# 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

| Year | Month | Event |
| --- | --- | --- |
| 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`.

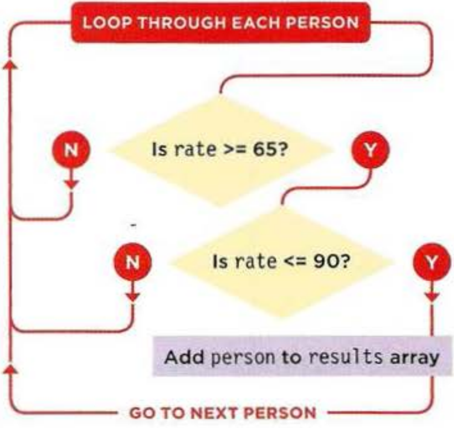
# FILTERING

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
③   if (person.rate >= 65 && person.rate <= 90) {    // Is rate in range
      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
① function priceRange(person) {                              // Declare priceRange()
    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/`

**CreativeFolk** find talented people for your creative projects

**Min:** 65 **Max:** 90

| NAME | HOURLY RATE ($) |
|---|---|
| Camille | 80 |
| Gordon | 75 |

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.

### ROWS ARRAY

| INDEX: | OBJECT: | |
|---|---|---|
| 0 | person | people[0] |
| | $element | <tr> |
| 1 | person | people[1] |
| | $element | <tr> |
| 2 | person | people[2] |
| | $element | <tr> |
| 3 | person | people[3] |
| | $element | <tr> |

