JavaScript: Array API

Lecture 24

Arrays: Basics

- Numbered starting at 0
- Indexed with `[ ]`
- Property length is # of elements

```
let sum = 0;
for (let i = 0; i < n.length; i++) {
  sum += n[i];
}
```

- Iteration over elements with `for…of` loop

```
let product = 1;
for (const e of n) {
  product *= e
}
```

Array Instantiation/Initialization

- Instantiate with new

```
let n = new Array(3);
```

- Initially, each element is empty
- Note: Elements can be a mix of types

```
n[0] = 10;
n[1] = "hi";
n[2] = new Array(100);
```

- Array literals usually preferred

```
let n = [10, 20, 30, 40];
let m = ["hi", , "world", 3.14];
[3, "hi", 17, [3, 4]].length == 4
```

Dynamic Size

- Arrays can grow

```
let n = ["tree", 6, -2];
n.length == 3 //=> true
n[8] = 17;
n.length == 9 //=> true
```

- Arrays can shrink

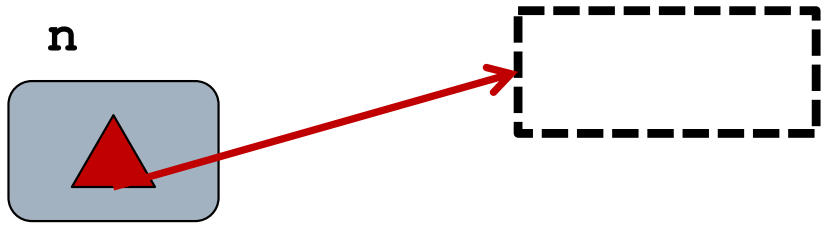
```
n.length = 2;
// n is now ["tree", 6 ]
```

