Computer Science 180
Web Design
Siena College
Fall 2011

Topic Notes: JavaScript

Now that we know how to put some interactive controls on a web page, let's see if we can make them do some work. We'll worry later about form submissions. For now, we'll look at making our pages responsive to user input using the JavaScript programming language.

What is JavaScript?

JavaScript is a programming language that allows us to add interactivity to web pages. It is a *scripting language*, which means that the script can execute without any compiling step.It is supported by all major browsers and is by far the most popular scripting language on the web.

We will see that even simple JavaScript code can add some impressive functionality to pages.

JavaScript Basics and Event-Driven Programming

JavaScript programs are often invoked (i.e., executed, run) in response to a user's interaction with a displayed HTML element.

This is an example of an *event-driven program*. The program does nothing except in response to *events*, often the interaction with a user through a keyboard or mouse.

Let's consider a page with a button:

See Example: alertbutton.html

The `onclick` attribute allows us to define an action called an *event handler* to take place when the mouse is clicked on the element. In this case, it occurs when the button is pressed. The action is to bring up an *alert box* with the given method.

`alert` is a built-in JavaScript *function*. This is responsible for bringing up the alert box with a message and an "OK" button. We send the function a single *parameter*, in this case the message we would like printed in our alert box. The message is sent as a *string*, in this case placed within single quotes. We would usually use double quotes, but since the value for the `onclick` attribute is already double-quoted, we use single quotes instead.

Simple Functions and Using Events with HTML Elements

We sometimes will want to have our JavaScript event handlers do other things, perhaps interacting with HTML elements on our page. The next example expands our capabilities in a number of ways.

See Example: function.html

This page has the same button we saw before, but it now calls a function `buttonPressed` instead of `alert`. While `alert` is built into JavaScript, `buttonPressed` is not. It is a function that the page will have to provide.

It is provided in a `<script>` element within the document's `<head>` element. As part of the `<script>`'s opening tag, we specify `type="text/javascript"` to indicate that the element contains JavaScript code.

The code in this case consists of one *function definition*. The function's *header* consists of the keyword `function` followed by the name of the function we are defining, in this case `buttonPressed`. The name is followed by `()` for now – we will see what we can place within the parentheses soon. Then the *body* of the function is enclosed within a `{ }` pair. The body is the JavaScript code that will be executed when the function is invoked. Here, that will happen when someone clicks on our button.

The body of the function in this case contains a single JavaScript *statement*, albeit a fairly complex one. Let's look at it piece by piece.

- `document` is a built-in JavaScript *variable* that refers to the HTML document we are dealing with. It refers to a JavaScript *object* that we can interact with to retrieve information about the document or to modify it.
- `getElementById` is a *method* of the `document` object that can look up an HTML element of our document by its `id` attribute. In this case, we look up the element with an `id` of `text`. This gives is the `<p>` element with the text "You have not yet clicked the button." If more than one element in the document has the same `id`, the first one found is the one returned.
- Each HTML element has a *property* named `innerHTML` that contains the HTML text contained within that element. We can either look at or modify this property. In this case, we modify that property to a different string. Once we do this, the page rendering is updated to have the new HTML text displayed in the element. In this case, our `<p>` element now displays "Thanks for clicking!".

Let's extend this example just a bit:

See Example: function2.html

We now have two buttons, each of which calls the `buttonPressed` function when clicked. But now, we have added a *parameter* to our function. When we call the function in the `onclick` attribute of our buttons, we specify a character string inside the parentheses following the function name.

Then, in the function definition in our `<script>` element, the word `txt` has been added inside the parentheses in the function header. One such name needs to be given for each parameter you intend to send to the function. Here, depending on which button is pressed, the name `txt` will have the value "Left button rules!" or "Right button all the way!".

Within the function, we can then use that name `txt` anywhere we would normally use a string. In this case, we use it as the new value of the `innerHTML` property. So the same function can now produce two different results, depending which button was pressed.

Changing the `innerHTML` attribute is just one thing we may choose to do in a JavaScript function. Let's see how we can change the style of an element instead.

See Example: functionstyle.html

In this case, we add a second parameter to our function which will specify a color for the text. We also rename the function to something more appropriate and descriptive for what it does: `setTextAndColor`.

The function still sets the `innerHTML` attribute but now also uses the `style` attribute of our element to change it color to the color specified by our second parameter, which is appropriately named `color`.

The syntax here is a bit different, as an element can contain many style properties. We place the property desired into `[ ]` and can then assign it a new value.

Let's make one more quick improvement to this function and expand our example to take advantage of it:

See Example: functionstylemore.html

In this example, the function has been renamed again, now to `setTextAndColorById` and takes three parameters:

- `id` is the value of an `id` attribute that will be used to look up which element is to be affected by the text and color changes.
- `txt` is the text to be placed in the element.
- `color` is the new value to give to the `color` style property of the element.