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
});
```

## PEOPLE ARRAY

| INDEX: | OBJECT: | |
|---|---|---|
| 0 | name | Casey |
| | rate | 70 |
| 1 | name | Camille |
| | rate | 80 |
| 2 | name | Gordon |
| | rate | 75 |
| 3 | name | Nigel |
| | rate | 120 |

## HTML TABLE

table
tbody
tr
td
td
tr
td
td
tr
td
td
tr
td
td

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.

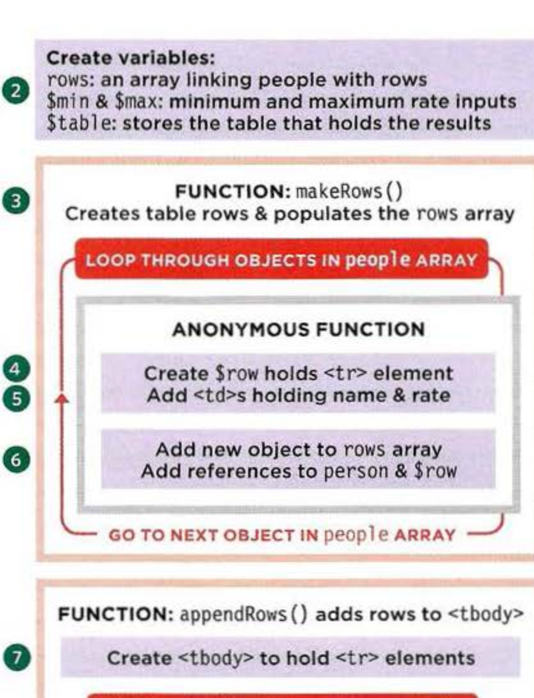
**3.** `makeRows()` loops through each person in the `people` array calling an anonymous function for each object in the array. Note how `person` is used as a parameter name. This means that within the function, `person` refers to the current object in the array.
**4.** For each person, a new jQuery 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.

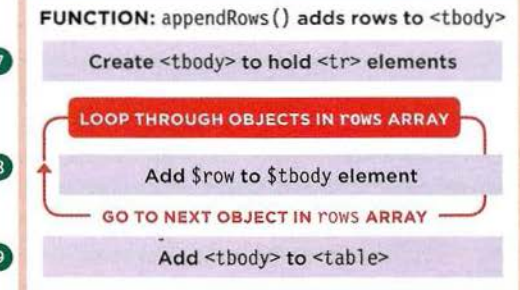
**7.** `appendRows()` creates a new jQuery object called `$tbody` containing a `<tbody>` element.
**8.** It then loops through all of the objects in the `rows` array and adds their `<tr>` element to `$tbody`.
**9.** The new `$tbody` selection is added to the `<table>`.

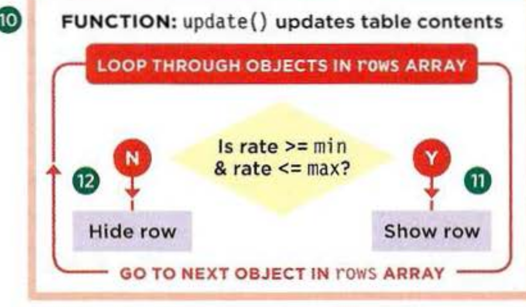
