

Lab 4.4

Secret Messages: Indexing, Arrays, and Iteration

This JavaScript lab (the last of the series) focuses on indexing, arrays, and iteration, but it also provides another context for practicing with functions and web forms. The page you will create in this lab allows the user to encrypt and decrypt “secret messages.” You can read more about encryption and decryption in Chapter 17, but it is not formally required reading for the lab, because this is not the primary focus of the lab. We briefly explain all of the basic encryption concepts and vocabulary in the lab.

Vocabulary

All key vocabulary used in this lab are listed below, with closely related words listed together:

- iteration
- array
- element
- index
- cleartext/plaintext vs. cipher text
- encrypt vs. decrypt
- key

Post-lab Questions

Write your answers after completing the lab, but read them carefully now and keep them in mind during the lab.

1. For the encryption algorithm described in the lab, explain why a key of 95 is not very useful.
2. As we discuss later, the “shifting” encryption method used in this lab is too simple to ensure any real secrecy. If you were given a message encrypted using the shifting algorithm but were not given the key, how could you use a computer program to help you quickly decrypt the message?
3. Assume that `i` is a declared variable in the code fragments below. For how many iterations will each of these loops run?

```
for (i = -2; i <= 2; i = i + 1) {
```

```

    alert("another iteration");
}

for (i = 128; i >= 2; i = i / 2) {
    alert("another iteration");
}

for (i = 1; i <= 100; i = 2 * i) {
    alert("another iteration");
}

```

Discussion and Procedure

As with the other JavaScript labs, we'll produce a small, web-based program by the end of this lab. The program will let the user encrypt and decrypt short "secret messages." (*Encryption* is the technical term for converting messages into a form that's only readable to the intended recipient, e.g., by using some secret code. *Decryption* is the reverse process the intended recipient uses to recover the original message.) As you'll see soon, the encryption algorithm we'll use in this lab is actually quite simplistic, but it will give us a chance to practice working with arrays and iteration.

Part 1. User interface

The suggested starting user interface is shown at the right. It's mostly composed of form elements you're already familiar with: text box, buttons. However, you'll notice that there's a new kind of form element: the multi-line text entry area. Text areas are sizeable and are better suited for text input that is longer than a few words. We'll give you the HTML tags necessary to set up the three text areas on this page.

To encrypt a message, the user enters it in the top text area and clicks Encrypt. Your program will use the key value to encrypt the message and put the result in the middle



text area. To decrypt a message, the user needs to use the exact same key value that was used to encrypt it. The user enters the encrypted version of the message in the middle text area and clicks Decrypt to get the decrypted version in the lower text area.

1. *Start a new HTML file and begin setting up the interface with the text box for the key.*

The tag for a text area looks like this and uses its own tag, rather than the single **input** tag, like buttons and text boxes:

```
<textarea rows=6 cols=30 name=clearText></textarea>
```

Size of the text area can be set using the **rows** (height) and **cols** (width, where “cols” is short for columns) attributes.

2. *Add the text areas and buttons to the web page.* You can put them all in one form or, if you wish, split the interface into multiple forms, but remember to give each form a unique name.

Part 2. “Code-shifting” encryption/decryption

Back in Chapter 8 (Bits and the “Why” of Bytes), we discussed how computers store text (as well as other kinds of data). One common way that computers store text is by representing each character using an integer according to a standard code called ASCII. (The ASCII table is such an essential resource for programmers that you should have no trouble finding a copy on the web by searching for “ascii table.”) Here’s an example string and its representation as an array of integers:

Lee Morgan!

76 101 101 32 77 111 114 103 97 110 33

Every character, including spaces and punctuation marks, has a corresponding ASCII code number. ASCII codes 32 through 126 correspond to the printable characters needed for English, including the alphabet in upper and lower case, numbers, punctuation marks, common typewriter symbols (@#\$%^&* etc.), and the space character. (Codes less than 32 are “non-printing” characters, as is 127, which is actually the code for Delete. This will require us to add a few steps to our algorithm later.)

