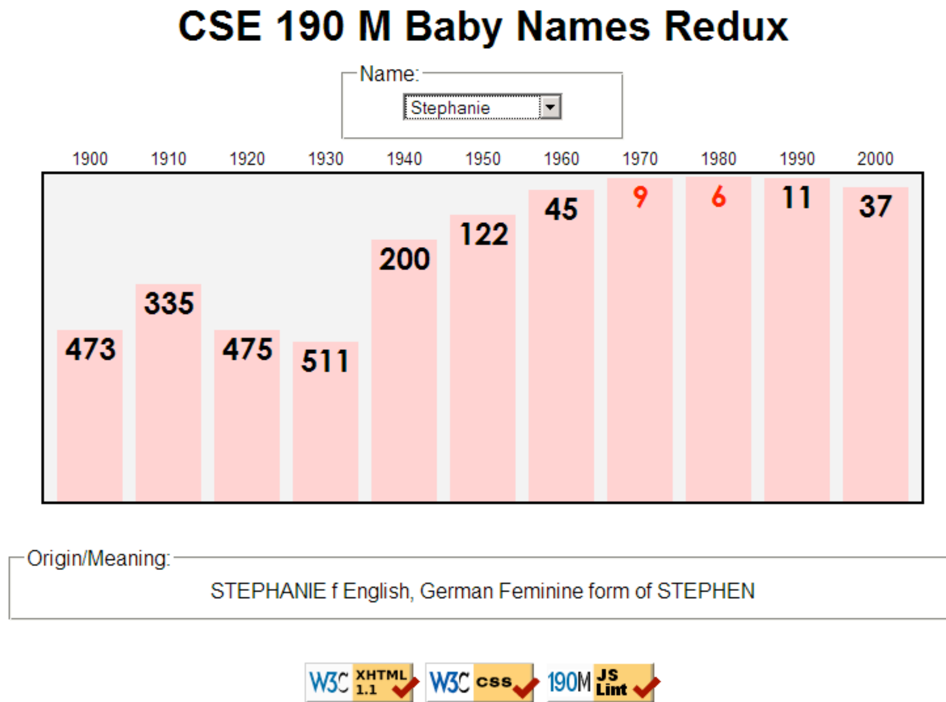
University of Washington, CSE 190 M
Homework Assignment 7: Baby Names

This assignment tests your understanding of fetching data from files and web services using Ajax requests. You must match in appearance and behavior the following web page:



Background Information:

Every 10 years, the Social Security Administration provides data about the 1000 most popular boy and girl names for children born in the US, at **http://www.ssa.gov/OACT/babynames/** . Your task for this assignment is to write the JavaScript code for a web page to display the baby names, popularity rankings, and meanings.

You do not need to submit any HTML or CSS file. We will provide you with the HTML (**names.html**) and CSS (**names.css**) code to use. Turn in only the following file:

- **names.js**, the JavaScript code for your Baby Names web page

This program uses Ajax requests to fetch data from the Webster server. Since an Ajax request can only connect to the same web server a page is located on, **you must upload your page to Webster to test it**. (If you try to fetch data from Webster while viewing the page from your local hard drive, the request will fail with an exception.)

Data:

Your program will get its data from a web service located at the following URL:

https://webster.cs.washington.edu/cse190m/babynames.php

This web service has three "modes," each of which provides a different type of data. Specify which type of data you want from it by changing the parameter named `type`. Each query returns either plain text or XML as its output. If you make a malformed request, such as one missing a necessary parameter, the service will respond with an HTTP error code of 400 rather than the default 200. If you request data for a name the server doesn't have data for, the service will respond with an error code of 404. *(Hint: You can test queries by typing the URL of the web service, along with the appropriate query string parameters, in your web browser's address bar and seeing the result. You can also use the Net tab of Firebug to view the response details of an Ajax request.)*

1. list: The first type of query is `list`, which returns plain text containing all baby names on file, with each on its own line. The following query would return the results below (shortened by ...):

https://webster.cs.washington.edu/cse190m/babynames.php?type=list

```
Aaliyah
Aaron
Abagail
...
```

(Order of the names is not particularly important, but you should use the names in the order provided, which you can assume will be alphabetical.)

