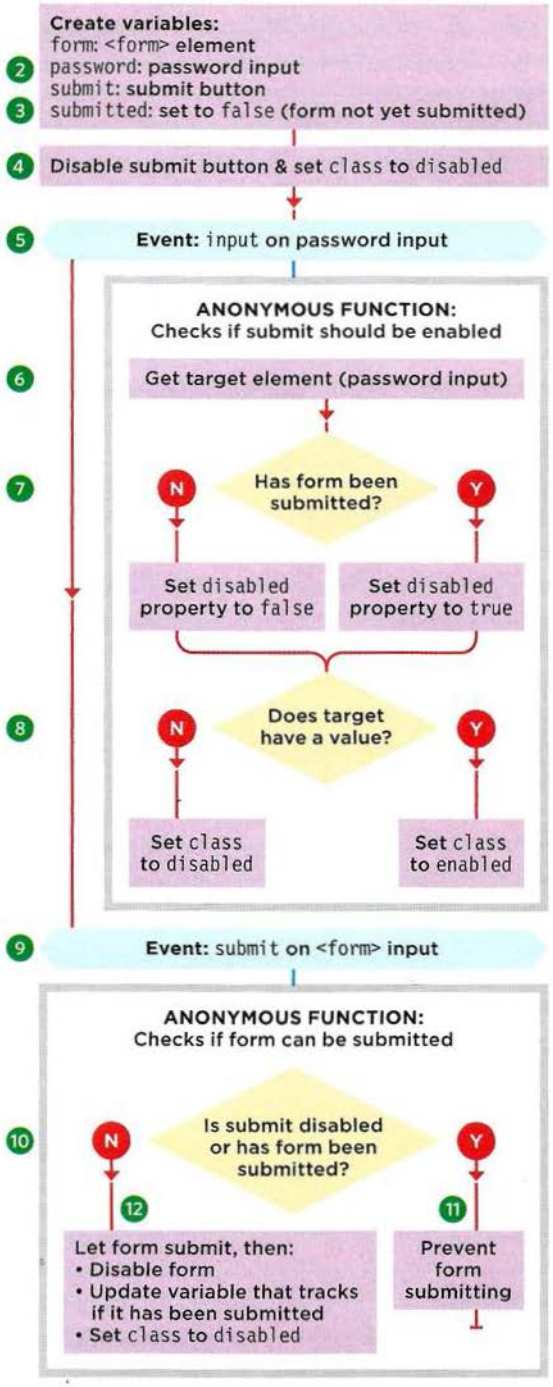
SUBMIT BUTTONS

This script disables the submit button when:

- The script first loads. The `change` event then checks when the password changes and enables submit if the password is given a value.
- The form has been submitted (to prevent the form being sent multiple times).

The button is disabled using the `disabled` property. It corresponds with the HTML `disabled` attribute, and can be used to disable any form elements that a user can interact with. A value of `true` disables the button; `false` lets the user click on it.

1. Place the script in an IIFE (not shown in flowchart).
2. Store the form, password input, and submit button in variables.
3. The `submitted` variable is known as a **flag**; it remembers if the form has been submitted yet.
4. The submit button is disabled at the start of the script (rather than in the HTML) so that the form can still be used if a visitor has JavaScript disabled.
5. An event listener waits for the `input` event on the password input; it triggers an anonymous function.
6. Store the target of the event in `target`.
7. If the password input has a value, the submit button is enabled, and (**8**) its style updated.
9. A second event listener checks for when the user submits the form (and runs an anonymous function).
10. If the submit button is disabled, or the form has been submitted, the subsequent code block is run.
11. The default action of the form (submitting) is prevented, and `return` leaves the function.
12. If step 11 did not run, the form is submitted, the submit button disabled, the `submitted` variable updated with a value of `true`, and its `class` updated.



DISABLE SUBMIT BUTTON

HTML

c13/disable-submit.html

```
<label for="pwd">New password:</label>
<input type="password" id="pwd" />
<input type="submit" id="submit" value="submit" />
```

JAVASCRIPT

c13/js/disable-submit.js

```
(function(){
  var form      = document.getElementById('newPwd');  // The form
  var password  = document.getElementById('pwd');     // Password input
  var submit    = document.getElementById('submit');  // Submit button

  var submitted = false;                              // Has form been submitted?

  submit.disabled = true;                             // Disable submit button
  submit.className = 'disabled';                      // Style submit button

  // On input: Check whether or not to enable the submit button
  addEvent(password, 'input', function(e) {           // On input of password
    var target = e.target || e.srcElement;            // Target of event
    submit.disabled = submitted || !target.value;     // Set disabled property
    // If form has been submitted or pwd has no value set CSS to disabled
    submit.className = (!target.value || submitted ) ? 'disabled' : 'enabled';
});

  // On submit: Disable the form so it cannot be submitted again
  addEvent(form, 'submit', function(e) {              // On submit
    if (submit.disabled || submitted) {               // If disabled OR sent
      e.preventDefault();                             // Stop form submission
      return;                                         // Stop processing function
    }                                                 // Otherwise continue...
    submit.disabled = true;                           // Disable submit button
    submitted = true;                                 // Update submitted var
    submit.className = 'disabled';                    // Update style

    // Demo purposes only: What would have been sent & show submit is disabled
    e.preventDefault();                               // Stop form submitting
    alert('Password is ' + password.value);           // Show the text
  });
}());
```

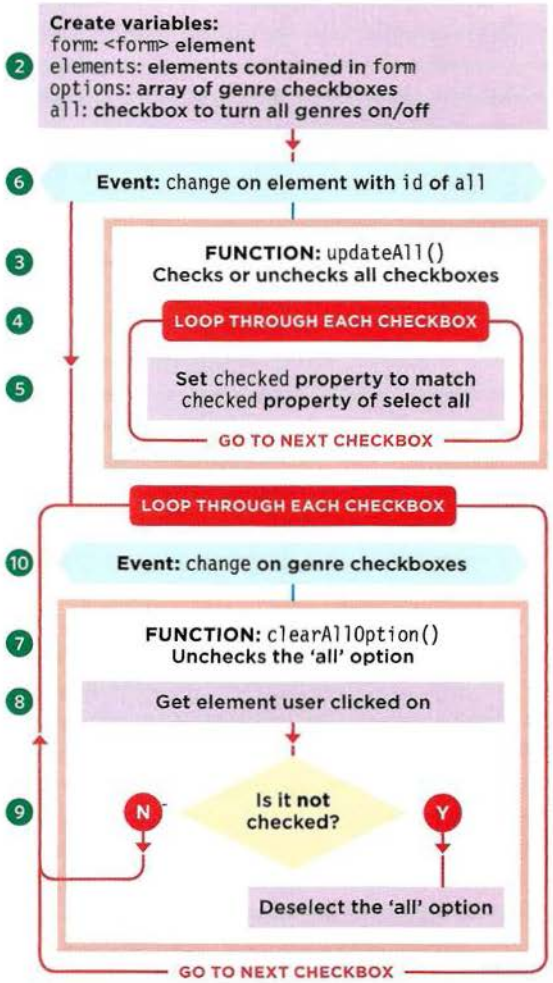
CHECKBOXES

This example asks users about their interests. It has an option to select or deselect all of the checkboxes. It has two event handlers:

- The first fires when the ***all*** checkbox is selected; it loops through the options, updating them.
- The second fires when the ***options*** change; if one is deselected, the *all* option must be deselected.

You can use the `change` event to detect when the value of a checkbox, radio button, or select box changes. Here, it is used to tell when the user selects / deselects a checkbox. The checkboxes can be updated using the `checked` property, which corresponds with HTML's `checked` attribute.

1. Place the script in an IIFE (not shown in flowchart).
2. The form, all of the form elements, the options, and the *all* checkbox are stored in variables.
3. The `updateAll()` function is declared.
4. A loop runs through each of the options.
5. For each one, the `checked` property is set to the same value as the `checked` property on the *all* option.
6. An event listener waits for the user to click on the *all* checkbox, which fires a `change` event and calls the `updateAll()` function.
7. The `clearAllOption()` function is defined.
8. It gets the target of the option the user clicked on.
9. If that option is deselected, then the *all* option is also deselected (as they are no longer all selected).
10. A loop runs through the options, adding an event listener. When the change event happens on any of them, `clearAllOption()` is called.



SELECT ALL CHECKBOXES

HTML

c13/all-checkboxes.html

```
<label><input type="checkbox" value="all" id="all">All</label>
<label><input type="checkbox" name="genre" value="animation">Animation</label>
<label><input type="checkbox" name="genre" value="docs">Documentary</label>
<label><input type="checkbox" name="genre" value="shorts">Shorts</label>
```

JAVASCRIPT

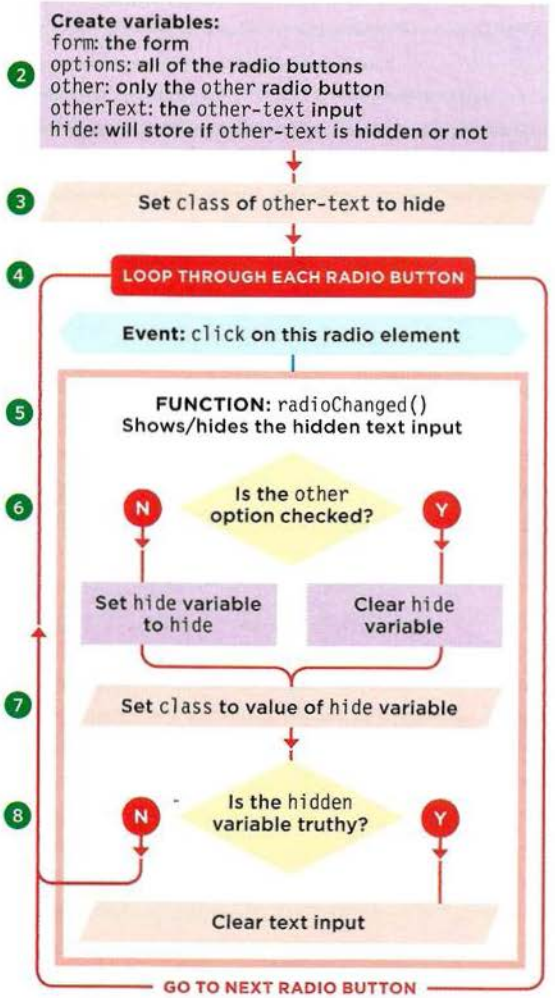
c13/js/all-checkboxes.js

```
(function(){
  var form     = document.getElementById('interests'); // Get form
  var elements = form.elements;                        // All elements in form
  var options  = elements.genre;                       // Array: genre checkboxes
  var all      = document.getElementById('all');       // The 'all' checkbox

  function updateAll() {
    for (var i = 0; i < options.length; i++) {         // Loop through checkboxes
      options[i].checked = all.checked;                // Update checked property
    }
  }
  addEvent(all, 'change', updateAll);                  // Add event listener

  function clearAllOption(e) {
    var target = e.target || e.srcElement;             // Get target of event
    if (!target.checked) {                             // If not checked
      all.checked = false;                             // Uncheck 'All' checkbox
    }
  }
  for (var i = 0; i < options.length; i++) {           // Loop through checkboxes
    addEvent(options[i], 'change', clearAllOption);    // Add event listener
  }

}());
```

RADIO BUTTONS

This example lets users say how they heard about a website. Every time the user selects a radio button, the code checks if the user selected the option that says *other*, and one of two things happens:

- If *other* is selected, a text input is shown so they can add further detail.
- If the first two options are selected, the text box is hidden and its value is emptied.