**10.** `update()` goes through each of the objects in the `rows` array and checks if the rate that the person charges is more than the minimum and less than the maximum rate shown on the slider.
**11.** If it is, jQuery's `show()` method shows the row.
**12.** If not, jQuery's `hide()` method hides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




13 **FUNCTION:** `init()` **sets up the script**
14 Set up `slider`
15 Call `makeRows()`, `appendRows()`, `update()`

16 Call `init()` when the DOM has loaded

# 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
       rows.forEach(function(row) {               // For each object in the rows array
⑧       $tbody.append(row.$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

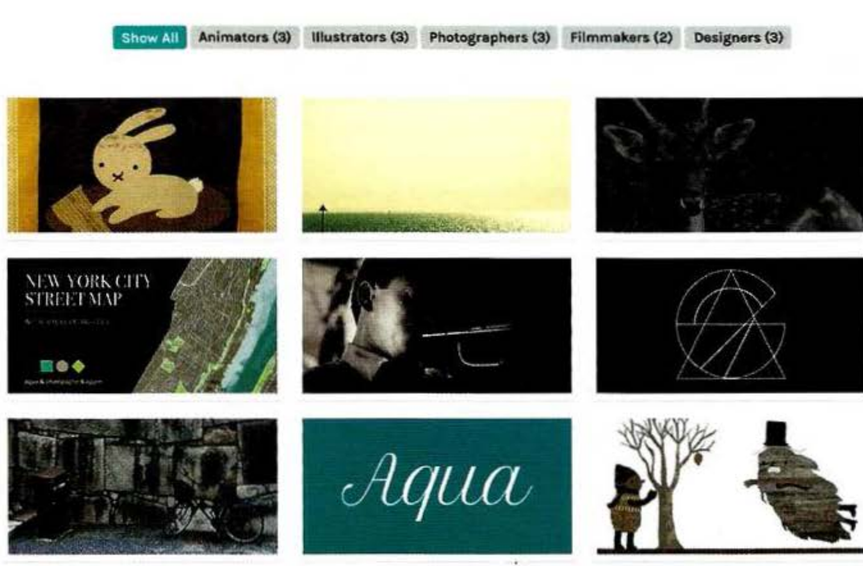
In this example, a series of photos are tagged. The tags are stored in an HTML attribute called `data-tags` on each of the `<img>` elements. HTML5 allows you to store any data with an element using an attribute that starts with the word `data-`. The tags are comma-separated.
(See right-hand page)

### 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.

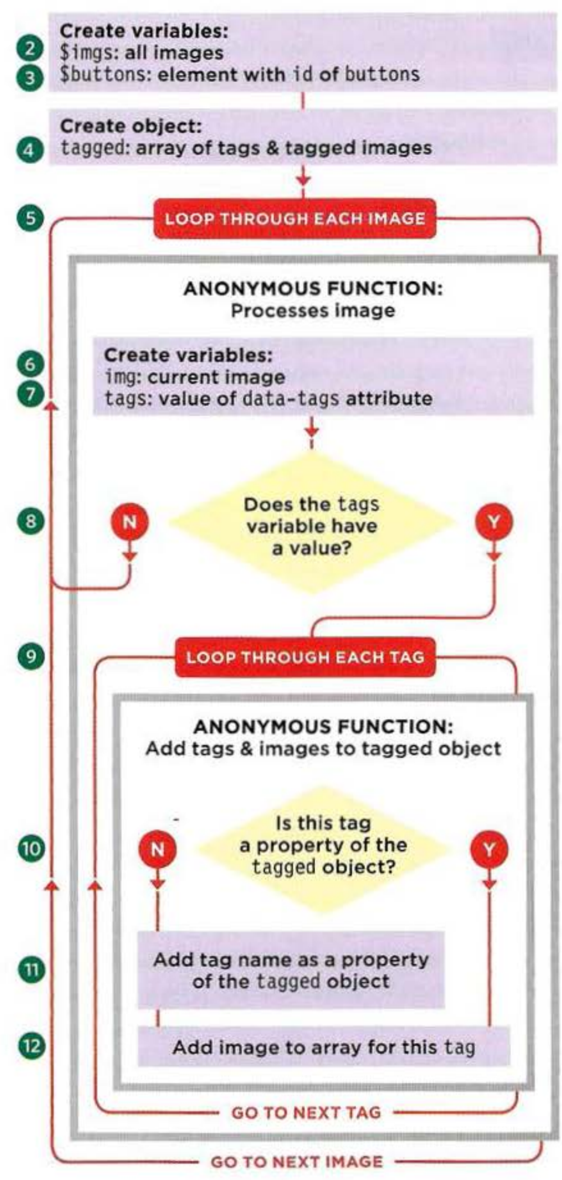
```
tagged = {
  animators: [p1.jpg, p6.jpg, p9.jpg],
  designers: [p4.jpg, p6.jpg, p8.jpg]
  filmmakers: [p2.jpg, p3.jpg, p5.jpg]
  illustrators: [p1.jpg, p9.jpg]
  photographers: [p2.jpg, p3.jpg, p8.jpg]
}
```

# 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

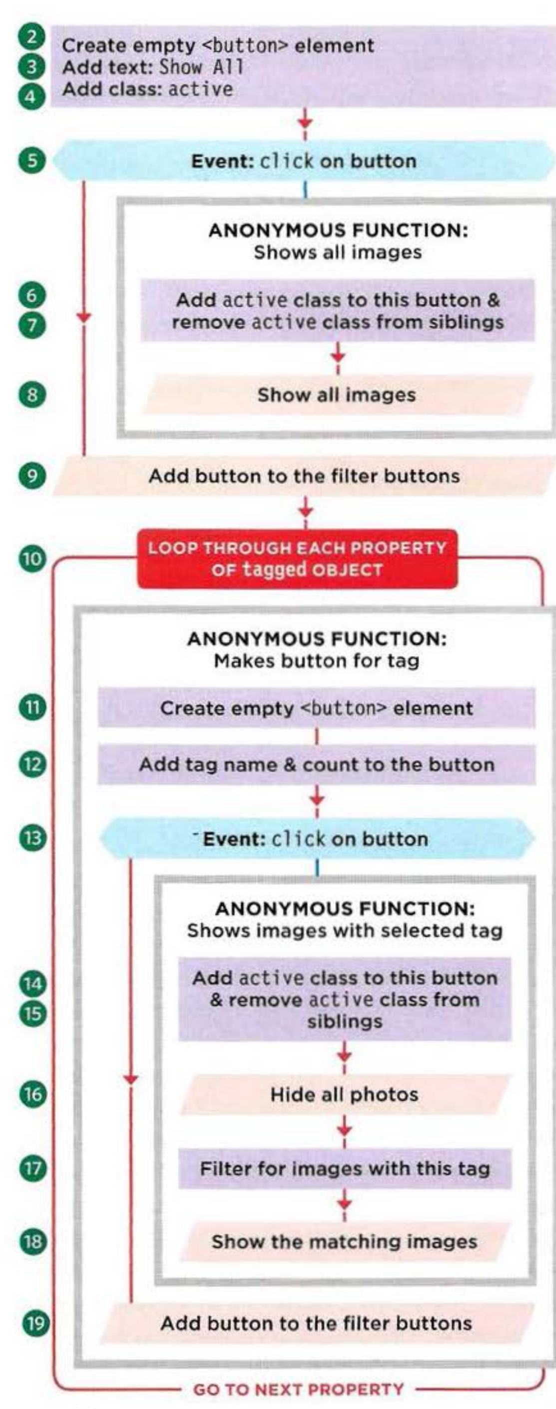
c12/js/filter-tags.js