Arrays are Dynamic

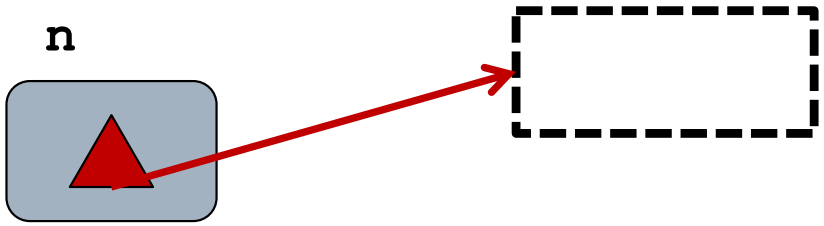

```
let n = [];
```

Arrays are Dynamic


```
let n = [];
```

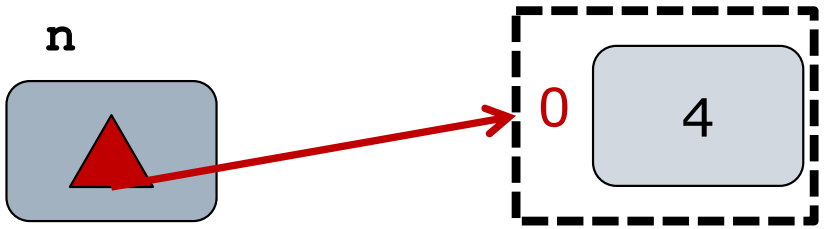


Arrays are Dynamic

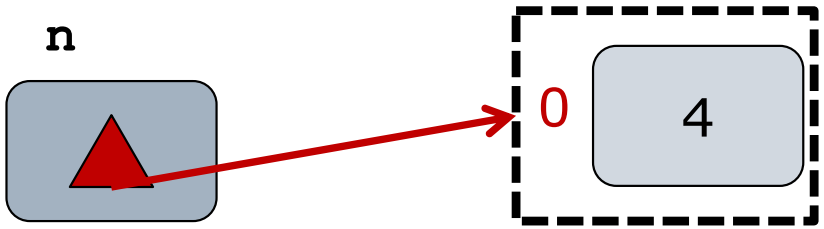



```
n[0] = 4;
```

Arrays are Dynamic

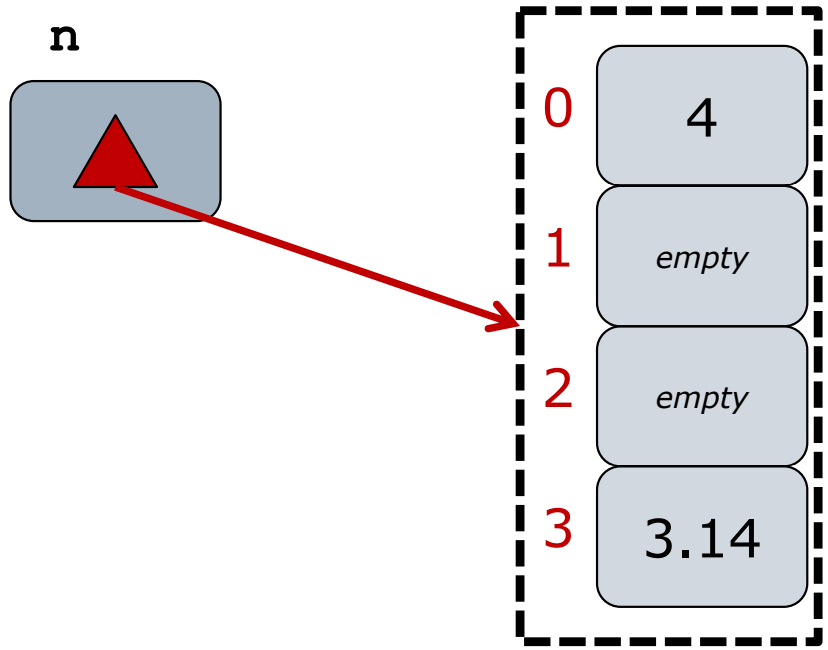


Arrays are Dynamic

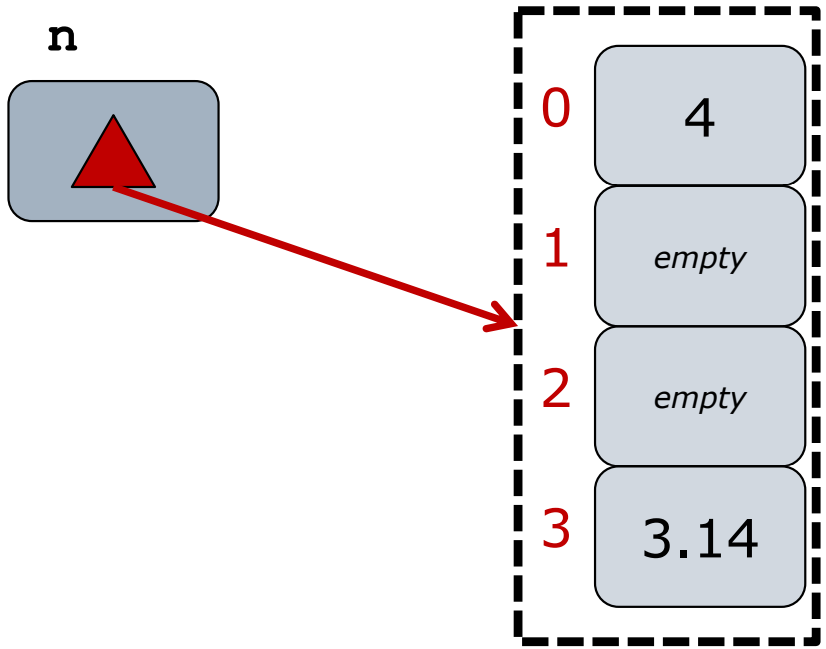



```
n[3] = 3.14;
```

Arrays are Dynamic

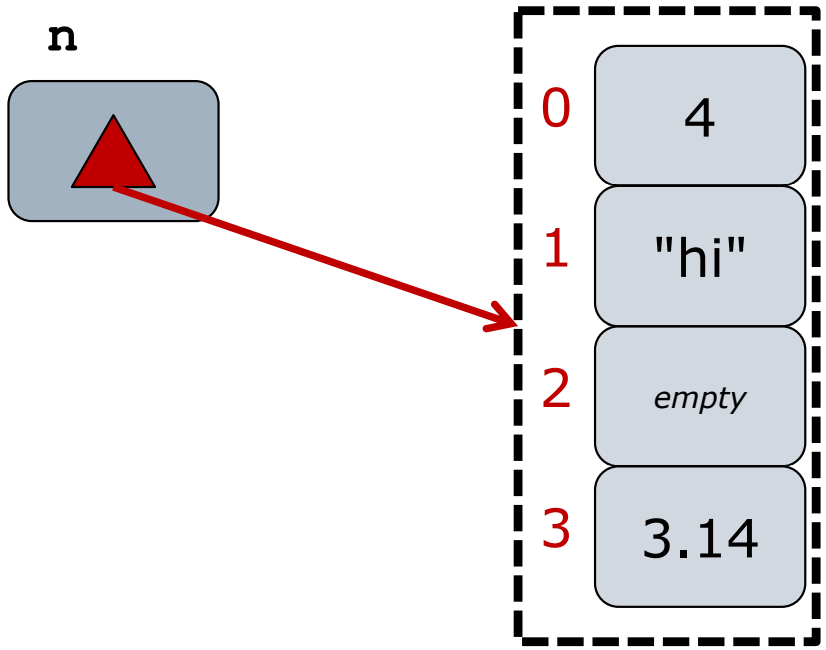


Arrays are Dynamic




```
n[1] = "hi";
```

Arrays are Dynamic



Accessors: Searching

- Find occurrence: indexOf/lastIndexOf
  - Returns -1 if not found

    `indexOf(`*`element`*`[, `*`startIndex`*`])`

    `lastIndexOf(`*`element`*`[, `*`lastIndex`*`])`
  - Optional parameter: start/end index
  - Uses strict equality (===)

```
let i = n.indexOf(elt);
while (i != -1) {
    report(i);
    i = n.indexOf(elt, i + 1);
}
```

Accessors: Extracting

- None of the following change the array
  - Return a new array/string with result
- Concatenate: **`concat`**

```
concat(a1, a2, …, aN)
let d = n.concat(n);
```

- Extract a sub-section: **`slice`**

```
slice(startIndex, endIndex)
k = n.slice(1, 3); // k is n[1], n[2]
```

- Combine into string: **`join`**

```
join(separator)
s = n.join(" "); // default is ","
```

Mutators: Growing/Shrinking

- Add/remove from end: **push/pop**

```
let n = [10, 20];
newLength = n.push(30, 40); //=> 4
lastValue = n.pop(); //=> 40
```

- Add/remove from beginning: **unshift/shift**

```
let n = [10, 20];
newLength = n.unshift(30, 40); //=> 4
firstValue = n.shift(); //=> 30
```

- Push/shift gives FIFO queue

Push Example


```