1. Place the script in an IIFE (not shown in flowchart).
2. The code starts out by setting up variables to hold the form, all radio buttons, the radio button for the *other* option, and the text input.
3. The text input is hidden. This uses JavaScript to update the `class` attribute so that the form still works if the user has JavaScript disabled.
4. Using a `for` loop, an event listener is added to each of the radio buttons. When one of them is clicked, the `radioChanged()` function is called.
5. The `radioChanged()` function is declared.
6. If `other` is checked, the value of the `hide` variable is set to be a blank string, otherwise it is set to `hide`.
7. The `hide` variable is, in turn, used to set the value of the `class` attribute on the text input. If it is blank, the other option is shown; if it has a value of `hide`, the text input is hidden.
8. If the `hide` attribute has a value of `hide`, then the contents of the text input are emptied (so that the text input is blank if it is shown).

RADIO BUTTONS

HTML

c13/show-option.html

```
<form id="how-heard" action="/heard" method="post">
  ...
  <input type="radio" name="heard" value="search" id="search" />
  <label for="search">Search engine</label><br>

  <input type="radio" name="heard" value="print" id="print" />
  <label for="print">Newspaper or magazine</label><br>

  <input type="radio" name="heard" value="other" id="other" />
  <label for="other">Other</label><br>
  <input type="text" name="other-input" id="other-text" />

  <input id="submit" type="submit" value="submit" />
  ...
</form>
```

JAVASCRIPT

c13/js/show-option.js

```
(function(){
  var form, options, other, otherText, hide;          // Declare variables
  form      = document.getElementById('how-heard');   // Get the form
  options   = form.elements.heard;                    // Get the radio buttons
  other     = document.getElementById('other');       // Other radio button
  otherText = document.getElementById('other-text');  // Other text input
  otherText.className = 'hide';                       // Hide other text input

  for (var i = [0]; i < options.length; i++) {        // Loop through radios
    addEvent(options[i], 'click', radioChanged);      // Add event listener
  }

  function radioChanged() {
    hide = other.checked ? '' : 'hide';               // Is other checked?
    otherText.className = hide;                       // Text input visibility
    if (hide) {                                       // If text input hidden
      otherText.value = '';                           // Empty its contents
    }
  }
}());
```

SELECT BOXES

The <select> element is more complex than the other form controls. Its DOM node has a number of extra properties and methods. Its <option> elements contain the values a user can select.

This example features two select boxes. When the user selects an option from the first select box, the contents of the second select box are updated with corresponding options.

In the first select box, users can choose to rent a camera or a projector. When they make their choice, a list of options are shown in the second select box. Because this example is a bit more complex than the ones you have seen so far in this chapter, the HTML and screen shots are shown to the right, and the JavaScript file is discussed on p586-p587.

When the user selects an option from the drop-down list, the change event fires. This event is often used to trigger scripts when the user changes the value of a select box.

The <select> element also has some extra properties and methods that are specific to it; these are shown in the tables below.

If you want to work with the individual options the user can select from, a collection of <option> elements is available.

PROPERTY	DESCRIPTION
options	A collection of all the <option> elements
selectedIndex	Index number of the option that is currently option
length	Number of options
multiple	Allows users to select multiple options from the select box (Rarely used because the user-experience is not very good)
selectedOptions	A collection of all the selected <option> elements

METHOD	DESCRIPTION
add(*option*, *before*)	Adds an item to the list: The first parameter is the new option; the second is the element it should go before If no value is given, the item will be added to the end of the options
remove(*index*)	Removes an item from the list: Has only one parameter – the index number of the option to be removed

SELECT BOXES

HTML

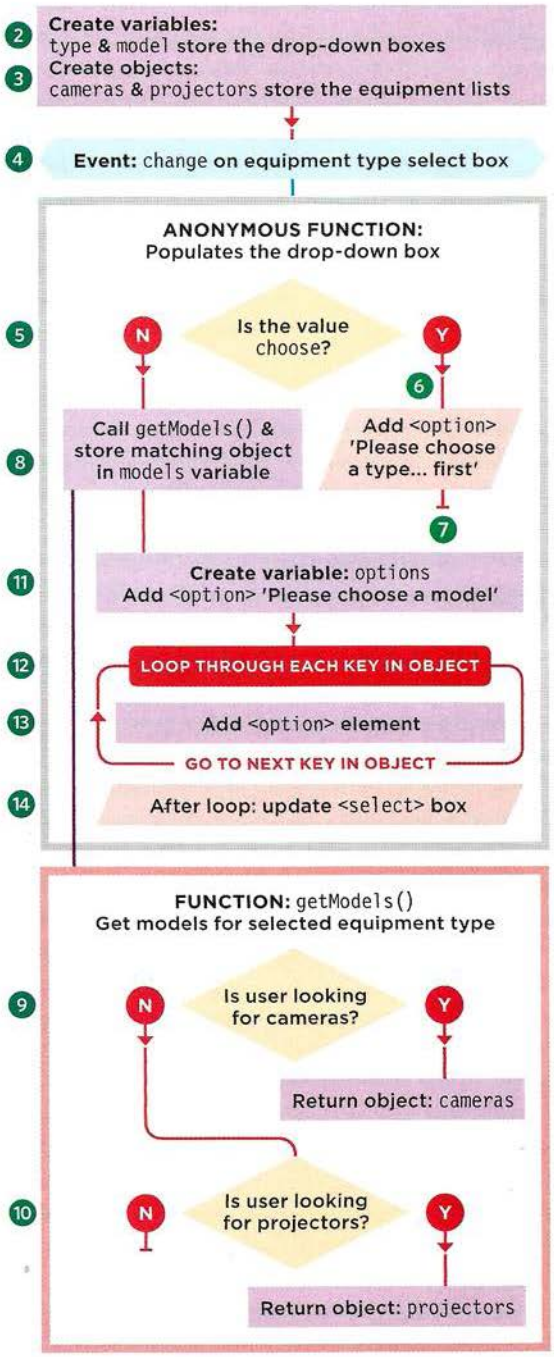
c13/populate-selectbox.html

```
<label for="equipmentType">type</label>
<select id="equipmentType" name="equipmentType">
  <option value="choose">Please choose a type</option>
  <option value="cameras">camera</option>
  <option value="projectors">projector</option>
</select><br>

<label for="model">model</label>
<select id="model" name="model">
  <option>Please choose a type first</option>
</select>

<input id="submit" type="submit" value="submit" />
```

RESULT

SELECT BOXES

1. Place the script in an IIFE (not shown in flowchart).
2. Variables hold the two select boxes.
3. Two objects are created; each one holds options used to populate the second select box (one has types of cameras, the other has types of projectors).
4. When the user changes the first select box, an event listener triggers an anonymous function.
5. The anonymous function checks if the first select box has a value of `choose`.
6. If so, the second select box is updated with just one option, which tells the user to select a type.
7. No further processing is needed, and the `return` keyword exits the anonymous function (until the user changes the first select box again).
8. If a type of equipment has been selected, the anonymous function continues to run, and a `models` variable is created. It will store one of the objects defined in step 3 (cameras or projectors). This correct object is retrieved using the `getModels()` function declared at the end of the script **(9+10)**. The function takes one parameter `this.value`, which corresponds to the value from the option that was selected in first select box.
9. Inside the `getModels()` function, an if statement checks if the value passed in was `cameras`; if so, it returns the `cameras` object.
10. If not, it continues to run, checking to see if the value was `projectors`, and if so, it returns the `projectors` object.
11. A variable called `options` is created. It will hold all the `<option>` elements for the second select box. When this variable is created the first `<option>` is added to it; it tells users to choose a model.
12. A `for` loop goes through the contents of the object that was placed in the `models` variable in step **(8-10)**. Inside the loop, `key` refers to the individual items in the object.
13. Another `<option>` element is created for every item in the object. Its `value` attribute uses the property name from the object. The content that sits between the `<option>` tags is that property's value.
14. The options are then added to the second select box using the `innerHTML` property.



SELECT BOXES

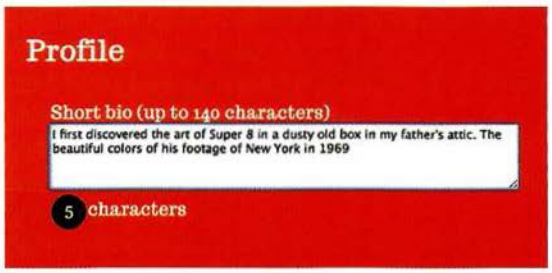
JAVASCRIPT

c13/js/populate-selectbox.js

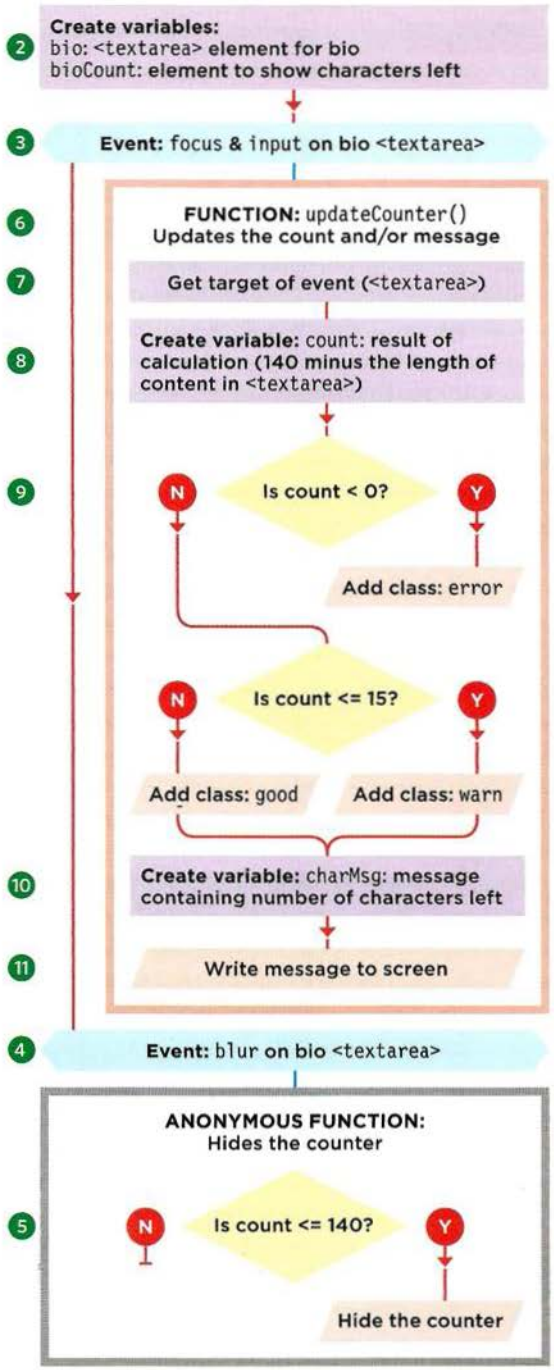
```
(function(){
  var type  = document.getElementById('equipmentType');// Type select box
  var model = document.getElementById('model');        // Model select box
  var cameras = {                                      // Object stores cameras
    bolex: 'Bolex Paillard H8',
    yashica: 'Yashica 30',
    pathescape: 'Pathescape Super-8 Relax',
    canon: 'Canon 512'
  };
  var projectors = {                                   // Store projectors
    kodak: 'Kodak Instamatic M55',
    bolex: 'Bolex Sound 715',
    eumig: 'Eumig Mark S',
    sankyo: 'Sankyo Dualux'
  };

  // WHEN THE USER CHANGES THE TYPE SELECT BOX
  addEvent(type, 'change', function() {
    if (this.value === 'choose') {                     // No selection made
      model.innerHTML = '<option>Please choose a type first</option>';
      return;                                          // No need to proceed further
    }
    var models = getModels(this.value);                // Select the right object

    // LOOP THROUGH THE OPTIONS IN THE OBJECT TO CREATE OPTIONS
    var options = '<option>Please choose a model</option>';
    for (var key in models) {                          // Loop through models
      options += '<option value="' + key + '">' + models[key] + '</option>';
    } // If an option could contain a quote, key should be escaped
    model.innerHTML = options;                         // Update select box
  });

  function getModels(equipmentType) {
    if (equipmentType === 'cameras') {                 // If type is cameras
      return cameras;                                  // Return cameras object
    } else if (equipmentType === 'projectors') {       // If type is projectors
      return projectors;                               // Return projectors object
    }
  }
}());
```