2. rank: The second type of query is `rank`, which requires a second parameter called `name` and returns an XML document containing the provided baby name's popularity rank for each decade from 1900 to 2000.

The overall XML document tag is called `<baby>` and has an attribute name for the baby's first name. Inside each `<baby>` tag are `<rank>` tags, one for each decade. Each `<rank>` tag contains an attribute `year` representing the year of the data reading, and the text content inside the `<rank>` tag is that name's popularity ranking in that year.

The ranking value will be between 1 (popular) and 999 (unpopular), or 0. A rank of 0 means the name was not in the top 1000. The following query would return the results below:

https://webster.cs.washington.edu/cse190m/babynames.php?type=rank&name=morgan

```
<baby name="Morgan">
    <rank year="1900">430</rank>
    <rank year="1910">477</rank>
    ...
    <rank year="2000">25</rank>
</baby>
```

Although the web service currently always returns exactly 11 years' worth of data, from 1900 to 2000, your program should not rely on this. Your code should examine the XML to determine the year for each ranking.

3. meaning: The third type of query is `meaning`, which requires a second parameter called `name` and returns a plain-text description of the meaning of the provided name. The following query would return the results below:

https://webster.cs.washington.edu/cse190m/babynames.php?type=meaning&name=martin

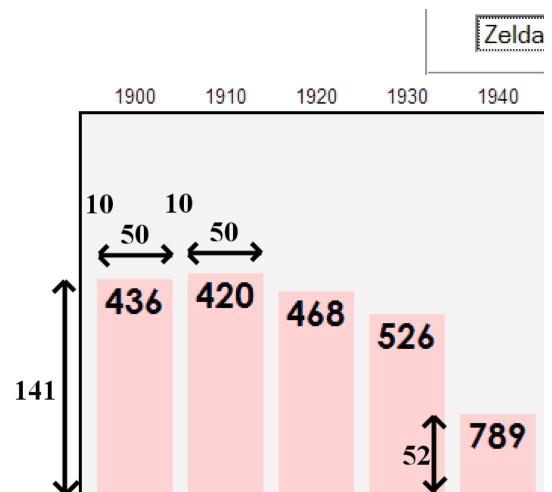
```
MARTIN m English, French, German From the Roman name Martinus, which was
derived from Martis, the genitive case of the name of Roman god MARS.
```

All names returned by `list` have ranking data, but not all will have meaning data (such as "Mogran"). If the requested name has no meaning associated with it, nothing at all should be displayed in the Origin/Meaning area.

You may assume that all XML and text data sent to your program is valid and does not contain any errors.

Appearance and Behavior:

The HTML page given to you shows a heading followed by a select element with `id` of `babyselect`. When the page first loads, the select box will be disabled and contain no baby names. In the background, JavaScript code should fetch the list of names from the web service (described in the "Data" section above), fill the select box with an `option` element for each name, and then enable it.



Beneath the select box, in the center of the page, is a large 670×250px shaded bar graph section that is initially empty. When a name is selected from the dropdown, JavaScript code should fetch ranking data from the web service (described in the "Data" section above) and populate the graph with bars indicating that name's relative popularity for each year in the data. If a different name had previously been selected, the data for that name will need to be cleared from the graph before the new data is added. If the user selects the dummy "Select a name..." prompt from the list rather than a name, nothing whatsoever should happen.

The bars are positioned absolutely with respect to the overall shaded graph section. Each ranking bar's width is 50px. The height of each bar, in px, is calculated by taking the inverse of its ranking—that is, subtracting its ranking from 1000—and taking ¼ of that. For example, the ranking of 880 leads to a bar with a height of 30px (one fourth of 1000 − 880). (Note that division is exact in JavaScript—`13 / 4` is `3.25`—so you'll want to round down any height values you compute using `Math.floor` or `parseInt`.) The bars' *x*-coordinates are such that the first bar's left edge is 10px from the left edge of the graph section, and there is 10px of horizontal space between each bar. The first bar's left edge is at $x = 10$px, the second is at $x = 70$px, the third at $x = 130$px, and so on. All bars' *y*-positions are such that their bottom edges are flush with the bottom edge of the shaded graph area.