```
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

```
① (function() {

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

②   $('<button/>', {                                  // Create empty button
③     text: 'Show All',                               // Add text 'show all'
④     class: 'active',                                // Make it active
⑤     click: function() {                             // Add onclick handler to it
⑥       $(this)                                       // Get the clicked on button
          .addClass('active')                         // Add the class of active
⑦         .siblings()                                 // Get its siblings
          .removeClass('active');                     // Remove active from them
⑧       $imgs.show();                                 // Show all images
      }
⑨   }).appendTo($buttons);                            // Add to buttons

⑩   $.each(tagged, function(tagName){                 // For each tag name
⑪     $('<button/>', {                                // Create empty button
⑫       text: tagName + ' (' + tagged[tagName].length + ')', // Add tag name
⑬       click: function() {                           // Add click handler
⑭         $(this)                                     // The button clicked on
            .addClass('active')                       // Make clicked item active
⑮           .siblings()                               // Get its siblings
            .removeClass('active');                   // Remove active from them
⑯         $imgs                                       // With all of the images
            .hide()                                   // Hide them
⑰           .filter(tagged[tagName])                  // Find ones with this tag
⑱           .show();                                  // Show just those images
        }
⑲     }).appendTo($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).

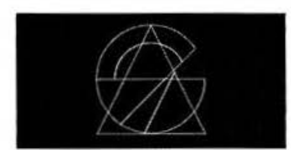
### SEARCH A CUSTOM CACHE OBJECT

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.

**CreativeFolk** find talented people for your creative projects

s



# 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

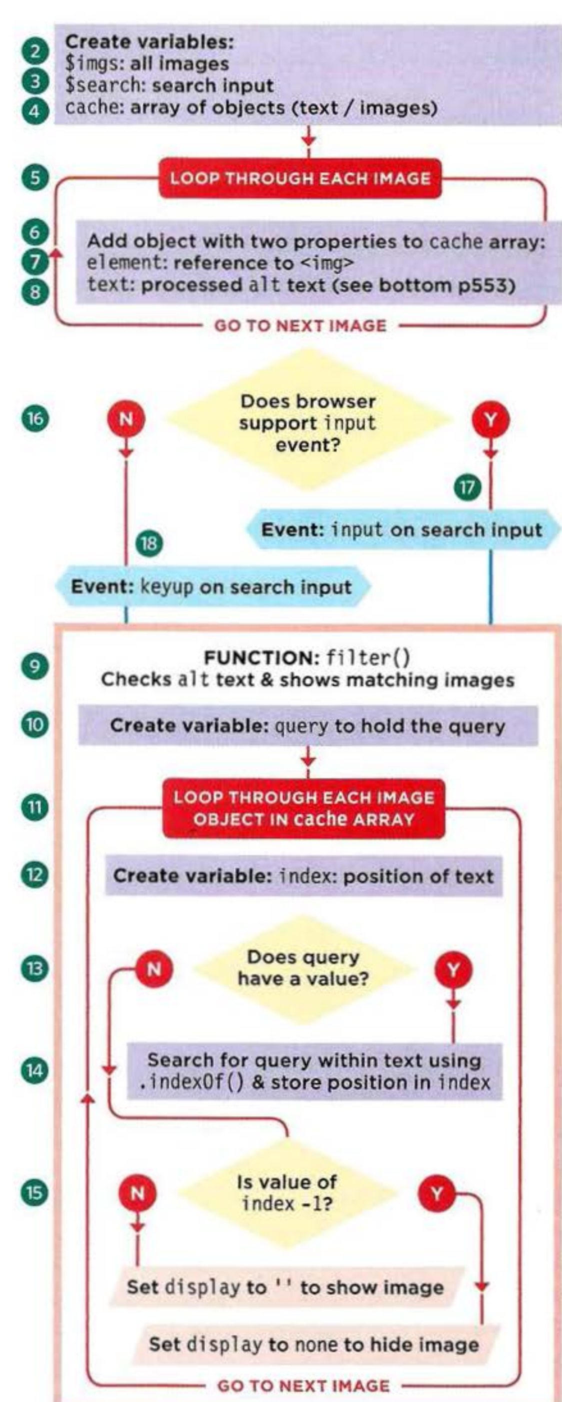
This script can be divided into two key parts:

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