TEXTAREA

In this example, users can enter a biography of up to 140 characters. When the cursor is in the textarea, a element will be shown with a count of how many characters the user has remaining. When the textarea loses focus, this message is hidden.



1. Place the script in an IIFE (not shown in flowchart).
2. The script sets up two variables to hold: a reference to the <textarea> element and a reference to the that holds the message.
3. Two event listeners monitor the <textarea>. The first checks for when the element gains focus; the second checks for a `input` event. Both events trigger a function called `updateCounter()` (6-11) The `input` event does not work in IE8, but you can use `keyup` to support older browsers.
4. A third event listener triggers an anonymous function when the user leaves the <textarea>.
5. If the number of characters is less than or equal to 140 characters, the length of the bio is okay, and it hides the message (because it is not needed when the user is not interacting with the element).
6. The `updateCounter()` function is declared.
7. It gets a reference to the element that called it.
8. A variable called `count` holds the number of characters left to use (it does this by subtracting the number of characters used from 140).
9. `if... else` statements are used to set the CSS class for the element that holds the message (these can also show the message if it was hidden).
10. A variable called `charMsg` is created to store the message that will be shown to the user.
11. The message is added to the page.



CHARACTER COUNTER

HTML

c13/textarea-counter.html

```
  <label for="bio">Short Bio (up to 140 characters)</label>
  <textarea name="bio" id="bio" rows="5" cols="30"></textarea>
  <span id="bio-count" class="hide"></span>
...
<script src="js/utilities.js"></script>
<script src="js/textarea-counter.js"></script>
```

JAVASCRIPT

c13/js/textarea-counter.js

```
① (function () {
②   var bio      = document.getElementById('bio');        // <textarea> element
    var bioCount = document.getElementById('bio-count');  // Character count el

③   addEvent(bio, 'focus', updateCounter);       // Call updateCounter() on focus
    addEvent(bio, 'input', updateCounter);       // Call updateCounter() on input

④   addEvent(bio, 'blur', function () {          // On leaving the element
      if (bio.value.length <= 140) {             // If bio is not too long
⑤       bioCount.className = 'hide';             // Hide the counter
      }
    });

⑥   function updateCounter(e) {
⑦     var target = e.target || e.srcElement;     // Get the target of the event
⑧     var count  = 140 - target.value.length;    // How many characters are left
      if (count < 0) {                           // If less than 0 chars free
        bioCount.className = 'error';            // Add class of error
      } else if (count <= 15) {                  // If less than 15 chars free
⑨       bioCount.className = 'warn';             // Add class of warn
      } else {                                   // Otherwise
        bioCount.className = 'good';             // Add class of good
      }
⑩     var charMsg = '<b>' + count + '</b>' + ' characters'; // Message to display
⑪     bioCount.innerHTML = charMsg;              // Update the counter element
    }

  }());
```

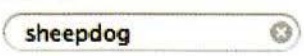
HTML5 ELEMENTS & ATTRIBUTES

HTML5 adds form elements and attributes to perform tasks that had previously been performed by JavaScript. However, their appearance can vary a lot between different browsers (especially their error messages).

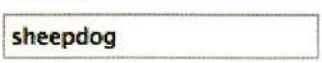
SEARCH

```
<input type="search"
  placeholder="Search..."
  autofocus>
```

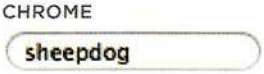
SAFARI



FIREFOX



CHROME

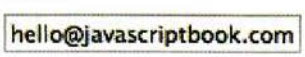


Safari rounds the corners of its search inputs to match the user interface of the operating system. When you enter text, Safari shows a cross icon which, when clicked or tapped, allows the user to clear the text from the field. Other browsers show an input like any other text input.

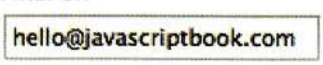
EMAIL, URL, PHONE

```
<input type="email">
<input type="url">
<input type="telephone">
```

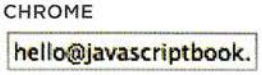
SAFARI



FIREFOX



CHROME



Email, URL, and phone inputs all look like text input fields, but the browser performs checks on the data entered into these inputs to see if it is in the right format to be an email address, URL, or phone number, then shows a message if it is not.

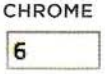
NUMBER

```
<input type="number"
  min="0"
  max="10"
  step="2"
  value="6">
```

SAFARI

FIREFOX

CHROME



Number inputs sometimes add arrows to increase or decrease the number specified (also known as **spinboxes**). You can specify a minimum and a maximum value, a step (or increment), and an initial value. The browser checks that the user entered a number, and shows a message if a number was not entered.

ATTRIBUTE	DESCRIPTION
`autofocus`	Gives focus to this element when the page is loaded
`placeholder`	Content of this attribute is shown in the `<input>` element as a hint (see p594)
`required`	Checks that the field has a value – could be text entered or an option selected (see p606)
`min`	Minimum permitted number
`max`	Maximum permitted number
`step`	Intervals by which numbers should increase or decrease
`value`	Default value for a number when the control first loads on the page
`autocomplete`	On by default: shows list of past entries (disable for credit card numbers / sensitive data)
`pattern`	Lets you to specify a regular expression to validate a value (see p612)
`novalidate`	Used on the `<form>` element to disable the HTML5 built-in form validation (see p604)

RANGE

```
<input type="range"
  min="0"
  max="10"
  step="2"
  value="6">
```

SAFARI

FIREFOX

CHROME

The range input offers another way to specify a number – this time the control shows a **slider**. As with the spinbox, you can specify a minimum and a maximum value, a step, and an initial value.

COLOR PICKER

```
<input type="color">
```

CHROME

At the time of writing, Chrome and Opera are the only browsers to implement a color input. It allows users to specify a color. When they click on the control, the browser will usually show the operating system's default color picker (except for Linux, which offers a more basic palette). It inserts a hex color value based on the user's selection.

DATE

```
<input type="date"> (below)
<input type="month">
<input type="week">
<input type="time">
<input type="datetime">
```

CHROME

There are several different date inputs available. At the time of writing, Chrome was the only browser to have implemented a date picker.

SUPPORT & STYLING

HTML5 form elements are not supported in all browsers and, when they are, the inputs and error messages can look very different.

DESKTOP BROWSERS

At the time of writing, many developers were still using JavaScript instead of these new HTML5 features because:

- Older browsers do not support the new input types (they just show a text box in their place).
- Different browsers present the elements and their error messages in very different ways (and designers often want to give users a consistent experience across browsers).

Below, you can see how the error messages look very different in two of the main browsers.

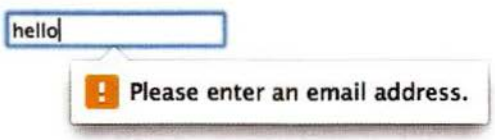
MOBILE

On mobile devices the situation is very different, as most modern mobile browsers:

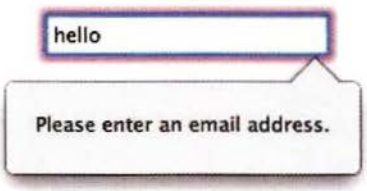
- Support the main HTML5 elements
- Show a keyboard that's adapted to the type: `email` brings up a keyboard with the @ sign `number` type brings up a number keyboard
- Give helpful versions of the date picker

Therefore, in mobile browsers, the new HTML5 types and elements make forms more accessible and usable for your visitors.

ERROR MESSAGE FOR AN EMAIL INPUT IN CHROME:



ERROR MESSAGE FOR AN EMAIL INPUT IN FIREFOX:



DATE INPUT IN IOS:

CURRENT APPROACHES

Until more visitors' browsers support these new features, and do so in a consistent way, developers will think carefully about how they use them.

POLYFILLS

A polyfill is a script that provides functionality you may expect a browser to support by default. For example, because older browsers do not support the new HTML5 elements, polyfills can be used to implement a similar experience / functionality in those older browsers. Typically this is achieved using JavaScript or a jQuery plugin.

Polyfills often come with CSS files that are used to style the functionality the script adds.

You can find a list of polyfills for various features here: `http://html5please.com`

There is an example of how to use a polyfill on p594, where you see how to get the HTML5 `placeholder` attribute to show up in older browsers.

FEATURE DETECTION

Feature detection means checking whether a browser supports a feature or not. You can then decide what to do if a feature is, or is not, supported. On p415 you learned about a script called `modernizr.js`, which tests for browser features.

Commonly, if a feature is not supported, a polyfill script will be loaded to emulate that feature. To save loading the polyfill script into browsers that do not need it, Modernizr includes a **conditional loader**; it will only load a script if the test indicates that the script is needed.

Another popular conditional loader is `Require.js` (available from `http://requirejs.org`), but it is a bit more complex when you are first starting out because it offers many other features.

CONSISTENCY

Many designers and developers want to control the appearance of form controls and error messages to give a consistent experience across all browsers. (Consistency in error messages is considered important because different styles of error messages can confuse users.)