Within each ranking bar appears its ranking number. The ranking numbers appear in an 18pt sans-serif font, horizontally centered at the top of the bars. Note that some less popular rankings (around 900 and up) have numbers that drop below the graph region's bottom edge or overlap its black border. You shouldn't attempt to "fix" this; it's expected behavior.

The provided CSS file contains a class called `ranking` that contains all necessary CSS styles for these bars—with the exception of the *x*-position and height, which are unique to each bar. You should create a DOM element for each bar and assign this class to it. The element itself constitutes the bar, and its text content is the ranking.

If the ranking is very popular (1 through 10 inclusive), it should appear in red. There is a class called `popular` in the provided CSS file that contains the styles necessary for this.

Just above the shaded graph region, directly above each ranking bar, are labels representing the corresponding years. These are vertically positioned 1.5em above the top edge of the shaded graph and centered horizontally within the width of the ranking bar. The provided CSS file contains a class called `year` that contains all necessary styles for these elements—with the exception of their *x*-positions, which are unique to each year label.

Underneath the graph area appears a paragraph of text with information about the meaning of the selected name. When a new name is chosen from the select box, that name's meaning information should be loaded from the web service (described in the "Data" section above) and appear in the paragraph with the `id` of `meaning`. If no meaning is known to the web service, nothing at all should be displayed in this area.

Error messages: If an error or exception occurs during a `list` or `rank` Ajax request, your program should show a descriptive error message about what went wrong. Error handling for a `meaning` request is a little different: it is expected that meanings will not be found for some names (such as "Mogran"). If a name has no meaning data, an error message should **not** be shown (and no text should appear in the Meaning/Origin area). However, if there is an exception or a malformed request, an error message should still be shown, as with any other request. *(Hint: To differentiate between errors, look into Prototype's `on###` handlers or `ajax.status` property.)*

For full credit, your error message should not be an `alert`; it must be injected into the page as HTML content. The exact format of the error message is up to you, but it should at least include the HTTP error code and some descriptive error text.

All other style elements on the page are subject to the preference of the web browser. The screenshots in this document were taken on Windows XP in Firefox 3.5, which may differ from your system.

Extra Features:

A separate specification document on the course web site describes several additional features you can complete to add functionality to your program. But you must complete at least one of these extra features as part of your assignment; in other words, **one extra feature is required**. The one required feature can be any extra feature of your choice. If you complete more than one extra feature, you can receive additional points/rewards as described in the other spec document.

Implementation and Grading:

Submit your work from the course web site. For reference, our .js file has roughly 95 lines (54 "substantive").

Fetch the necessary data for the program using **Ajax** requests. We suggest using Prototype's `Ajax.Request` and `Ajax.Updater` objects rather than the raw `XMLHttpRequest` object, but either approach is acceptable if the code is not redundant. Process XML data by examining the request's XML tree using the XML DOM.

Make extra effort to minimize **redundant code**. Capture common operations as functions to keep code size and complexity from growing. You can reduce your code size by using the `this` keyword in your event handlers.

For full credit, your JavaScript code should pass the provided **JSLint** tool with no errors reported. You should follow reasonable style guidelines similar to those of a CSE 14x programming assignment. In particular, minimize global variables, avoid redundant code, and use parameters and return values properly.

A few **global variables** are allowed, but it is not appropriate to declare lots of them; values should be local as much as possible. If a particular constant value is used frequently throughout your code, declare it as a global "constant" variable named `IN_UPPER_CASE` and use the constant throughout your code.

You should separate content (XHTML), presentation (CSS), and behavior (JS). As much as possible, your JS code should **use styles and classes from the CSS** rather than manually setting each style property in the JS.

For full credit, you must write your code using **unobtrusive JavaScript**, so that no JavaScript code, `onclick` handlers, etc. are embedded into the XHTML code.

Your JavaScript code should have adequate **commenting**. The top of your file should have a descriptive comment header describing the assignment, and each function and complex section of code should be documented.

Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and method names. Avoid lines of code or comments more than 100 characters wide.

Do not place a solution to this assignment on a public web site. Upload your files to the **Webster** server at:

https://webster.cs.washington.edu/*your_uwnetid*/hw7/names.html

© Copyright Marty Stepp / Jessica Miller, licensed under Creative Commons Attribution 2.5 License.