```
① (function() {                             // Lives in an IIFE
②   var $imgs = $('#gallery img');          // Get the images
③   var $search = $('#filter-search');      // Get the input element
④   var cache = [];                         // Create an array called cache

⑤   $imgs.each(function() {                 // For each image
⑥     cache.push({                          // Add an object to the cache array
⑦       element: this,                      // This image
⑧       text: this.alt.trim().toLowerCase() // Its alt text (lowercase trimmed)
      });
    });

⑨   function filter() {                     // Declare filter() function
⑩     var query = this.value.trim().toLowerCase();  // Get the query

⑪     cache.forEach(function(img) {         // For each entry in cache pass image
⑫       var index = 0;                      // Set index to 0
⑬       if (query) {                        // If there is some query text
⑭         index = img.text.indexOf(query);  // Find if query text is in there
        }

⑮       img.element.style.display = index === -1 ? 'none' : '';  // Show / hide
      });
    }

⑯   if ('oninput' in $search[0]) {          // If browser supports input event
⑰     $search.on('input', filter);          // Use input event to call filter()
    } else {                                // Otherwise
⑱     $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.

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

**0**

Indicates that the items should remain in the same order

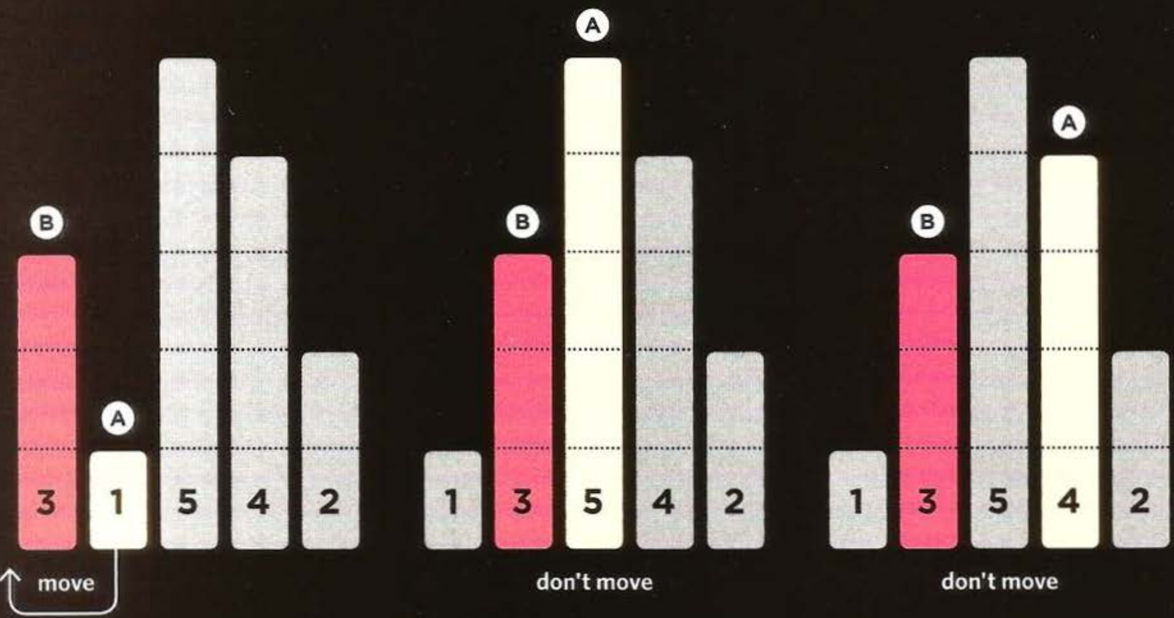
**>0**

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**

$5 - 3 = 2$

$a - b = > 0$

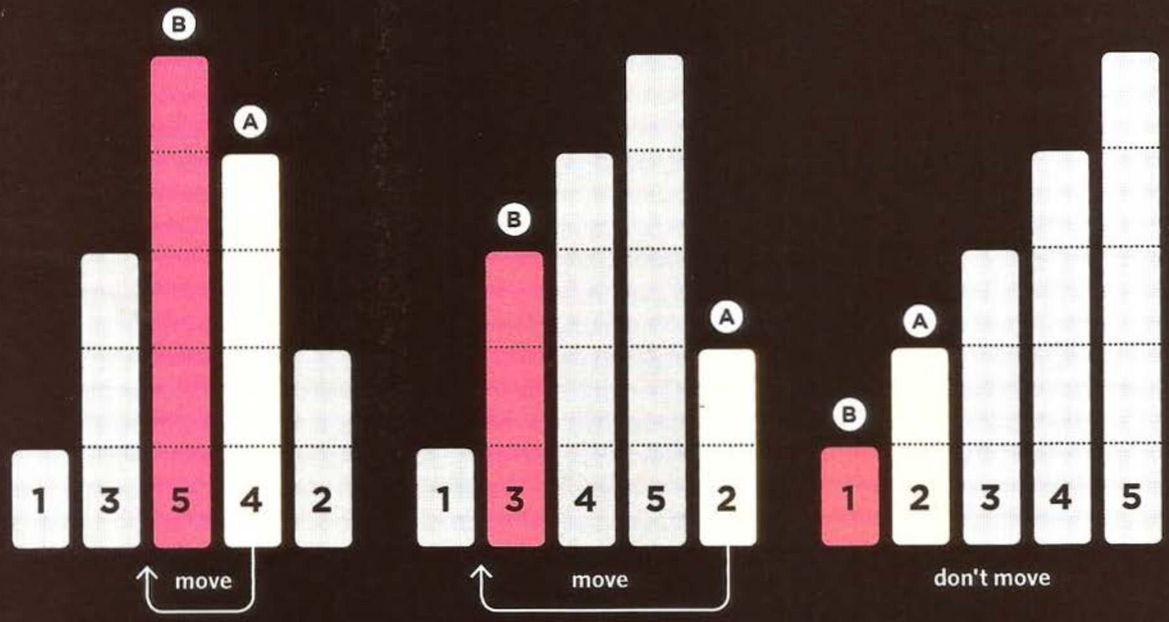
**a** should go *after* **b**

$4 - 3 = 1$

$a - b = > 0$

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** | **a** should go *before* **b** | **a** should go *after* **b** |
|---|---|---|
| 4 - 5 = -1 | 2 - 3 = -1 | 2 - 1 = 1 |
| a - b = <0 | a - b = <0 | 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

CreativeFolk find talented people for your creative projects

My Videos

Camille Berger
Paris, France

| GENRE | ▲ TITLE | DURATION | DATE |
|---|---|---|---|
| Film | Animals | 6:40 | 2005-12-21 |
| Film | The Deer | 6:24 | 2014-02-28 |
| Animation | The Ghost | 11:40 | 2012-04-10 |
| Animation | Wagons | 21:40 | 2007-04-12 |
| Animation | Wildfood | 3:47 | 2014-07-16 |

# 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>
①  <table class="sortable">
    <thead>
      <tr>
        <th data-sort="name">Genre</th>
②       <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**

