Set 32: Traversing Arrays

Skill 32.01: Apply loops to iterate (traverse) an array

Skill 32.01 Concepts

In programming, we use lists to store sequences of related data. We often want to perform the same operation on every element in a list, like displaying each element or manipulating them mathematically. To do that, we can use a loop to **iterate or traverse** over each element, repeating the same code for each element. The video below explains,

Recall that to iterate over all the elements in an array we need to know the address of the first element and the last element. For the houses array below, the length is 4. However, do not confuse length with the location of the last element! Notice, that because array indexing begins at 0, the last index is always 1 less than the length, or 3, in the example below.

The code below could be used to access the first and last element in the houses array shown above,

```
var firstElement = houses[0]; //Bart
var lastElement = houses[houses.length-1]; //Marvin
```

Now that we know the address of the first and last element, we can apply the loops we've learned previously to iterate over any array. In each example, we will iterate over the array below,

```
var houses = [];
houses[0] = "Bart";
houses[1] = "Kyle";
houses[2] = "Bugs";
houses[3] = "Marvin";
```

for-loop

Code	Output

`for(var i = 0; i < houses.length; i++){`
`  console.log(houses[i]);`
`}`
	Bart
	Kyle
	Bugs
	Marvin

That code above could be used to display *every single element in our list*, no matter how many values there are. Let's break down how it works.

The 3-part loop header controls how many times it repeats:

- **var i = 0**: This initializes the counter variable i to 0. In JavaScript, the first element in an array has an index of 0, so that is always the start value for a loop that iterates through an array from start to finish.
- **i < houses.length**: This condition checks to make sure that the counter variable i is less than the length of the houses array. Programming languages always provide a way to find out the length of the array, and in JavaScript, you can find out with the array.length property.
- **i++** : This is executed after each iteration of the loop, and adds one to the counter variable. If we added two, we'd only process every other element. Adding one ensures we process *every* element once.

The body of the for loop contains the code that will be run for each iteration. In this case, it's a single line of code that displays the current element:

```
console.log(houses[i]);
```

There's one really important thing to notice about this line of code: the index inside the brackets isn't a number like 0 or 1 or 2. Instead, it's the counter variable i. That means that the index will change each time this code is run, starting with 0, then 1, then 2.

while-loop

All for loops can also be written as while-loops. Below illustrates how the houses array can be traversed with a while loop.

Code	Output

`var i = 0;`
`while(i < houses.length){`
`  console.log(houses[i]);`
`  i++;`
`}`
	Bart
	Kyle
	Bugs
	Marvin

The while loop above has the same 3 parts as the for loop, they are just arranged differently. In the while loop above, *i* is initialized before the loop and incremented inside the body of the loop. The stop conditional is the only argument in the header of the loop.

Skill 32.01 Exercises 1 & 2

Skill 32.02: Combine computation with iteration

Skill 32.02 Concepts

Of course, we can do much more than simply display the elements in a list! For example, we can compute new values based on all the values in the list. The code below computes a total price based on a list of prices of individual products,

```
var totalPrice = 0;
var prices = [1.75, 3.50, 4.99, 2.50];
for (var i = 0; i < prices.length; i++) {
   totalPrice += prices[i];
}
```

Let's get a bit fancier: we can use conditionals inside a loop to compute values based on which elements satisfy a condition.

The code below computes the number of freezing temperatures in a list of temperatures from a 10-day period,

```
var numFreezing = 0;
var temps = [29, 33, 31, 30, 28, 33, 35, 34, 32, 28];
for (var i = 0; i < temps.length; i++) {
   if (temps[i] <= 32) {
      numFreezing++;
   }
}
```

We could even create an entirely new list while iterating through a list.

The code below processes a list of prices as well, like the first example, but it instead calculates the discounted price of each item (25% off) and adds that price to a new list:

```
var prices = [39.99, 29.99, 19.99];
var discountedPrices = [];
for (var i = 0; i < prices.length; i++) {
    var newPrice = prices[i] * 0.75;
    discountedPrices.push(newPrice);
}
```

[Skill 32.02 Exercise 1]

Skill 32.03: Avoid out of bounds errors

Skill 32.03 Concepts

Consider a list with 8 elements, like this one storing the planets of our solar system,

```
var planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "
Neptune"];
```

In JavaScript (and any language that uses 0-based indexing), the first element is `planets[0]` and the eighth element is `planets[7]`.