We have also added a second `<p>` element and two more buttons which will operate on that element.

Adding a Variable

Next, we will consider an example where we can keep track of how many times a button has been clicked. This requires the page not only to be able to react to a click, but to remember and update some information: the number of times the button has been clicked.

This requires that we declare and use a *variable*, which will store a number in our case. This number starts at 0 and goes up by 1 each time the button is clicked. Then, when we update the text that displays the number of clicks, we use that variable to help construct the string to be displayed.

See Example: functioncount.html

Variables in JavaScript can be *declared* either outside of any function, like this one, in which case they are *global variables*, or inside of a function, when we refer to them as *local variables*.

Global variables come into existence when the page is first loaded. They can be used and modified by any function. They cease to exist when the page is closed.

Local variables exist only while the function in which they are declared is executing. Each time the function executes, a brand new instance of the variable is created, and it does not remember any values it may have had previously.

Standard Event Attributes

It is not only buttons and not only mouse clicks that can generate events to trigger JavaScript function calls.

Each HTML element we have considered is capable of reacting to a variety of events.

Consider the `<p>` element. The following event attributes are supported:

- `onclick` – execute a script on a mouse click
- `ondblclick` – execute a script on a mouse double-click
- `onmousedown` – execute a script when the mouse button is pressed
- `onmousemove` – execute a script when the mouse pointer moves
- `onmouseout` – execute a script when the mouse pointer moves out of the element
- `onmouseover` – execute a script when the mouse pointer moves over the element
- `onmouseup` – execute a script when the mouse button is released
- `onkeydown` – execute a script when a key is pressed
- `onkeypress` – execute a script when a key is pressed and released
- `onkeyup` – execute a script when a key is released

All of these are triggered in this example:

See Example: eventattributes.html

Note that a `<p>` element is not capable of receiving keyboard events normally, but by adding the `tabindex="0"` attribute, the page "focus" can be directed to the element and it receives keyboard events.

Dynamic Page Content

In lab, you looked at an example that placed the current time and date into a page when it loads. In this example, we saw:

- The use of a `<script>` element within the `<body>` of a document.
- The use of JavaScript code outside any event handler or JavaScript function.
- The use of `document.write` to create dynamic content during the loading of a page.
- The use of the built-in `Date` object to insert the current date and time.

Retrieving Data From Form Controls

Our first method of getting meaningful input into our pages involves using JavaScript to retrieve the values in form controls and to use them to update the page appropriately.

The next two examples were also in the lab handout.

See Example: helloperson.html

- We set an `id` attribute on the text `input` so we can access it through JavaScript with `document.getElementById`.
- In our JavaScript function, we declare a *local variable* `name`, which exists only within one execution of that function.
- We assign that variable `name` the `value` attribute of the text `input`. This works like the `.innerHTML` but refers to the value typed into the text field.
- We then use that variable `name` as part of what we assign to the `.innerHTML` of the "`message`" paragraph to personalize its contents.

We can take this a step further and display only the form when the page is first rendered, then only the message after the button has been clicked:

See Example: helloperson2.html

- The "`message`" is initially an empty `<div>`, essentially just a placeholder in the page, with a name we can look up, where we can later place something else.
- We give our `<form>` element an `id`, so we can look it up using `document.getElementById`.
- The JavaScript function still retrieves the `.value` of the text input control and uses it to personalize the message. But now that message is a `<div>`, so we enclose the text we wish to display in a `<p>` element.
- Finally, the function sets the `innerHTML` of the `<form>` element to `""` – the empty string – which makes its contents empty.

The effect is to place what looks like a brand new message onto the page, and to remove the form from the page.

Text Area Input

The contents entered into a `<textarea>` are also accessed with the `.value` property.

See Example: textarea.html

So this behaves no differently than an `<input>` element of type `text`, other than that the value we type in can span multiple lines.

A couple of notes:

- Since we're simply placing the text entered into the `<textarea>` as the contents of the `<p>` element, we can enter HTML elements and they will be rendered.
- There is a commented-out line that would clear the contents of the `<textarea>` each time its contents have been used to update the paragraph.

Accessing Drop-Down Menu Values

Recall that we can build drop-down menus by defining a `<select>` element with a group of `<option>` elements within.

The following example uses a drop-down menu to define a list of colors we can use to recolor a paragraph of text.

See Example: recolorparagraph.html

When the button is clicked, we want to look at the currently-selected color in the `<select>`, and use it.

This takes a few steps.

- In the event-handler function, we first look up the `<select>` element. We save it in the variable `picker`.
- This element is used for two things: to determine the index (or number) of the currently-selected `<option>` with the `selectedIndex` property, and to obtain a list (actually, an array) of the `<option>`s, with the `options` property.
- The `selectedIndex` is used to choose the proper `option` from the `options` array.
- Once we have that option, we get its `value` attribute, which will be a valid HTML color.
- That color is then applied to the paragraph's style as in our previous examples.

