#### My own views on ArrayBuffer

 ArrayBuffer is a JavaScript object that allows you to store and manipulate binary data, such as images, audio, and video. It's like an array that can only store binary data and has a fixed length.  
  
To use an ArrayBuffer, you can create a new instance of it by specifying the number of bytes you want to allocate. You can then create views of the buffer, which are used to read and write to specific parts of the buffer.  
  
For example, if you have an ArrayBuffer of 16 bytes and you want to store four 32-bit integers, you can create an Int32Array view of the buffer, which will allow you to read and write to the buffer as if it were an array of integers.  
  
ArrayBuffer is commonly used in web applications to handle binary data, but it's important to use it securely to prevent security vulnerabilities.

#### Daily Notes - ArrayBuffer

 ArrayBuffer is a JavaScript object that allows you to store and manipulate binary data, such as images, audio, and video. It's like an array that can only store binary data and has a fixed length.  
  
To use an ArrayBuffer, you can create a new instance of it by specifying the number of bytes you want to allocate. You can then create views of the buffer, which are used to read and write to specific parts of the buffer.  
  
For example, if you have an ArrayBuffer of 16 bytes and you want to store four 32-bit integers, you can create an Int32Array view of the buffer, which will allow you to read and write to the buffer as if it were an array of integers.  
  
ArrayBuffer is commonly used in web applications to handle binary data, but it's important to use it securely to prevent security vulnerabilities.

#### Think and Reflect

 TypedArray is a built-in object in JavaScript that provides a way to work with arrays of typed data, such as integers or floating-point values, in a more efficient and controlled manner. It is part of the ArrayBuffer object, which represents a fixed-length binary data buffer.  
  
There are several types of TypedArrays available in JavaScript, including Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, and Float64Array. Each type represents a different type of data and provides methods for working with that data in a more efficient and performant way.  
  
Using TypedArray, you can create a new buffer of a specified size, allocate memory for it, and read and write data to and from the buffer. For example, you can create a new Uint8Array with a length of 10:  
  
javascript  
Copy code  
let buffer = new ArrayBuffer(10);  
let uint8View = new Uint8Array(buffer);  
You can also initialize a TypedArray with an array of values:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
To read or modify the values in a TypedArray, you can use standard array indexing syntax:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
console.log(uint8View[0]); // 1  
uint8View[0] = 10;  
console.log(uint8View[0]); // 10  
TypedArray also provides several useful methods for working with the data, such as slice(), subarray(), and set(). For example, you can use the slice() method to create a new TypedArray that is a subset of an existing one:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
let slicedView = uint8View.slice(2, 4);  
console.log(slicedView); // Uint8Array [3, 4]  
Overall, TypedArray provides a powerful and efficient way to work with typed data in JavaScript, making it a useful tool for many applications, such as game development, data visualization, and scientific computing.

#### Daily Notes - TypedArray

 TypedArray is a built-in object in JavaScript that provides a way to work with arrays of typed data, such as integers or floating-point values, in a more efficient and controlled manner. It is part of the ArrayBuffer object, which represents a fixed-length binary data buffer.  
  
There are several types of TypedArrays available in JavaScript, including Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, and Float64Array. Each type represents a different type of data and provides methods for working with that data in a more efficient and performant way.  
  
Using TypedArray, you can create a new buffer of a specified size, allocate memory for it, and read and write data to and from the buffer. For example, you can create a new Uint8Array with a length of 10:  
  
javascript  
Copy code  
let buffer = new ArrayBuffer(10);  
let uint8View = new Uint8Array(buffer);  
You can also initialize a TypedArray with an array of values:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
To read or modify the values in a TypedArray, you can use standard array indexing syntax:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
console.log(uint8View[0]); // 1  
uint8View[0] = 10;  
console.log(uint8View[0]); // 10  
TypedArray also provides several useful methods for working with the data, such as slice(), subarray(), and set(). For example, you can use the slice() method to create a new TypedArray that is a subset of an existing one:  
  
javascript  
Copy code  
let uint8View = new Uint8Array([1, 2, 3, 4, 5]);  
let slicedView = uint8View.slice(2, 4);  
console.log(slicedView); // Uint8Array [3, 4]  
Overall, TypedArray provides a powerful and efficient way to work with typed data in JavaScript, making it a useful tool for many applications, such as game development, data visualization, and scientific computing.

#### Daily Notes - Out-of-bounds behaviour

 In JavaScript, accessing an array index that is out-of-bounds of the array's size will result in undefined behavior.  
  
Another approach is to use built-in array methods like Array.prototype.slice() or Array.prototype.splice(), which automatically handle out-of-bounds cases and return empty arrays or modified arrays, respectively.  
  
Overall, it is important to be aware of the potential for out-of-bounds behavior in JavaScript and to take steps to prevent it in your code.

#### Daily Notes - TypedArray methods

 TypedArrays in JavaScript provide several methods for manipulating and working with their contents. Here are some of the commonly used methods:  
  
TypedArray.from(iterable[, mapFn[, thisArg]]): Creates a new TypedArray from an iterable object, such as an array or a string. The optional mapFn parameter allows you to transform the elements during the creation process.  
  
TypedArray.of(...items): Creates a new TypedArray with the given elements as its contents.  
  
TypedArray.prototype.copyWithin(target, start[, end]): Copies a sequence of elements within the same TypedArray, optionally overwriting existing elements. The target parameter is the index to copy the sequence to, while start and end specify the start and end indices of the sequence to copy.  
  
TypedArray.prototype.fill(value[, start[, end]]): Fills the entire TypedArray or a subset of it with a given value. The optional start and end parameters specify the range of indices to fill.  
  