What would the following code display, then?

```
console.log(planets[8]);
```

In JavaScript, it displays undefined. In many languages, it causes a runtime error, preventing the rest of the code from running. In Python, you'll see a "list index out of range" error and in Java, you'll see "java.lang.ArrayIndexOutOfBoundsException".

We never want to cause an out of bounds error in any language, so as programmers, we need to be make sure we specify in-bounds indices for our lists.

Here's a for loop with an out of bounds error:

```
for (var i = 0; i <= planets.length; i++) {
    println(planets[i]);
}
```

It looks very similar to the correct for loop from before, but there's one tiny but crucial difference. Do you see it?

The condition in the header uses <= instead of < when comparing `i` to `planets.length`. That means the loop *will* execute when `i` equals `planets.length`. In this case, that's when `i` equals 8. However, as we've just discussed, `planets[8]` is undefined. In languages with 0-based indices, the highest index is `array.length - 1`, *not* `array.length`.

To prevent programmers from causing an out of bounds error, many programming languages provide specialized for loops designed just for iterating through lists, known as for-each loops.

[Skill 32.03 Exercise 1]

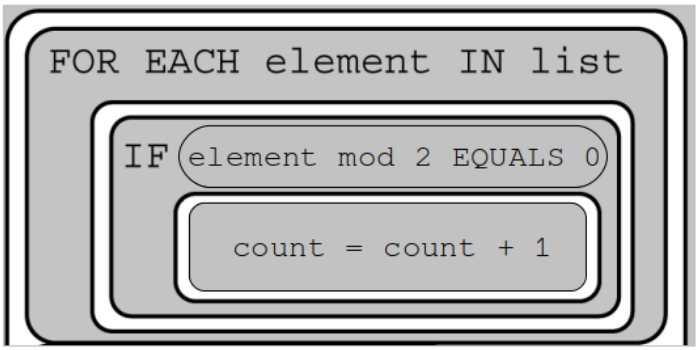
Skill 32.04: Interpret for-each statement pseudocode

Skill 32.04 Concepts

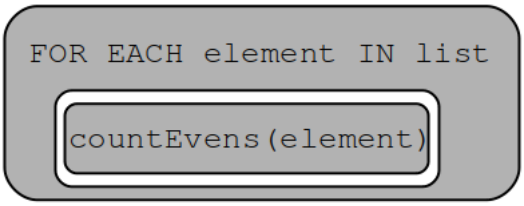
A for-each loops allows us to easily access each element in an array. Then once the element is accessed we can perform a function on the element. The diagram below illustrates the components of a *for-each* loop.

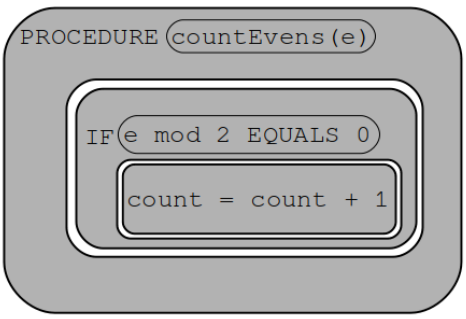
FOR EACH element IN list **doSomething(element)**	**FOR EACH** – key word for accessing an element **IN** – key word for referrencing the list **element** – the element in the list **list** – the name of the list **doSomething(element)** – the action we want to perform on each element in the list. It can be a function call or just a simple console log.

In the example below we use a *for-each* loop to count all the even numbers in a list. For each element in the list, we are checking if it is divisible by two, and if it is, we increment *count*.



The example above could have also been written as follows,





In the above example, the procedure *countEvens* is called each time an element is found in the list. Each element is passed to the procedure *countEvens*, where *count* is incremented if the element is even.

[Skill 32.04 Exercises 1 thru 3]