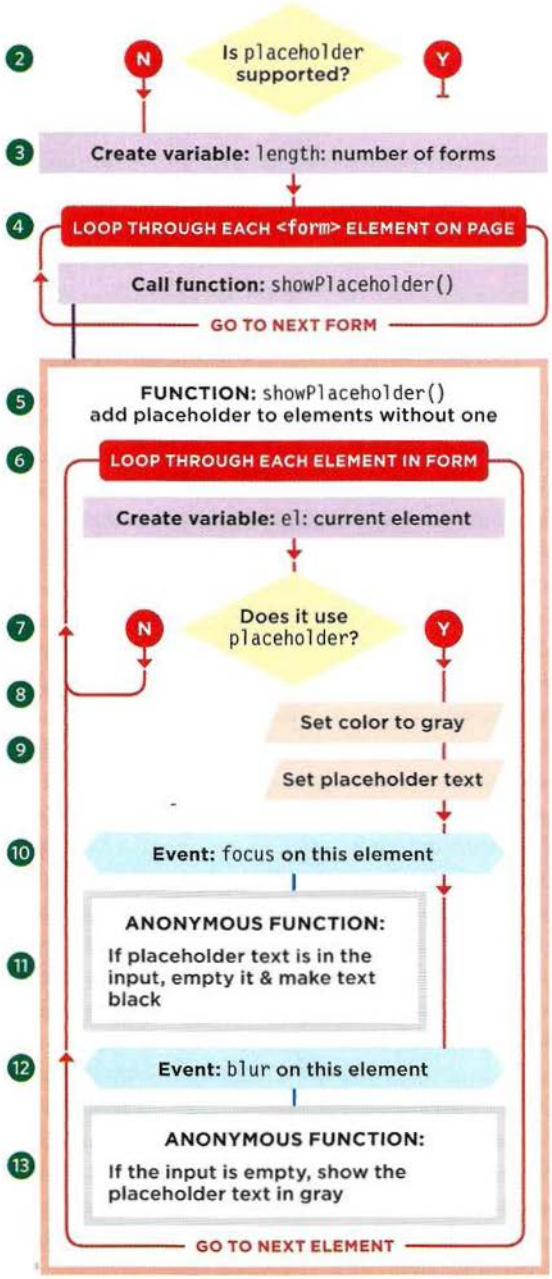
Therefore, the long example used at the end of this chapter will disable HTML5 validation and try to use JavaScript validation as its first choice. (HTML5 validation is only shown if the user does not have JavaScript enabled; it is used as a fallback in modern browsers.)

In that example, you also see jQuery UI used to ensure that the date picker is consistent across all devices, with as little code as possible.

PLACEHOLDER FALLBACK

The HTML5 `placeholder` attribute lets you put words in text inputs (to replace labels or to add hints about what to enter). When the input gains focus and the user starts typing, the text disappears. But it only works in modern browsers, so this script ensures that the user sees placeholder text in older browsers too. It is a basic example of a polyfill.

1. Place the script in an IIFE (not shown in flowchart).
2. Check if the browser supports the HTML5 `placeholder` attribute. If it does, there is no need for the fallback. Use `return` to exit the function.
3. Find out how many forms are on the page using the `length` property of the `forms` collection.
4. Loop through each `<form>` element on the page and call `showPlaceholder()` for each one, passing it the collection of elements in that form.
5. The `showPlaceholder()` function is declared.
6. A `for` loop runs through elements in the collection.
7. An `if` statement checks each element to see if the element has a `placeholder` attribute with a value.
8. If there is no `placeholder` attribute, `continue` tells it to go on to the next element. Otherwise, it:
9. Changes the text color to gray, and sets the value of the element to be the placeholder text.
10. An event listener triggers an anonymous function when the element gains `focus`.
11. If the current value of the element matches the placeholder text, the value is cleared (and color changed to black).
12. An event listener triggers an anonymous function when the element loses focus.
13. If the input is empty, the placeholder text is added back in (and its color changed to gray).



PLACEHOLDER POLYFILL

JAVASCRIPT c13/js/placeholder-polyfill.js

```
(function () {                                          // Place code in an IIFE
  // Test: Create an input element, and see if the placeholder is supported
  if ('placeholder' in document.createElement('input')) {
    return;
  }

  var length = document.forms.length;                   // Get number of forms
  for (var i = 0, l = length; i < l; i++ ) {            // Loop through each one
    showPlaceholder(document.forms[i].elements);        // Call showPlaceholder()
  }

  function showPlaceholder(elements) {                  // Declare function
    for (var i = 0, l = elements.length; i < l; i++) { // For each element
      var el = elements[i];                             // Store that element
      if (!el.placeholder) {                            // If no placeholder set
        continue;                                       // Go to next element
      }                                                 // Otherwise
      el.style.color = '#666666';                       // Set text to gray
      el.value = el.placeholder;                        // Add placeholder text

      addEvent(el, 'focus', function () {               // If it gains focus
        if (this.value === this.placeholder) {          // If value=placeholder
          this.value = '';                              // Empty text input
          this.style.color = '#000000';                 // Make text black
        }
      });

      addEvent(el, 'blur', function () {                // On blur event
        if (this.value === '') {                        // If the input is empty
          this.value = this.placeholder;                // Make value placeholder
          this.style.color = '#666666';                 // Make text gray
        }
      });
    }                                                   // End of for loop
  }                                                     // End showPlaceholder()
}());
```

There are a few differences from the HTML5's placeholder attribute: e.g., if the user deletes their text, the placeholder only returns when the user leaves the input (not immediately – as with some browsers). It will not submit text that has the same value as the placeholder. Placeholder values may be saved by autocomplete.

POLYFILL USING MODERNIZR & YEPNOPE

You met Modernizr in Chapter 9, here you can see it used with a conditional loader so that it only loads a fallback script if one is needed.

Modernizr lets you test whether or not a browser and device support certain features; this is known as feature detection. You can then take different courses of action depending on whether or not the features were supported. For example, if an older browser does not support a feature, you might decide to use a polyfill.

Modernizr is sometimes included in the `<head>` of an HTML page when it needs to perform checks before the page has loaded (for example, some HTML5 / CSS3 polyfills must be loaded before the page).

Rather than loading a polyfill script for everyone who visits your site (even if they do not need to use it), you can use something called a **conditional loader**, which will let you load different files depending on whether a condition returns true or false. Modernizr is commonly used with a conditional loader called `YepNope.js`, so polyfills are only loaded if needed.

Once you have included the YepNope script in your page, you can call the `yepnope()` function. It uses object literal syntax to indicate a condition to test, and then what files to load depending on whether the condition returned `true` or `false`.

MODERNIZR ON ITS OWN

Each feature you test using Modernizr becomes a property of the `Modernizr` object. If the feature is supported, the property contains `true`; if not, it contains `false`. You then use the properties of the `Modernizr` object in a conditional statement as shown below. Here, if `Modernizr`'s `cssanimations` property does not return `true` the code in the curly braces runs.

```
if (!Modernizr.cssanimations) {
  // CSS animations are not supported
  // Use jQuery animation instead
}
```

MODERNIZR + YEPNOPE

YepNope is passed an object literal, which usually contains a minimum of three properties:

- `test` is the a condition being checked. Here Modernizr is used to check if `cssanimations` are supported.
- `yep` is the file to load if the condition returns `true`.
- `nope` is the file to load if the condition returns `false` (here it loads two files using array syntax).

```
yepnope({
  test: Modernizr.cssanimations,
  yep: 'css/animations.css',
  nope: ['js/jquery.js', 'js/animate.js']
});
```

CONDITIONAL LOADING OF A POLYFILL

HTML

c13/number-polyfill.html

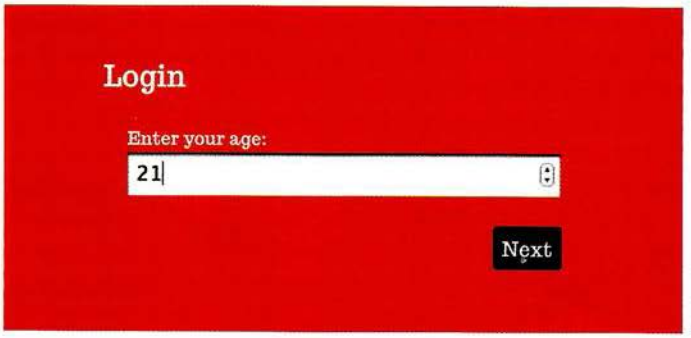
```
<head>
  ...
  <script src="js/modernizr.js"></script>
  <script src="js/yepnope.js"></script>
  <script src="js/number-polyfill-eg.js"></script>
</head>
<body>
  <label for="age">Enter your age:</label>
  <input type="number" id="age" />
</body>
```

JAVASCRIPT

c13/js/number-polyfill-eg.js

```
yepnope({
  test: Modernizr.inputtypes.number,
  nope: ['js/numPolyfill.js', 'css/number.css'],
  complete: function() {
    console.log('YepNope + Modernizr are done');
  }
});
```

RESULT



This example tests if the browser supports the `<input>` element using a `type` attribute with a value of `number`. Both Modernizr and YepNope are included in the `<head>` of the page so that the fallback is shown correctly.

The `yepnope()` function takes an object literal as a parameter. It's properties include:

- `test`: the feature you are checking for. In this case it is checking Modernizr to see if the number input is supported.
- `yep`: not used in this example can load files if the feature *is* supported.
- `nope`: what to do if feature *is not* supported (you can load multiple files in an array).
- `complete`: can run a function when the checks are complete, and any necessary files have loaded. Here it adds a message to the console to demonstrate how it works.

Note that Modernizr stores the value of the `<input>` element's `type` attribute, in a child object called `inputtypes`. E.g., to check if the HTML5 date selector is supported, you use: `Modernizr.inputtypes.date` (not `Modernizr.date`).

FORM VALIDATION

The final section of this chapter uses one big script to discuss the topic of form validation. It helps users give you responses in the format you need. (The example also has some form enhancements, too.)

Validation is the process of checking whether a value meets certain rules (for example, that a password has a minimum number of characters). It lets you tell users if there is a problem with the values they entered so that they can correct the form before they resubmit it. This has three key advantages:

- You are more likely to get the information you need in a format you can use.
- It is faster to check values in the browser than it is to send data to the server to be checked.
- It saves resources on the server.

In this section you see how to check the values a user enters into a form. These checks happen when the form is submitted. To do this users could press submit or use the Enter on the keyboard, so the validation process will be triggered by the `submit` event (not the `click` event of a submit button).

We will look at validation using one long example. You can see the form below, and the HTML is shown on the right. It uses HTML5 form controls, but the validation is going to be done using JavaScript to make sure that the experience is consistent across all browsers (even if they do support HTML5).

FORM HTML

This example uses HTML5 markup, but validation is performed using JavaScript (not HTML5 validation).

Due to limited space, the code below only shows the form inputs (not the markup for the columns).

HTML

c13/validation.html

```
<form method="post" action="/register">
  <!-- Column 1 -->
  <div class="name">
    <label for="name" class="required">Name:</label>
    <input type="text" placeholder="Enter your name" name="name" id="name"
           required title="Please enter your name">
  </div>
  <div class="email">
    <label for="email" class="required">Email:</label>
    <input type="email" placeholder="you@example.com" name="email" id="email"
           required>
  </div>
  <div class="password">
    <label for="password" class="required">Password:</label>
    <input type="password" name="password" id="password" required>
  </div>
  <div class="password">
    <label for="conf-password" class="required">Confirm password:</label>
    <input type="password" name="conf-password" id="conf-password" required>
  </div>
  <!-- Column 2 -->
  <div class="birthday">
    <label for="birthday" class="required">Birthday:</label>
    <input type="date" name="birthday" id="birthday" placeholder="yyyy-mm-dd"
           required>
    <div id="consent-container" class="hide">
      <label for="parents-consent"> You need a parent's permission to join.
        Tick here if your child can join:</label>
      <input type="checkbox" name="parents-consent" id="parents-consent">
    </div>
  </div>
  <div class="bio">
    <label for="bio">Short Bio (max 140 characters):</label>
    <textarea name="bio" id="bio" rows="5" cols="30"></textarea>
    <span id="bio-count" class="hide">140</span>
  </div>
  <div class="submit"><input type="submit"></div>
</form>
```

VALIDATION OVERVIEW

This example has over 250 lines of code and will take 22 pages to explain. The script starts by looping through each element on the page performing two generic checks on every form control.