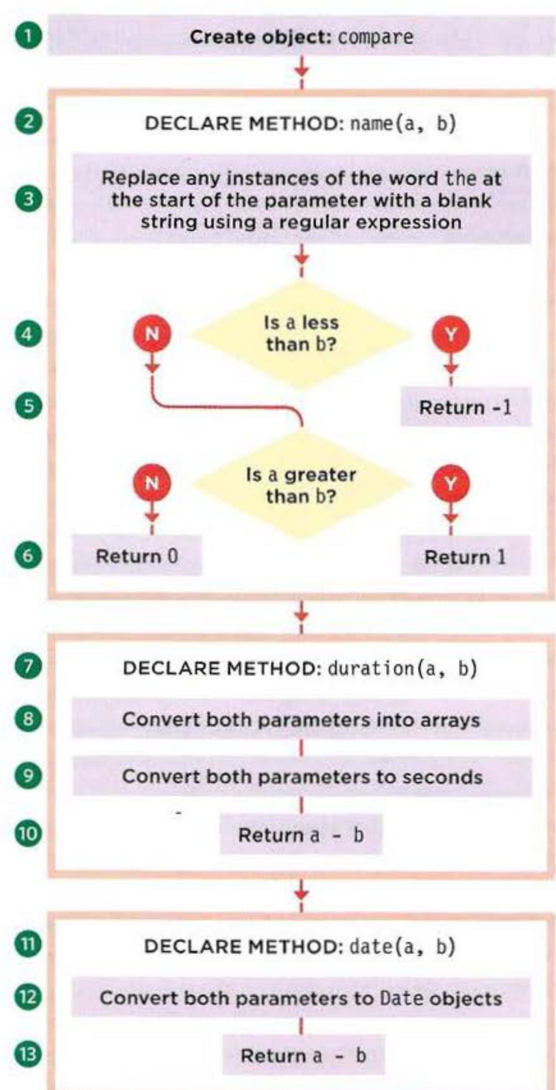
**2.** Add a method called `name()`. Like all compare functions, it should take two parameters: `a` and `b`.
**3.** Use a regular expression to remove the word 'the' from the beginning of both of the arguments that have been passed into the function (for more on this technique, see the bottom of the 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, '');`

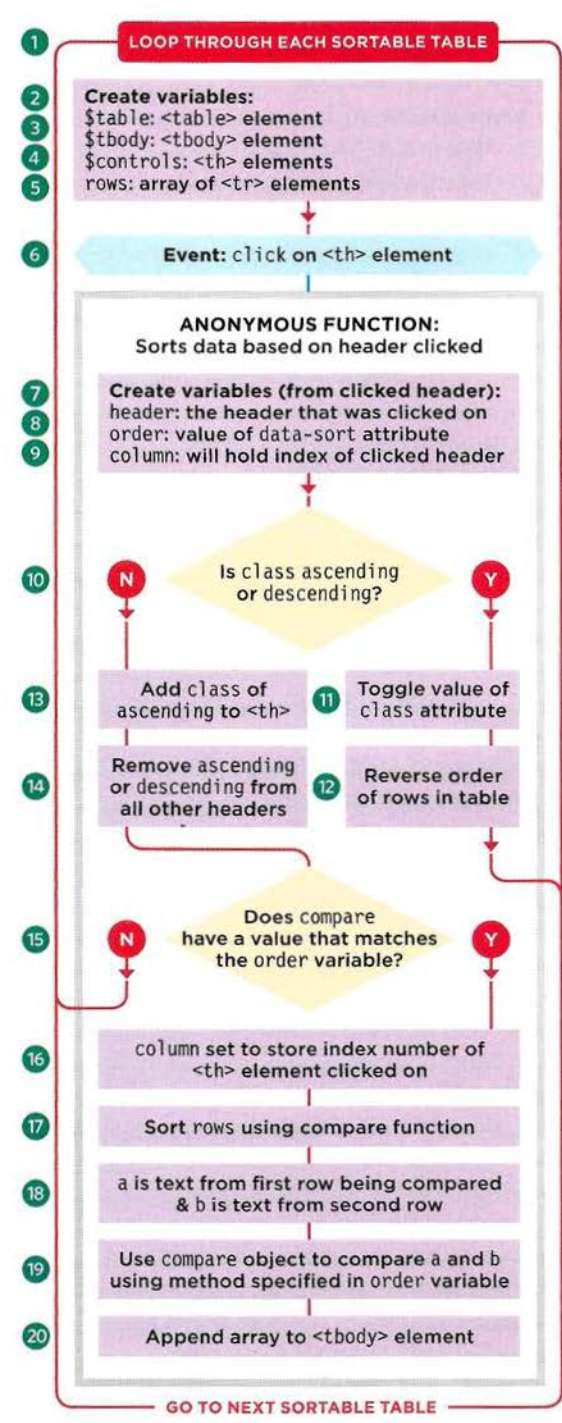
The `replace()` method is used to remove any instances of *The* from the start of a string. `replace()` works on any string and it takes one argument: a regular expression (see p612). It is helpful when *The* is not always used in a name, e.g., for band names or film titles. The regular expression is 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
⑱           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.