function findAll(n, elt) {
  let indices = [];
  let i = n.indexOf(elt);
  while (i != -1) {
    indices.push(i);
    i = n.indexOf(elt, i + 1);
  }
  return indices;
}
```

Mutators: Delete/Insert/Replace

- Delete/insert/replace sub-array: **`splice`**

  `splice (`*`index`*`, `*`howMany`*`[, `*`e1`*`, `*`e2`*`, …, `*`eN`*`])`

  - Modifies array (*cf.* **`slice`**, an *accessor*)
  - Returns array of removed elements

```
let magic = [34, -17, 6, 4];
let removed = magic.splice(2, 0, 13);
// removed is []
// magic is [34, -17, 13, 6, 4]

removed = magic.splice(3, 1, "hi", "yo");
// removed is [6]
// magic is [34, -17, 13, "hi", "yo", 4]
```

Mutators: Rearrange

- Transpose all elements: **reverse**

```
let n = [5, 300, 90];
n.reverse(); // n is [90, 300, 5]
```

- Order all elements: **sort**

```
let f = ["blue", "beluga","killer"];
f.sort(); // f is
          // ["beluga", "blue", "killer"]
n.sort(); // n is [300, 5, 90]
```

Mutators: Rearrange

- Transpose all elements: **reverse**

```
let n = [5, 300, 90];
n.reverse(); // n is [90, 300, 5]
```

- Order all elements: **sort**

```
let f = ["blue", "beluga", "killer"];
f.sort(); // f is
          // ["beluga", "blue", "killer"]
n.sort(); // n is [300, 5, 90]
```

- Problem: Default ordering is based on string representation (lexicographic)
- Solution: Use a function that compares

Sorting with Comparator

- A comparator (a, b) returns a number
  - < 0 iff a is *smaller than* b
  - == 0 iff a is *same size* as b
  - > 0 iff a is *greater than* b
- Examples

```
function lenOrder(a, b) {
  return a.length - b.length;
}
function compareNumbers(a, b) {
  return a - b;
}
```