It’s easier to understand the encryption algorithm we’ll use in this lab if you think of text in this numeric form. The basic idea of the algorithm is to convert the original text into ASCII code numbers, add a small number (which we call the key) to the codes (“shifting” the code), then convert the codes back into characters. Let’s try encrypting the example phrase above using key value 17.

First, we shift the codes by adding 17 to each of them:

76 101 101 32 77 111 114 103 97 110 33

93 118 118 49 94 128 131 120 114 127 50

Let's consider what happens if we proceed to convert these codes back into their character representations:

]vv1^⊠⊠xr⊠2

The ⊠ symbols stand for characters that are either non-printing or are outside the standard range of ASCII codes for English (the original 7-bit ASCII code). The problem with these characters is that they might not display properly in a text area on a web page, and they might not also properly copy-and-paste into an e-mail or some other program you might want to use to exchange encrypted messages with friends.

In order to avoid the problem of ending up with these “bad” characters, we will add another step before converting the codes back to characters, so let's go back to the codes for our example phrase, after we have shifted them:

93 118 118 49 94 128 131 120 114 127 50

It's really only the numbers that are greater than 126 that we need to worry about. The strategy for dealing with these numbers is to convert them to codes at the beginning of the printable ASCII code range, which starts at 32. The partial table at the right should give you an idea of how we'll do this. For any code greater than 126, we just “start” over starting at 32, the first printable ASCII code. As long as the key value isn't too large, this keeps all of the resulting codes within the 32 to 126 range.

127	⇒	32
128	⇒	33
129	⇒	34
130	⇒	35
131	⇒	36
132	⇒	37
133	⇒	38
...		

93 118 118 49 94 128 131 120 114 127 50

93 118 118 49 94 33 39 120 114 32 50

Now if we convert the codes back to characters, we get a string that can be reliably displayed and copied-and-pasted:

]vv1^!'xr 2

That completes the encryption process. Before you think about decryption, try another example yourself, to make sure you understand the algorithm:

3. Use an ASCII table and your age as the key to encrypt this phrase. Show your work, including the ASCII code version of the phrase, the codes after adding the key and correcting any large codes, and the encrypted characters you end up with.

original phrase: **Who is Grace Hopper?**

4. Given what you know about this encryption algorithm, try to reverse the process and decrypt the following string. Use key 20. (The encrypted message is the answer to the question you encoded above: “Who is Grace Hopper?”)

HINT: Just like we did for encryption at first, start by ignoring the fact that some of the codes might have been corrected for being too large after shifting. Then, consider how you can recognize which codes ended up too large during encryption and ended up being converted. [verify that some of the codes in this example actually got rotated to the lower part of the range; adjust key if not]

encrypted string: }#+y#)\$'4\$z4)|y4w\$"%}!y'

Not appropriate for state secrets. In case you were wondering, the encryption method we use here is hardly secure and is more for fun than anything else. (Writing code to provide more secure encryption is a little too complicated for this lab.) Interestingly enough, until 1997, the United States government considered some stronger, computer-based encryption algorithms so valuable (potentially for military applications) that it regulated their export as a munition! See Chapter 17 for discussion of stronger, more complicated encryption algorithms than the one in this lab.

Part 3. Implementing encryption

We’ll leave the task of writing the JavaScript code for the web page mostly to you, but we’ll provide a few essential technical details and suggestions.

First, there’s the task of converting the provided text into ASCII code numbers. We suggest you write a function called **stringToASCII** to do this. The function should take a string as a parameter and return an array of numbers containing the ASCII codes. You’ll find the following built-in JavaScript functions for working with strings and characters useful:

```
// a string to use in the examples below
var sampleString = "Hopper";

// getting a string's length
var stringLength = sampleString.length;

// getting the ASCII code of a character in a string;
// stores 112 (code for "p") in thirdCharCode
var thirdCharCode = sampleString.charCodeAt(2);
```

Note that the `charCodeAt` function indexes strings starting at 0 and ending with one less than the string length, just like with indexing of arrays. This is why 2 is passed to `charCodeAt` in the example above to get the third character's ASCII code.