GENERIC CHECKS

First, the code loops through every element in the form and performs two types of **generic** checks. They are generic checks because they would work on any element, and would work with any form.

1. Does the element have the `required` attribute? If so, does it have a value?
2. Does the value match with the `type` attribute? E.g., Does an `email` input hold an email address?

CHECKING EACH ELEMENT

To work through each element in the form, the script makes use the form's `elements` collection (which holds a reference to each form control). The collection is stored in a variable called `elements`. In this example, the `elements` collection will hold the following form controls. The right-hand column tells you which elements are required to have a value:

INDEX	ELEMENT	REQUIRED
0	`elements.name`	Yes
1	`elements.email`	Yes
2	`elements.password`	Yes
3	`elements.conf-password`	Yes
4	`elements.birthday`	Yes
5	`elements.parents-consent`	If under 13
6	`elements.bio`	No

Some developers proactively cache form elements in variables in case validation fails. This is a good idea, but to keep this (already very long) example simpler, the nodes for the form elements are not cached.

If you have not already done so, it would be helpful to download the code for this example from the website, javascriptbook.com, and have it ready when you are reading through the following pages.

Once the generic checks have been performed, the script then makes some checks that apply to individual elements on the form. Some of these checks apply only to this specific form.

CUSTOM VALIDATION TASKS

Next the code performs checks that correspond with specific elements in the form (not *all* elements):

- Do the passwords match?
- Is the bio in the textarea under 140 characters?
- If the user is less than 13 years old, is the parental consent checkbox selected?

These checks are specific to this form and only apply to selected elements in the form (not all of them).

TRACKING VALID ELEMENTS

To keep track of errors, an object called `valid` is created. As the code loops through each element performing the generic checks, a property is added to the `valid` object for each element:

- The property name is the value of its `id` attribute.
- The value is a Boolean. Whenever an error is found on an element, this value is set to `false`.

PROPERTIES OF THE VALID OBJECT

`valid.name`
`valid.email`
`valid.password`
`valid.conf-password`
`valid.birthday`
`valid.parents-consent`
`valid.bio`

DEALING WITH ERRORS

If there are errors, the script needs to prevent the form being submitted and tell the user what they need to do in order to correct their answers.

As the script checks each element, if an error is found, two things happen:

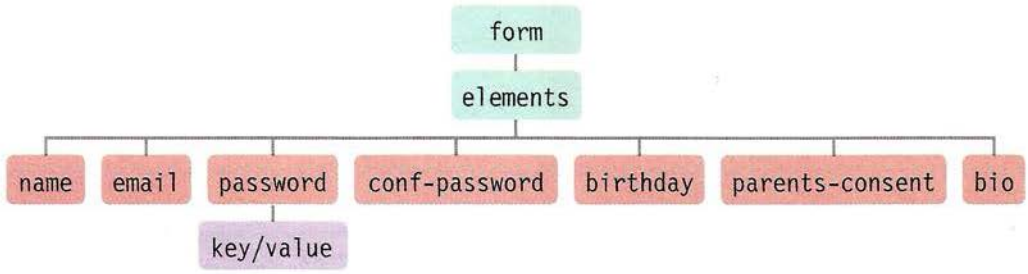
- The corresponding property of the `valid` object is updated to indicate the content is not valid.
- A function called `setErrorMessage()` is called. This function uses jQuery's `.data()` method, which allows you to store data *with* the element. So the error message is stored in memory along with the form element that has the problem.

After each element has been checked, then error messages can be shown using `showErrorMessage()`. It retrieves the error message and puts it in a `<span>` element, which is added after the form control.

Each time the user tries to submit the form, if an error was *not* found on an element it is important to remove any error messages from that element. Consider the following scenario:

a) A user filled out a form with more than one error.
b) This triggered multiple error messages.
c) The user fixes one problem, so its corresponding message must be removed, while error message(s) for problems that have not been fixed must remain visible.

Therefore, when each of the elements is looped through, either an error message is set, or the error message is removed.



Above you can see a representation of the form and its `elements` collection. There was a problem with the email input, so the `.data()` method has stored a key/value pair with that element.

This is how the `setErrorMessage()` function will store the error messages to show to the user. If the error is fixed, then the error value is cleared (and the element with the error message removed).

SUBMITTING THE FORM

Before sending the form, the script checks whether there were any errors. If there were, the script stops the file from being submitted.

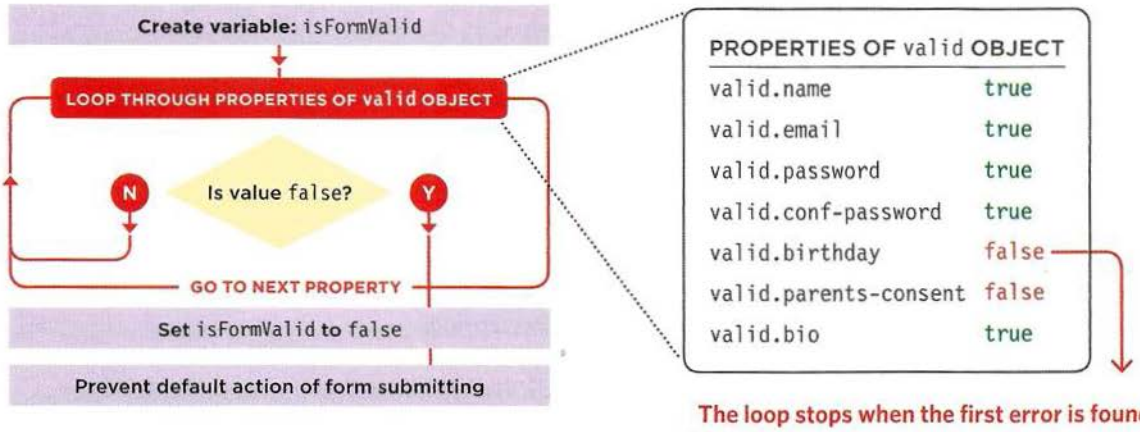
In order to check whether any errors were found, a variable called `isFormValid` is created and is given a value of `true`. The script then loops through each property of the `valid` object, and if there was an error (if *any* property of that object has a value of `false`), then there is an error in the form and the `isFormValid` variable is also set to `false`.

So, `isFormValid` is being used as a **flag** (you can think of it being like a master switch) if an error is found, it is turned off. At the end of the script, if `isFormValid` is `false` then an error must have been found and the form should not be submitted (using the `preventDefault()` method).

It is important to check and process all of the elements before deciding whether to submit the form so that you can show all of the relevant error messages in one go.

If every value has been checked, the user can be shown all of the things they have to amend before re-submitting the form.

If the form only showed the first error it came across, and stopped, the user would only see one error each time they submitted the form. This could soon become frustrating for the user if they were to keep trying to submit the form and see new errors.



The loop stops when the first error is found.
(Note that error messages are already visible.)

CODE OVERVIEW

On the right is an outline of the validation code, split into four sections. On line 3, an anonymous function is called when the form is submitted. It orchestrates the validation, in turn calling other functions (not all of which are shown on the right-hand page, see further pages for more).

A: SET UP THE SCRIPT

1. The code lives inside an IIFE (creating function-level scope).
2. This script uses JavaScript validation to ensure that error messages look the same on all browsers, so HTML5 validation is turned off by setting the `noValidate` property of the form to `true`.
3. When the user submits the form, an anonymous function is run (this contains the validation code).
4. `elements` holds a collection of all form elements.
5. `valid` is the object that keeps track of whether or not each form control is valid. Each form control is added as a property of the `valid` object.
6. `isValid` is a flag that is re-used to check whether individual elements are valid.
7. `isFormValid` is a flag that is used as a master switch to check whether the *entire* form is valid.

B: PERFORM GENERIC CHECKS

8. The code loops through each form control.
9. It performs two generic checks on each one:
i) Is the element required? If so, does it have a value? Uses `validateRequired()`. See p606.
ii) Does the value correspond with the type of data it should hold? Uses `validateTypes()`. See p610.
If either of these functions does not return `true`, then `isValid` is set to `false`.
10. An `if...else` statement checks if that element passed the tests (by checking if `isValid` is `false`).
11. If the control is not valid, `showErrorMessage()` shows an error message to the user. See p609.
12. If it is valid, `removeErrorMessage()` removes any errors associated with that element.
13. The value of the element's `id` attribute is added as a property `valid` object; its value is whether or not the element was valid.

C: PERFORM CUSTOM VALIDATION

14. After the code has looped through every element on the form, the custom validation can occur. There are three types of custom validation occurring (each one uses its own function):
i) Is the bio too long? See p615.
ii) Do passwords match?
iii) Is user old enough to join on own? If not, is the parental approval checkbox selected? See p617.
15. If an element fails one of the custom validation checks, `showErrorMessage()` will be called, and the corresponding property in the `valid` object will be set to `false`.
16. If the element passes the check, `removeErrorMessage()` is called for that element.

D: DID THE FORM PASS VALIDATION?

The `valid` object now has a property for each element, and the value of that property states whether or not the element was valid or not.
17. The code loops through each property in the `valid` object.
18. An `if` statement checks to see if the element was *not* valid.
19. If it was not valid, set `isFormValid` to false and stop the loop.
20. Otherwise, `isFormValid` is set to `true`.
21. Finally, having looped through the `valid` object, if `isFormValid` is not true, the `preventDefault()` method prevents the form being submitted. Otherwise, it is sent.

```
  // SET UP THE SCRIPT
① (function () {
②   document.forms.register.noValidate = true; // Disable HTML5 validation
③   $('form').on('submit', function(e) {       // When form is submitted
④     var elements = this.elements;            // Collection of form controls
⑤     var valid = {};                          // Custom valid object
⑥     var isValid;                             // isValid: checks form controls
⑦     var isFormValid;                         // isFormValid: checks entire form

      // PERFORM GENERIC CHECKS (calls functions outside the event handler)
⑧     for (var i = 0, l = (elements.length - 1); i < l; i++) {
        // Next line calls validateRequired() see p606 & validateTypes() p610
⑨       isValid = validateRequired(elements[i]) && validateTypes(elements[i]);
⑩       if (!isValid) {                        // If it does not pass these two tests
⑪         showErrorMessage(elements[i]);       // Show error messages (see p608)
        } else {                               // Otherwise
⑫         removeErrorMessage(elements[i]);     // Remove error messages
        }                                      // End if statement
⑬       valid[elements[i].id] = isValid;       // Add element to the valid object
      }                                        // End for loop

      // PERFORM CUSTOM VALIDATION (just 1 of 3 functions - see p614-p617)
⑭     if (!validateBio()) {                    // Call validateBio(), if not valid
⑮       showErrorMessage(document.getElementById('bio'));   // Show error
        valid.bio = false;                     // Update valid object-not valid
      } else {                                 // Otherwise
⑯       removeErrorMessage(document.getElementById('bio')); // Remove error
      } // two more functions follow here (see p614-p617)

      // DID IT PASS / CAN IT SUBMIT THE FORM?
      // Loop through valid object, if there are errors set isFormValid to false
⑰     for (var field in valid) {               // Check properties of the valid object
⑱       if (!valid[field]) {                   // If it is not valid
          isFormValid = false;                 // Set isFormValid variable to false
⑲         break;                               // Stop the for loop, error was found
        }                                      // Otherwise
⑳       isFormValid = true;                    // The form is valid and OK to submit
      }
      // If the form did not validate, prevent it being submitted
      if (!isFormValid) {                      // If isFormValid is not true
㉑      e.preventDefault();                    // Prevent the form being submitted
      }
    });                                        // End event handler
    ...                                        // Functions called above are here
  }());                                        // End of IIFE
```

REQUIRED FORM ELEMENTS

The HTML5 `required` attribute indicates a field must have a value. Our `validateRequired()` function will first check for the attribute. If present, it then checks whether or not it has a value.