Sorting with Comparator

- Optional argument to sort

  sort([*compareFunction*])

- Example

```
names.sort(lenOrder);
n.sort(compareNumbers);

n.sort(function(a, b) {
  return a - b;
});

n.sort((a, b) => a - b);
```

Iteration: Logical Quantification


```
let isBig = (elt, index, array) => {
  return (elt >= 10);
}
```

- Universal quantification: **every**

```
[5, 8, 13, 44].every(isBig); // false
[51, 18, 13, 44].every(isBig); // true
```

- Existential quantification: **some**

```
[5, 8, 13, 44].some(isBig); // true
[5, 8, 1, 4].some(isBig); // false
```

- Neither modifies original array

Iteration: Filter

- Pare down an array based on a condition: **`filter`**

  `filter(`*`predicate`*`)`

  ***`predicate`*`(`*`element`*`, `*`index`*`, `*`array`*`)`**

- Returns a new array, with elements that satisfied the predicate
  - Does not modify the original array
- Example

```
t = [12, 5, 8, 13, 44].filter(isBig);
```

Iteration: Map

- Transform an array into a new array, element by element: map
  - E.g. an array of strings into an array of their lengths
  - ["hi", "there", "world"] → [2, 5, 5]

  map(*callback*)

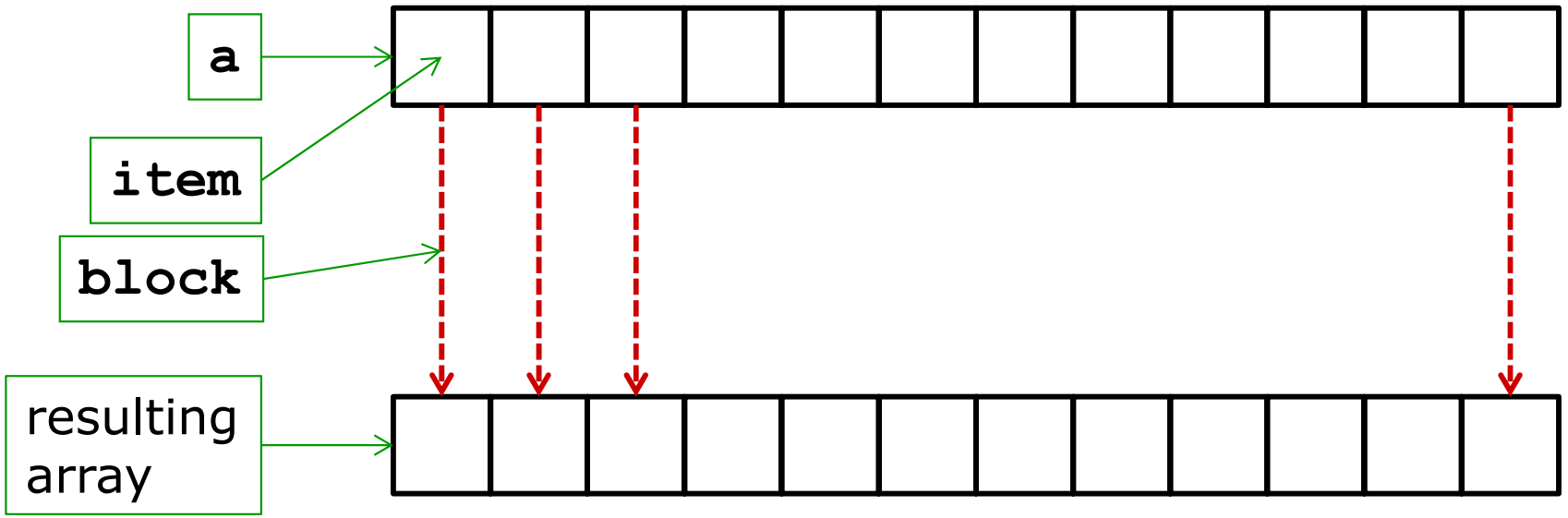
  ***callback*(*element*, *index*, *array*)**
- Examples

```
len = names.map(function(elt, i, a) {
  return elt.length
});
names.map(w => w[0].toUpperCase());
```

Recall: Ruby Map

- Transform an array into a new array, *element by element*
- Uses *block* to calculate each new value

```
a.map { |item| block }
```