5. Determine how you would set up a for loop to iterate through each character in a string. More specifically, try filling in the blanks below to ensure that the variable `i` starts at the first valid index and goes up to (but not past) the last valid index. The first blank should contain a value, the second should contain a Boolean expression, and the last blank should contain a statement to update the index variable `i`.

```
var i; // string index
for (i = _____; _____; _____) {
  // some code to work with someString.charCodeAt(i);
}
```

As an intermediate goal, before completing `stringToASCII`, consider writing a function that doesn't work with arrays:

6. Write a function similar to but simpler than `stringToASCII` that takes a string as a parameter and outputs (to a text area) a list of the ASCII codes corresponding to the characters in the string. This function doesn't return anything and doesn't even have to use an array, in fact. This could help you focus your attention on working with a string one character at a time. It also gives you an opportunity to verify that you're converting characters to ASCII codes properly.

HINT: Remember to put spaces (" ") between the ASCII code numbers as you concatenate them onto the end of a string to put into the text area. Otherwise, you'll have trouble reading the displayed codes.

Returning an array from a function is straightforward. The example function below, which just creates an array and returns it, shows you the syntax. Note that you do not use any square brackets with the array name when returning it from a function. (Square brackets are only used when working with a specific element of an array.) Although this function has no parameter and doesn't set the array contents, you could use it as a starting point for writing your `stringToASCII` function.

```
function sampleArrayFunction() {
  var arrayToReturn = new Array(10); // 10 elements
  return arrayToReturn;
}
```

You'll also need to know how to reverse the process and convert an array of ASCII codes to a string (i.e., a `ASCIIToString` function). This example code shows you how you can use a function `String.fromCharCode`, which takes an ASCII code and returns a

one-character string containing the character corresponding to the code.

```
// stores "p" in charFromCode
var charFromCode = String.fromCharCode(112);
```

7. *Implement the string and ASCII code array conversion functions (both directions) and test them.*

HINT: Choose one to implement and test before you move on to the reverse conversion. To test your function, you can reference an ASCII code table and make sure the conversion is working properly.

Once you have both functions working, you can try converting a string to an array of integers and back to make sure you recover the original string.

Don't just dive into writing the rest of the code right away. If you don't have a clear and complete plan for all of the code you need to write, you're likely to waste time getting stuck or starting over. We've identified a couple of functions that will be useful but small enough to write and test individually, i.e., before combining them with other functions for the page.

8. *Identify and plan the writing of other functions that this page will require. What functions will be called when the buttons are clicked? More importantly, what functions will those functions call to accomplish the encryption, decryption, and output of results to the text areas?*

HINT: We suggest you implement small functions that process one ASCII code at a time. In other words, have a function that takes an ASCII code and a key value and does the code shifting and whatever correction is necessary.

With that, we leave you to apply the lessons learned from the four programming labs to complete the encryption/decryption page. Make sure to test your page with a variety of different inputs (messages and keys). Consider pairing up with someone else in the class and exchanging encrypted messages. (You need to both agree on a key value to share messages, of course.)

Bug-tracking: Where programming meets databases. Now that you've seen debugging (Lab 2.1), databases (Labs 3.2–4), and programming, you might be interested to know that many software companies use specialized database software to help teams of software and quality assurance engineers communicate and work together. These databases, often known as "bug-tracking systems," store information about tests, testing results, procedures for reproducing bugs, status on fixing bugs, etc., serving as a centralized location where everyone involved in a software project (engineers and managers alike) can record and view how the project is progressing. For example, the engineers who work on the Netscape/Mozilla web browser use a system called Bugzilla. You can read more about it at <http://bugzilla.org/about.html>. IBM and NASA also have projects that use Bugzilla, as of June 2003.

Further Reading

- Chapter 17 (for more about encryption, decryption)