`validateRequired()` is called for each element individually (see step 9, p605). Its one parameter is the element it is checking.

In turn, it calls upon three other named functions.
i) `isRequired()` checks for the `required` attribute.

ii) `isEmpty()` can check if the element has a value.
iii) `setErrorMessage()` sets error messages if there are problems.

```
function validateRequired(el) {
  if (isRequired(el)) {                          // Is this element required
    var valid = !isEmpty(el);                    // Is value not empty (true/false)
    if (!valid) {                                // If valid variable holds false
      setErrorMessage(el, 'Field is required');  // Set the error message
    }
    return valid;                                // Return valid variable (true/false)
  }
  return true;                                   // If not required, all is okay
}
```

A: DOES IT HAVE A REQUIRED ATTRIBUTE?

1. An `if` statement uses a function called `isRequired()` to check whether the element carries the `required` attribute. You can see the `isRequired()` function on the right-hand page. If the attribute is present, the subsequent code block is run.

6. If not, the code skips to step to step 6 to say this element is OK.

B: IF SO, DOES IT HAVE A VALUE?

If the field is required, the next step is to check whether or not it has a value. This is done using a function called `isEmpty()`, also shown on the right-hand page.

2. The result from `isEmpty()` is stored in a variable called `valid`. If it is *not* empty, the `valid` variable will hold a value of `true`. If it is empty, it holds `false`.

C: SHOULD AN ERROR MESSAGE BE SET?

3. An `if` statement checks if the `valid` variable is *not* `true`.

4. If it is not true, an error message is set using the `setErrorMessage()` function, which you meet on p608.

5. The `valid` variable is returned on the next line, and that is where this function ends.

validateRequired() uses two functions to perform checks:
1: isRequired() checks whether the element has a required attribute.
2: isEmpty() checks whether the element has a value.

isRequired()

The isRequired() function takes an element as a parameter and checks if the required attribute is present on that element. It returns a Boolean.

There are two types of check: The first, in blue, is for browsers that support the HTML5 required attribute. The one in orange is for older browsers.

To check if the required attribute is present, the typeof operator is used. It checks what datatype the browser thinks the required attribute is.

```
function isRequired(el) {
  return ((typeof el.required === 'boolean') && el.required) ||
    (typeof el.required === 'string');
}
```

MODERN BROWSERS

Modern browsers know the required property is a Boolean, so the first part of this check tells us if it is a modern browser. The second part checks if it is present on this element.
If the attribute is present, it will evaluate to true. If not, it returns undefined, which is considered a falsy value.

OLDER BROWSERS

Browsers that do not know HTML5 can still tell whether or not an HTML5 attribute is present on an element. In those browsers, if the required attribute is present, it gets treated as a string, so the condition would evaluate to true. If not, the type would be undefined, which is falsy.

WHAT IS VALIDATED

It is important to note that the required attribute only indicates that *a* value is required. It doesn't stipulate how long the value should be, nor does it perform any other kind of validation. Specific checks, such as these, would have to be added in the validateTypes() function or the script's custom validation section.

isEmpty()

The isEmpty() function (below) takes an element as a parameter and checks to see if it has a value. As with isRequired(), two checks are used to handle both new and older browsers.

ALL BROWSERS

The first check looks to see if the element does *not* have a value. If it has a value, the function should return false. If it is empty, it will return true.

OLDER BROWSERS

If older browsers use a polyfill for placeholder text, the value would be the same as the placeholder, so it is considered empty if those values match.

```
function isEmpty(el) {
  return !el.value || el.value === el.placeholder;
}
```

CREATING ERROR MESSAGES

The validation code processes elements one by one; any error messages are stored using jQuery's `.data()` method.

HOW ERRORS ARE SET

Throughout the validation code, whenever an error is found, you will see calls to a function called `setErrorMessage()`, which takes two parameters:
i) `el`: the element that the error message is for
ii) `message`: the text the error message will display

For example, the following would add the message `'Field is required'` to the element that is stored in the `el` variable:

```
setErrorMessage(el, 'Field is required');
```

HOW DATA IS STORED WITH NODES

Each error message is going to be stored with the element node that it relates to using the jQuery `.data()` method. When you have elements in a jQuery matched set, the `.data()` method allows you to store information in key/value pairs for each individual element.

The `.data()` method has two parameters:
i) The key, which is always going to be `errorMessage`
ii) The value, which is the text that the error message will display

setErrorMessage()

```
① function setErrorMessage(el, message) {
②   $(el).data('errorMessage', message);              // Store error message with element
  }
```

DISPLAYING ERROR MESSAGES

After each element has been checked, if one or more were not valid, `showErrorMessage()` will display the error messages on the page.

HOW ERRORS ARE DISPLAYED

If an error message needs to be shown, first a `<span>` element will be added to the page directly after the form field with the error.

Next, the message is added into the `<span>` element. To get the text for the error message, the same jQuery `.data()` method that set the message is used again. This time, it only takes one parameter: the key (which is always `errorMessage`).

This all happens within the function called `showErrorMessage()` which is shown below.

1. `$el` holds a jQuery selection containing the element that the error message relates to.
2. `$errorContainer` looks for any existing errors on this element by checking if it has any sibling elements that have a class of `error`.
3. If the element does not have an error message associated with it, the code in the curly braces runs.
4. `$errorContainer` is set to hold a `<span>` element. Then `.insertAfter()` adds the `<span>` element into the page after the element causing the error.
5. The content of the `<span>` element is populated with the error message for that element, which is retrieved using the `.data()` method of the element.

showErrorMessage()

```
function showErrorMessage(el) {
  var $el = $(el);                                          // Find element with the error
  var $errorContainer = $el.siblings('.error');             // Does it have errors already

  if (!$errorContainer.length) {                            // If no errors found
    // Create a <span> to hold the error and add it after the element with the error
    $errorContainer = $('<span class="error"></span>').insertAfter($el);
  }
    $errorContainer.text($(el).data('errorMessage'));       // Add error message
}
```

VALIDATING DIFFERENT TYPES OF INPUT

HTML5's new types of input come with built-in validation. This example uses HTML5 inputs, but validates them with JavaScript to ensure that the experience is consistent across all browsers.

The `validateTypes()` function is going to perform the validation just like modern browsers do with HTML5 elements, but it will do it for all browsers. It needs to:

- Check what type of data the form element should hold
- Ensure the contents of the element matches that type

1. The first line in the function checks if the element has a value. If the user has not entered any information, you cannot validate the type of data. Furthermore, it is not the *wrong* type of data. So, if there is *no* value, the function returns `true` (and the rest of the function does not need to run).

2. If there is a value, a variable called `type` is created to hold the value of the `type` attribute. First, the code checks to see if jQuery stored info about the type using its `.data()` method (see why on p618). If not, it gets the value of the `type` attribute.

```
function validateTypes(el) {
① if (!el.value) return true;                          // If element has no value, return true
                                                       // Otherwise get the value from .data()
② var type = $(el).data('type') || el.getAttribute('type'); // or get the type of input
③ if (typeof validateType[type] === 'function') {      // Is type a method of validate object?
④   return validateType[type](el);                     // If yes, check if the value validates
  } else {                                             // If not
⑤   return true;                                       // Return true as it cannot be tested
  }
}
```

The `getAttribute()` method is used rather than the DOM property for `type` because all browsers can return the value of the `type` attribute, whereas browsers that don't recognize a new HTML5 DOM property types would just return `text`.

3. This function uses an object called `validateType` (shown on the next page) to check the content of the element. The `if` statement checks if the `validateType` object has a method whose name matches the value of the `type` attribute.

If it has a method name that matches the type of form control:

4. The element is passed to the object; it returns `true` or `false`.

5. If there is no matching method, the object is not able to validate the form control and no error message should be set.

CREATING AN OBJECT TO VALIDATE DATA TYPES

The `validateType` object (outlined below) has three methods:

```
var validateType = {
  email: function(el) {
  // Check email address
  },
  number: function(el) {
  // Check it is a number
  },
  date: function(el) {
  // Check date format
  }
}
```

The code inside each method is virtually identical. You can see the format of the `email()` method below. Each method validates the data using something called a **regular expression**. The regular expression is the only thing that changes in each method to test the different data types.

Regular expressions allow you to **check for patterns** in strings, and here they are used with a method called `test()`.

You can learn more about regular expressions and their syntax on the next two pages. For now, you just need to know that they are used to check the data contains a specific pattern of characters.

Storing these checks as methods of an object makes it easy to access each of the the different checks when it comes time to validate the different types of input in a form.

```
/[^@]+@[^@]+/.test(el.value);
```
(i) (ii) (iii)

i) The regular expression is `[^@]+@[^@]+` (it is between the / and / characters). It states a pattern of characters that are found in a typical email address.

ii) The `test()` method takes one parameter (a string), and checks whether the regular expression can be found within the string. It returns a Boolean.

iii) In this example, the `test()` method is passed the value of the element you want to check. Below you can see the method to test email addresses.

```
email: function (el) {                                  // Create email method
  var valid = /[^@]+@[^@]+/.test(el.value);             // Store result of test in valid
  if (!valid) {                                         // If the value of valid is not true
    setErrorMessage(el, 'Please enter a valid email'); // Set error message
  }
  return valid;                                         // Return the valid variable
},
```
(1) (2) (3) (4)

1. A variable called `valid` holds the result of the test using the regular expression.

2. If the string does not contain a match for the regular expression, **3.** an error message is set.

4. The function returns the value of the `valid` variable (which is `true` or `false`).

REGULAR EXPRESSIONS

Regular expressions search for characters that form a pattern. They can also replace those characters with new ones.

Regular expressions do not just search for matching letters; they can check for sequences of upper/lowercase characters, numbers, punctuation, and other symbols.

The idea is similar to the functionality of find and replace features in text editors, but it makes it possible to create far more complicated searches for combinations of characters.

Below you can see the building blocks of regular expressions. On the right-hand page, you can see some examples of how they are combined to create powerful pattern-matching tools.