TypedArray.prototype.filter(callback[, thisArg]): Creates a new TypedArray with all elements that pass a test implemented by the provided callback function. The thisArg parameter specifies the value to use as this when executing the callback.  
  
TypedArray.prototype.find(callback[, thisArg]): Returns the value of the first element in the TypedArray that satisfies the provided testing function callback, or undefined if no such element is found.  
  
TypedArray.prototype.includes(searchElement[, fromIndex]): Determines whether the TypedArray includes a certain value searchElement, optionally starting the search at a specified fromIndex.  
  
TypedArray.prototype.join([separator]): Joins all elements of the TypedArray into a string, with an optional separator string between each element.  
  
TypedArray.prototype.map(callback[, thisArg]): Creates a new TypedArray with the results of calling a provided callback function on every element in the TypedArray. The thisArg parameter specifies the value to use as this when executing the callback.  
  
TypedArray.prototype.reverse(): Reverses the order of the elements in the TypedArray.  
  
TypedArray.prototype.slice([begin[, end]]): Returns a new TypedArray that is a shallow copy of a portion of the original TypedArray. The optional begin and end parameters specify the start and end indices of the slice.  
  
TypedArray.prototype.sort([compareFunction]): Sorts the elements of the TypedArray in place according to a provided compareFunction.  
  
TypedArray.prototype.subarray([begin[, end]]): Returns a new TypedArray that references a subset of the original TypedArray. The optional begin and end parameters specify the start and end indices of the subset.  
  
TypedArray.prototype.toString(): Returns a string representation of the TypedArray and its contents.  
  
TypedArray.prototype.valueOf(): Returns the TypedArray itself as its primitive value.  
  
These methods provide a range of functionality for working with TypedArrays in JavaScript, and can be very useful for efficiently manipulating large sets of data.

#### Daily Notes - DataView

 Here are some common methods provided by DataView:  
  
new DataView(buffer[, byteOffset[, byteLength]]): Creates a new DataView object that is bound to the given ArrayBuffer, SharedArrayBuffer, or TypedArray. The optional byteOffset and byteLength parameters allow you to specify a subset of the buffer to work with.  
  
DataView.prototype.getInt8(byteOffset[, littleEndian]): Returns a signed 8-bit integer (byte) at the specified byte offset. The optional littleEndian parameter specifies the byte order.  
  
DataView.prototype.getUint8(byteOffset[, littleEndian]): Returns an unsigned 8-bit integer (byte) at the specified byte offset. The optional littleEndian parameter specifies the byte order.  
  
DataView.prototype.getInt16(byteOffset[, littleEndian]): Returns a signed 16-bit integer (short) at the specified byte offset. The optional littleEndian parameter specifies the byte order.  
  
DataView.prototype.getUint16(byteOffset[, littleEndian]): Returns an unsigned 16-bit integer (unsigned short) at the specified byte offset. The optional littleEndian parameter specifies the byte order.

#### Daily Notes - Activity 2 - Applying new Concepts

 Implementing CSP: Content Security Policy (CSP) is a powerful security feature that helps to prevent cross-site scripting (XSS) attacks, clickjacking, and other code injection attacks. By adding a CSP header to your website's HTTP response, you can restrict the sources from which your site can load scripts, stylesheets, images, and other resources. To implement CSP, you would need to modify your web server configuration or use a library to add the CSP header to your HTTP response.  
  
Using SameSite cookies: SameSite cookies are a way to prevent cross-site request forgery (CSRF) attacks. By setting the SameSite attribute to "Strict" or "Lax", you can ensure that cookies are only sent with requests that originate from the same site as the cookie. To use SameSite cookies, you would need to modify your website's cookie settings to include the SameSite attribute.  
  
Using TypedArrays and DataViews: TypedArrays and DataViews are powerful features of JavaScript that allow you to work with binary data in a more efficient and flexible way. You can use TypedArrays to create and manipulate arrays of binary data, and DataViews to access and modify the individual bytes of a binary buffer. To use TypedArrays and DataViews, you would need to write JavaScript code that creates and manipulates these objects.  
  
Preventing out-of-bounds behavior: To prevent out-of-bounds behavior in JavaScript, you can use defensive programming techniques such as checking array bounds and validating user input. By checking the length of an array before accessing an element, you can prevent errors that occur when you try to access an element that doesn't exist. To validate user input, you can use regular expressions or other techniques to ensure that input values are within a valid range.  
  
Overall, implementing these techniques can help to improve the security and performance of your website, and using TypedArrays and DataViews can enable you to work with binary data in a more efficient and flexible way.

#### Daily Notes - Activity 1 - Code Practice

 To concatenate an array of Uint8Arrays into a single array, you can use the Uint8Array constructor and the set() method. Here is an example implementation of the concat() function:  
  
javascript  
Copy code  
function concat(arrays) {  
// Compute the total length of the concatenated array  
let totalLength = arrays.reduce((acc, array) => acc + array.length, 0);  
  
// Allocate a new Uint8Array with the total length  
let result = new Uint8Array(totalLength);  
  
// Copy each array into the result array using set()  
let offset = 0;  
for (let array of arrays) {  
result.set(array, offset);  
offset += array.length;  
}  
  
return result;  
}

#### My Views on the Day

 What were the important features of the day?  
learning about dataviews and arrayBuffers  
What activities were beneficial?  
What activities (if any) were too easy?

#### Daily Notes - Day 4 Reflections

 What were the important features of the day?  
learning about dataviews and arrayBuffers  
What activities were beneficial?  
yes it was

Copyright © 2013–2015 OpenCraft, Harvard, edX, McKinsey, and The People's Science, released under the [APGLv3 license](https://github.com/open-craft/problem-builder/blob/master/LICENSE)