We can extend this idea:

See Example: recoloreverything.html

In this example, there are two main functionality differences:

1. We can set both the foreground and background colors.
2. We can set these on several elements in the document by clicking on that element.

Several things need to change in our document to make this happen.

- Since many elements can have their colors changed, we generalize the JavaScript event-handler function to take the `id` of the element whose colors are supposed to change in the parameter `whichId`.
- We give each element whose colors we want to be able to change a unique `id` attribute and an `onclick` attribute that will call the `changeColors` function, passing the `id` as a parameter.
- We add a second `<select>`, with a different `id` from the original, that will be used to select background colors.
- We extend the body of the `changeColors` function to look up the second `<select>` and retrieve its current value. That is then used to set the background color style property. Note: in a style sheet, we would define the property `background-color`. When modifying this property through JavaScript's `.style`, we need to specify the property as `backgroundColor`.

Our next extension is to add more `<select>` elements for other properties we can apply to our changeable elements.

The procedure here is pretty straightforward, if potentially tedious. We add another `<select>` with appropriate options for each property whose values we want to be able to modify, and augment our JavaScript function to make use of each.

This is *almost* what we do in this new example:

See Example: restyleeverything.html

The big change, other than the expected ones mentioned above, is that there are now two JavaScript functions.

- The `changeOneStyle` function takes three parameters: the `id` of the element to be changed, the `id` of a `<select>` element whose currently selected option's value has a new value for the style property specified in the third parameter, `propertyName`.

- `changeStyle` now is a series of calls to `changeOneStyle`, each of which is responsible for one of the style properties we will be changing.

Essentially what this does is reduced the amount of repeated code. It makes it easier for us to add more and more styles.

Accessing the State of a Checkbox

The next type of form control we will consider for processing purposes is the checkbox.

This example shows how we can access its state:

See Example: checkbox.html

Most of what is here is familiar, and simpler than recent examples. The main item of interest is the line in the `buttonPressed` function where we set the value of the `isChecked` local variable.

We see here that an `<input>` element of type `checkbox` has an attribute we can examine named `checked`. When we use that value as part of the new `innerHTML` of the message paragraph, we see that the value is either `true`, if the box is checked, `false` otherwise.

That's great, but we don't want to print out whether the checkbox is pressed, we want to make some decisions in our program based on this.

We see how to do this in our slightly-modified example:

See Example: checkbox2.html

We are looking at a JavaScript `if` ... `else` statement, or a *conditional statement*. When the box is checked, meaning that the expression inside the parentheses following the `if` keyword will be `true`, we perform the statement inside the `{ }` pair that follows. The `else` clause that follows gives the statement that will be executed when the box is not checked (i.e., the exrpression inside the `if` is `false`.

The conditional statement is a fundamental construct in any programming language, and is what allows our program to perform different actions depending on the input it is given.

We can use these conditionals in conjunction with checkboxes to make a big improvement to the restyling example:

See Example: restylesomethings.html

- For each style property for which we are providing a changing capability, we also add a checkbox to indicate whether that should be applied.
- In `changeOneStyle`, we add a new parameter `checkId` where we provide the `id` attribute of the checkbox.
- A conditional is then used to perform the restyling of the given property only if the checkbox is currently checked.

- Note that there is no `else` on this conditional – it simply does nothing in the case that the condition inside the `if` statement is `false` (which is exactly what we want in this case).
- In `changeStyles`, each call to `changeOneStyle` is augmented to pass the appropriate checkbox `id` as the second parameter.

Accessing radio button values

We might like to think of radio buttons as interchangeable with `<select>` menus. Both let a user choose at most one of a collection of options. However, accessing the values in radio buttons is done identically to the way we access the value of a checkbox. The element corresponding to one of the radio buttons will have a `checked` attribute.

For example:

See Example: radio.html

Note that the `buttonClicked` function declares a local variable `year` that is set to the appropriate string when one of the buttons' `checked` attribute is `true`.

This function assumes that one of the buttons will be checked. If not, the variable `year` will not have a value when it gets used to set the message after the series of `if` statements.

To prevent this case from happening, some extra care is taken in this example to disable the "Press Here" button until one of the radio buttons has been selected.

To accomplish this, the `<button>` is created in a disabled mode by setting the `disabled` attribute to `disabled`. Each radio button is given an `onchange` attribute that calls the function `enableButton`. This will be called any time the status of one of the radio buttons changes. The function simply sets the "Press Here" button's `disabled` attribute to `false`, enabling it.

Once the radio buttons have some initial value, there is no way to get back to a state where none of the buttons are checked, so there is no need to "re-disable" the "Press Here" button later.

Other form inputs

The other `<input>` types: `file` and `password` work the same as the `text` type. We retrieve or set their contents by accessing the `value` attribute.