`.`
any single character (except newline)

`[ ]`
single character contained within brackets

`[^ ]`
single character not contained within brackets

`^`
the starting position in any line

`$`
the ending position in any line

`( )`
sub expressions (sometimes called a block or capturing group)

`*`
preceding element zero or more times

`\n`
*n*th marked subexpression (*n* is digit 1-9)

`{m,n}`
preceding element at least *m*, but no more than *n*, times

`\d`
digit

`\D`
non-digit character

`\s`
whitespace character

`\S`
anything but whitespace

`\w`
alphanumeric character (A-Z, a-z, 0-9)

`\W`
non-alphanumeric character (except _)

COMMON REGULAR EXPRESSIONS

Here are a selection of regular expressions you can use in your code. Some of these are more powerful than those adopted by browsers.

At the time of writing, some of the validation rules applied by the major browsers were not very strong. Some of the regular expressions shown below are more stringent.

But regular expressions are not perfect. There are still strings that would not be valid data, but would pass these tests below.

Also, bear in mind that there are many different ways to express the same thing using regular expressions. So you may see a very different regular expression that does something similar.

```
/^\d+$/
```

number

```
^[ \s]+
```

whitespace at start of line

```
/[^@]+@[^@]+/
```

email

```
/^#[a-fA-F0-9]{6}$/
```

hex color value

```
!"#$%&\'()*+,-./@:;<=>[\\]^_`{|}~
```

hex color value

```
/^(\d{2}\/\d{2}\/\d{4})|(\d{4}-\d{2}-\d{2})$/
```

date yy-mm-dd

CUSTOM VALIDATION

The final part of the script performs three checks that apply to individual form elements; each check lives in a named function.

On the next pages, you will see these three functions. Each is called in the same manner as the `validateBio()` function shown below. (The full code that calls them is available from the website, along with the code for all examples from the book.)

FUNCTION	PURPOSE
`validateBio()`	Check bio is 140 characters or less
`validatePassword()`	Check password is at least 8 characters
`validateParentsConsent()`	If user is under 13, test if parental consent box is checked

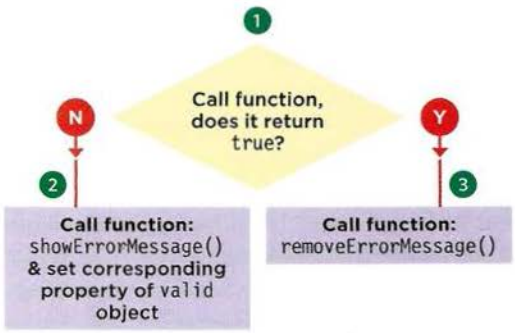
Each of these functions will return a value of `true` or `false`.

```
1 if (!validateBio()) {                                  // Call validateBio(), if not valid
2   showErrorMessage(document.getElementById('bio'));    // Show error message
    valid.bio = false;                                   // Update valid object - not valid
  } else {                                               // Otherwise remove error message
3   removeErrorMessage(document.getElementById('bio'));
  }
```

1. The function is called as a condition in an `if... else` statement. This was shown in steps 14-16 on p605.

2. If the function returns `false`, an error message is shown and the corresponding property of the `valid` object is set to `false`.

3. If the function returns `true`, the error message is removed from the corresponding element.



BIO & PASSWORD VALIDATION

The `validateBio()` function:
1. Stores the form element containing the user's biography in a variable called `bio`.
2. If the length of the bio is less than or equal to 140 characters, the `valid` variable is set to `true` (otherwise, it is set to `false`).
3. If `valid` is not `true`, then...
4. The `setErrorMessage()` function is called (see p608).
5. The `valid` attribute is returned to the calling code, which will show or hide the error.

JAVASCRIPT — c13/js/validation.js

```
function validateBio() {
①  var bio   = document.getElementById('bio');  // Store ref to bio text area
②  var valid = bio.value.length <= 140;         // Is bio <= 140 characters?
③  if (!valid) {                                // If not, set an error message
④    setErrorMessage(bio, 'Your bio should not exceed 140 characters');
   }
⑤  return valid;                                // Return Boolean value
}
```

The `validatePassword()` function starts by:
1. Storing the element containing the password in a variable called `password`.
2. If the length of the value in the password input is greater than or equal to 8, `valid` is set to `true` (otherwise, it is set to `false`).
3. If `valid` is not `true`, then...
4. The `setErrorMessage()` function is called.
5. The `valid` attribute is returned to the calling code, which will show or hide the error.

JAVASCRIPT — c13/js/validation.js

```
function validatePassword() {
①  var password = document.getElementById('password');// Store ref to element
②  var valid    = password.value.length >= 8;        // Is its value >= 8 chars
③  if (!valid) {                                     // If not, set error msg
④    setErrorMessage(password, 'Password must be at least 8 characters');
   }
⑤  return valid;                                     // Return true / false
}
```

CODE DEPENDENCIES & REUSE

In any project, avoid writing two sets of code that perform the same task. You can also try to reuse code across projects (for example, using utility scripts or jQuery plugins). If you do, note any dependencies in your code.

DEPENDENCIES

Sometimes one script will require another script to be included in the page in order to work. When you write a script that relies on another script, the *other* script is known as a dependency.

For example, if you are writing a script that uses jQuery, then your script depends upon jQuery being included in the page in order to work; otherwise, you would not be able to use its selectors or methods.

It is a good idea to note dependencies in a comment at the top of the script so that they are clear to others. The final custom function in this example depends on another script that checks the user's age.

CODE REUSE VS. DUPLICATION

When you have two sets of code that do the same job, it is referred to as code duplication. This is usually considered bad practice.

The opposite is code reuse where the same lines of code are used in more than one part of a script (functions are a good example of code reuse).

You may hear programmers refer to this as the **DRY principle**: don't repeat yourself. "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." It was formulated by Andrew Hunt and Dave Thomas in a book called *The Pragmatic Programmer* (Addison-Wesley, 1999).

To encourage reuse, programmers sometimes create a set of smaller scripts (instead of one big script). Therefore, code reuse can lead to more code dependencies. You have already seen an example of this with the helper functions for event handling. You are about to see another example...

VALIDATING PARENTAL CONSENT

When the validation script was introduced, it was noted that the form would use a couple of scripts to enhance the page. You start to see those scripts on the next page, but one of them needs to be noted now because it hides the parental consent checkbox when the page loads.

That parental consent checkbox is only shown again if the user indicates that they are 13 years old or younger.

The validation code to check whether the parent has given their consent will only run if that checkbox is showing.

So the code to check whether the parent has given consent depends upon (reuses) the same code that checked if the checkbox should be shown. This works well as long as the other script (to show/hide the checkbox) is included in the page before the validation script.

The `validateParentsConsent()` function is called in the same way as the other two custom validation checks (see p614). Inside the function:

1. It stores the checkbox for parental consent and its containing element in variables.
2. Sets a `valid` variable to `true`.
3. An `if` statement checks whether the container for the checkbox is *not* hidden. It does this by fetching the value of its `class` attribute and using the `indexOf()` function (which you saw on p128) to check if it contains a value of `hide`. If the value is *not* found, then `indexOf()` will return `-1`.
4. If it is not hidden, the user is under 13. So, if the checkbox is selected, the `valid` variable is set to the `true`, and if it was not selected, it will be set to `false`.
5. If it is not valid, an error message is added to the element.
6. The function returns the value of the `valid` variable to indicate whether the consent was given.

JAVASCRIPT

c13/js/validation.js

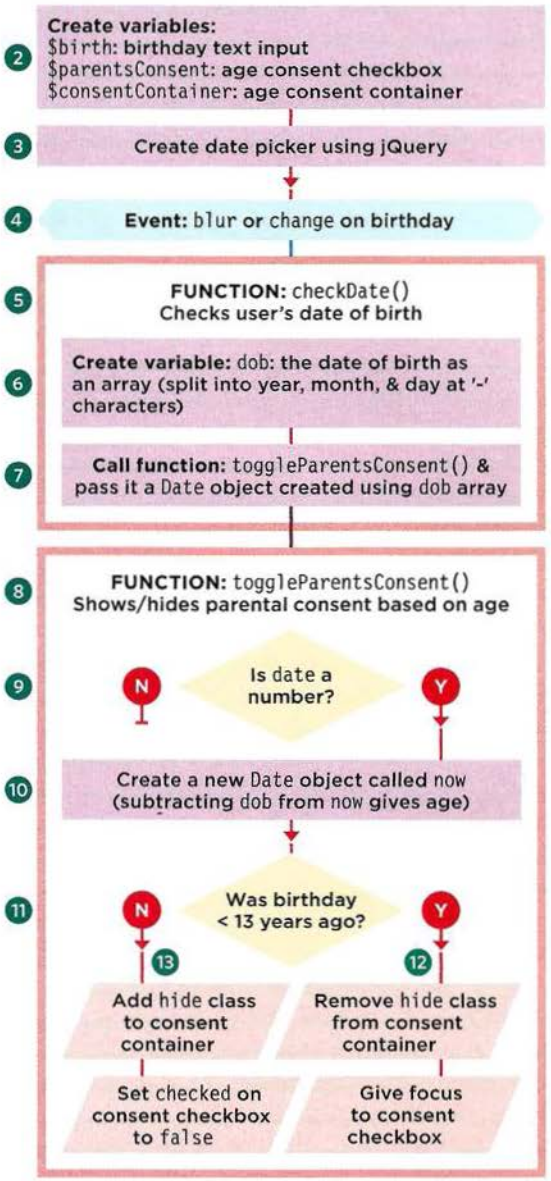
```
function validateParentsConsent() {
  var parentsConsent   = document.getElementById('parents-consent');
  var consentContainer = document.getElementById('consent-container');
  var valid = true;                                  // Variable: valid set to true
  if (consentContainer.className.indexOf('hide') === -1) { // If checkbox shown
    valid = parentsConsent.checked;                  // Update valid: is it checked/not
    if (!valid) {                                    // If not, set the error message
      setErrorMessage(parentsConsent, 'You need your parents\' consent');
    }
  }
  return valid;                                      // Return whether valid or not
}
```

HIDE PARENTAL CONSENT

As you saw on the previous page, the subscription form uses two extra scripts to enhance the user experience. Here is the first; it does two things:

- Uses the jQuery UI date picker to show a consistent date picker across browsers
- Checks whether the parental consent checkbox should be shown when the user leaves the date input (it does this if they are under 13)

1. Place the script in an IIFE (not shown in flowchart).
2. Three jQuery selections store the input where users enter their birthday, the consent checkbox, and the container for the consent checkbox.
3. The jQuery selection for the date of birth input is converted from a date input to a text input so that it does not conflict with HTML5 date picker functionality (done using the jQuery `.prop()` method to alter the value of its `type` attribute). The selection uses `.data()` to note that it is a date input and jQuery UI's `.datepicker()` method to create the jQuery UI date picker.
4. When the user leaves the date input, the `checkDate()` function is called.
5. The `checkDate()` function is declared.
6. A variable called `dob` is created to hold the date the user selected. The date is converted into an array of three values (month, day, and year) using the `split()` method of the `String` object.
7. `toggleParentsConsent()` is called. It has one parameter: the date of birth. It is passed into the function as a `Date` object.
8. `toggleParentsConsent()` is declared.
9. Inside the function, it checks the date is a number. If not, `return` indicates the function should stop.
10. The current time is obtained by creating a new `Date` object (the current time is the default value of a new `Date` object). It is stored in a variable called `now`.
11. To find the user's age, the date of birth is subtracted from the current date. For simplicity, leap years are ignored. If that is less than 13 years:
12. Show the container for the parental consent.
13. Otherwise, the container of the consent box is hidden, and the checkbox is unchecked.



