Reverse 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 `reverse` method does: it *modifies* the array given as argument by reversing its elements. Neither may use the standard `reverse` method.

console.log(reverseArray(["A", "B", "C"]));

// → ["C", "B", "A"];

let arrayValue = [1, 2, 3, 4, 5];

reverseArrayInPlace(arrayValue);

console.log(arrayValue);

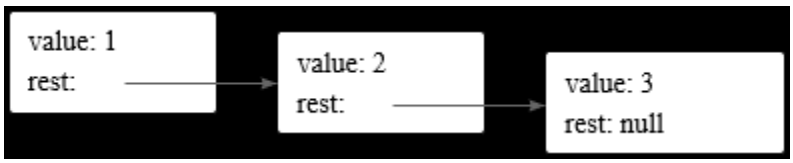
// → [5, 4, 3, 2, 1]

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 `listToArray` function 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`.

```
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
```

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

```
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
```