Basic 04 – Data Structure

# Exercise 01: Sum of a range

Write a range function that takes two arguments, start and end, and returns an array containing all the numbers from start up to (and including) end.

Next, write a sum function that takes an array of numbers and returns the sum of these numbers. Run the example program and see whether it does indeed return 55.

As a bonus assignment, modify your range function to take an optional third argument that indicates the “step” value used when building the array. If no step is given, the elements go up by increments of one, corresponding to the old behavior. The function call range(1, 10, 2) should return [1, 3, 5, 7, 9]. Make sure it also works with negative step values so that range(5, 2, -1) produces [5, 4, 3, 2].

|  |
| --- |
| // Your code here.  console.log(range(1, 10));  // → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  console.log(range(5, 2, -1));  // → [5, 4, 3, 2]  console.log(sum(range(1, 10)));  // → 55 |

# Exercise 02: Reversing an array

Arrays have a reverse method that changes the array by inverting the order in which its elements appear. For this exercise, write two functions, reverseArray and reverseArrayInPlace. The first, reverseArray, takes an array as argument and produces a *new* array that has the same elements in the inverse order. The second, reverseArrayInPlace, does what the reversemethod does: it *modifies* the array given as argument by reversing its elements. Neither may use the standard reverse method.

Thinking back to the notes about side effects and pure functions in the [previous chapter](https://eloquentjavascript.net/03_functions.html#pure), which variant do you expect to be useful in more situations? Which one runs faster?

|  |
| --- |
| // Your code here.  console.log(reverseArray(["A", "B", "C"]));  // → ["C", "B", "A"];  let arrayValue = [1, 2, 3, 4, 5];  reverseArrayInPlace(arrayValue);  console.log(arrayValue);  // → [5, 4, 3, 2, 1] |

# Exercise 03: A List

Objects, as generic blobs of values, can be used to build all sorts of data structures. A common data structure is the *list* (not to be confused with array). A list is a nested set of objects, with the first object holding a reference to the second, the second to the third, and so on.

|  |
| --- |
| let list = {  value: 1,  rest: {  value: 2,  rest: {  value: 3,  rest: null  }  }  }; |

The resulting objects form a chain, like this:

A nice thing about lists is that they can share parts of their structure. For example, if I create two new values {value: 0, rest: list} and {value: -1, rest: list} (with list referring to the binding defined earlier), they are both independent lists, but they share the structure that makes up their last three elements. The original list is also still a valid three-element list.

Write a function arrayToList that builds up a list structure like the one shown when given [1, 2, 3] as argument. Also write a listToArrayfunction that produces an array from a list. Then add a helper function prepend, which takes an element and a list and creates a new list that adds the element to the front of the input list, and nth, which takes a list and a number and returns the element at the given position in the list (with zero referring to the first element) or undefined when there is no such element.

If you haven’t already, also write a recursive version of nth.

|  |
| --- |
| // Your code here.  console.log(arrayToList([10, 20]));  // → {value: 10, rest: {value: 20, rest: null}}  console.log(listToArray(arrayToList([10, 20, 30])));  // → [10, 20, 30]  console.log(prepend(10, prepend(20, null)));  // → {value: 10, rest: {value: 20, rest: null}}  console.log(nth(arrayToList([10, 20, 30]), 1));  // → 20 |

# Exercise 04: Deep comparison

The == operator compares objects by identity. But sometimes you’d prefer to compare the values of their actual properties.

Write a function deepEqual that takes two values and returns true only if they are the same value or are objects with the same properties, where the values of the properties are equal when compared with a recursive call to deepEqual.

To find out whether values should be compared directly (use the === operator for that) or have their properties compared, you can use the typeof operator. If it produces "object" for both values, you should do a deep comparison. But you have to take one silly exception into account: because of a historical accident, typeof null also produces "object".

The Object.keys function will be useful when you need to go over the properties of objects to compare them.

|  |
| --- |
| // Your code here.  let obj = {here: {is: "an"}, object: 2};  console.log(deepEqual(obj, obj));  // → true  console.log(deepEqual(obj, {here: 1, object: 2}));  // → false  console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));  // → true |