AGE CONFIRMATION

JAVASCRIPT

c13/js/birthday.js

```
(function() {
  var $birth           = $('#birthday');              // D-O-B input
  var $parentsConsent  = $('#parents-consent');       // Consent checkbox
  var $consentContainer = $('#consent-container');    // Checkbox container
  // Create the date picker using jQuery UI
  $birth.prop('type', 'text').data('type', 'date').datepicker({
    dateFormat: 'yy-mm-dd'                            // Set date format
  });
  $birth.on('blur change', checkDate);                // D-O-B loses focus
  function checkDate() {                              // Declare checkDate()
    var dob = this.value.split('-');                  // Array from date
    // Pass toggleParentsConsent() the date of birth as a date object
    toggleParentsConsent(new Date(dob[0], dob[1] - 1, dob[2]));
  }
  function toggleParentsConsent(date) {               // Declare function
    if (isNaN(date)) return;                          // Stop if date invalid
    var now = new Date();                             // New date obj: today
    // If difference (now minus date of birth, is less than 13 years
    // show parents consent checkbox (does not account for leap years)
    // To get 13 yrs ms * secs * mins * hrs * days * years
    if ((now - date) < (1000 * 60 * 60 * 24 * 365 * 13)) {
      $consentContainer.removeClass('hide');          // Remove hide class
      $parentsConsent.focus();                        // Give it focus
    } else {                                          // Otherwise
      $consentContainer.addClass('hide');             // Add hide to class
      $parentsConsent.prop('checked', false);         // Set checked to false
    }
  }
}());
```

When creating a date picker using jQuery UI, you can specify the format in which you want the date to be written. On the right you can see several options for the format of the date and what this would look like if the date were the 20th December 1995. In particular note that y gives you two digits for the year, and yy gives you four digits for the year.

FORMAT	RESULT
mm/dd/yy	12/20/1995
yy-mm-dd	1995-12-20
d m, y	20 Dec, 95
mm d, yy	December 20, 1995
DD, d mm, yy	Saturday, 20 December, 1995

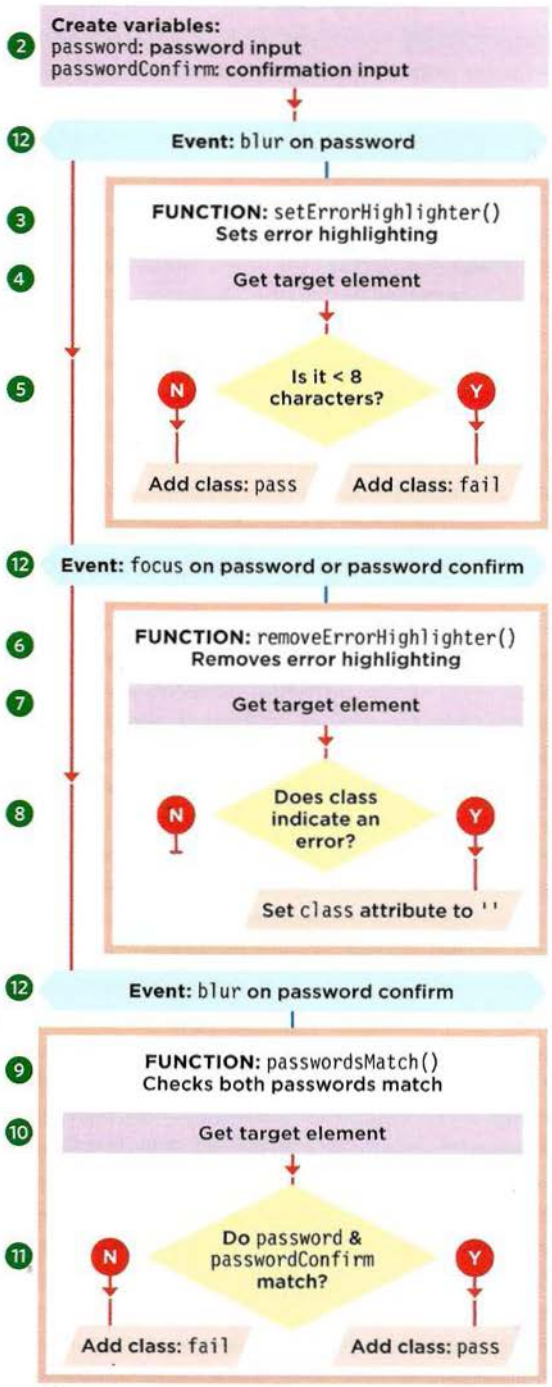
PASSWORD FEEDBACK

The second script designed to enhance the form provides feedback to the users as they leave either of the password inputs. It changes the value of the `class` attribute for the password inputs, offering feedback to show whether or not the password is long enough and whether or not the value of the password and its confirmation box match.

1. Place the script in an IIFE (not shown in flowchart).
2. Variables store references to the password input and the password confirmation input.
3. `setErrorHighlighter()` function is declared.
4. It retrieves the target of the event that called it.
5. An `if` statement checks the value of that element. If it is less than 8 characters, that element's `class` attribute is given a value of `fail`. Otherwise, it is given a value of `pass`.
6. `removeErrorHighlighter()` is declared.
7. It retrieves the target of the event that called it.
8. If the value of the `class` attribute is `fail`, then the value of the `class` attribute is set to a blank string (clearing the error).
9. `passwordsMatch()` is declared (it is only called by the password confirm box).
10. It retrieves the target of the event that called it.
11. If the value of that element is the same as the first password input, its `class` attribute is given a value of `pass`; otherwise, it is given a value of `fail`.
12. Event listeners are set up:

ELEMENT	EVENT	METHOD
`password`	`focus`	`removeErrorHighlighter()`
`password`	`blur`	`setErrorHighlighter()`
`conf-password`	`focus`	`removeErrorHighlighter()`
`conf-password`	`blur`	`passwordsMatch()`

This demonstrates how scripts often group all of the functions and the event handlers together.



PASSWORD SCRIPT

JAVASCRIPT

c13/js/password-signup.js

```
(function () {
  var password = document.getElementById('password');  // Store password inputs
  var passwordConfirm = document.getElementById('conf-password');
  function setErrorHighlighter(e) {
    var target = e.target || e.srcElement;                    // Get target element
    if (target.value.length < 8) {                            // If its length is < 8
      target.className = 'fail';                              // Set class to fail
    } else {                                                  // Otherwise
      target.className = 'pass';                              // Set class to pass
    }
  }
  function removeErrorHighlighter(e) {
    var target = e.target || e.srcElement;                    // Get target element
    if (target.className === 'fail') {                        // If class is fail
      target.className = '';                                  // Clear class
    }
  }
  function passwordsMatch(e) {
    var target = e.target || e.srcElement;                    // Get target element
    // If value matches pwd and it is longer than 8 characters
    if ((password.value === target.value) && target.value.length >= 8){
      target.className = 'pass';                              // Set class to pass
    } else {                                                  // Otherwise
      target.className = 'fail';                              // Set class to fail
    }
  }
  addEvent(password, 'focus', removeErrorHighlighter);
  addEvent(password, 'blur', setErrorHighlighter);
  addEvent(passwordConfirm, 'focus', removeErrorHighlighter);
  addEvent(passwordConfirm, 'blur', passwordsMatch);
}());
```

SUMMARY

FORM ENHANCEMENT & VALIDATION

- Form enhancements make your form easier to use.
- Validation lets you give users feedback before the form data is sent to the server.
- HTML5 introduced new form controls which feature validation (but they only work in modern or mobile browsers).
- HTML5 inputs and their validation messages look different in various browsers.
- You can use JavaScript to offer the same functionality as the new HTML5 elements in all browsers (and control how they appear in all browsers).
- Libraries like jQuery UI help create forms that look the same across different browsers.
- Regular expressions help you find patterns of characters in a string.

INDEX

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

XYZ


flybikes™
bmx toys from spain

LEARN HOW TO

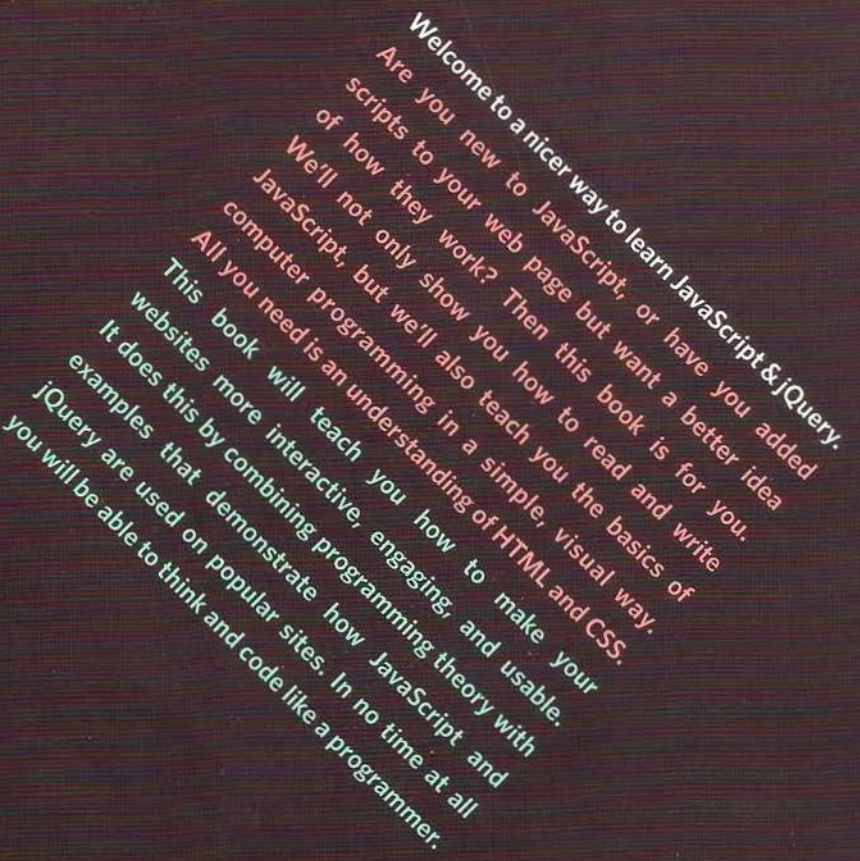
Read and write JavaScript
Make your sites more interactive
Use jQuery to simplify your code
Recreate popular web techniques

TECHNIQUES INCLUDE

Slideshows and lightboxes
Improved forms and validation
Using Ajax, APIs, and JSON
Filtering, searching, and sorting

ONLINE SUPPORT

Code samples and practical exercises available online at:
www.javascriptbook.com
Plus bonus reference materials



FROM THE AUTHOR OF THE BEST-SELLING
HTML & CSS: Design and Build Websites
www.htmlandcssbook.com

WILEY

Web Programming / JavaScript
USA $39.99 / CAN $47.99

ISBN 978-1-118-53164-8