Iteration: For Each

- Similar to map, but preferred for side-effects and changing an array in place

  forEach(*callback*)

  ***callback*(*element*, *index*, *array*)**

- Example

```
let logArrayElts = (elt, i, arr) => {
  console.log("[" + i + "] = " + elt);
}
[2, 5, 9].forEach(logArrayElts);
```

Iteration: Reduce

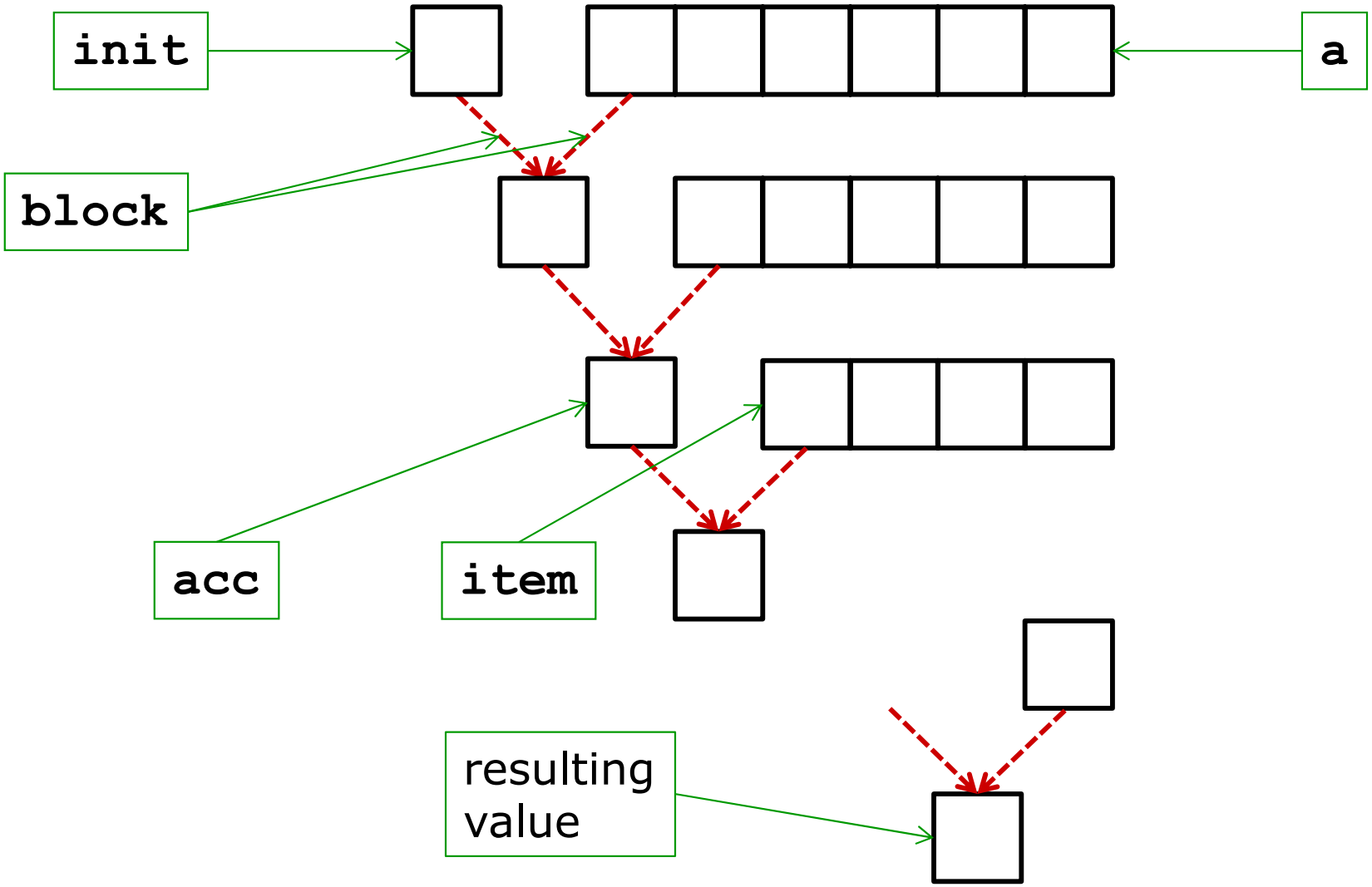
- Applies a binary operator between all the elements of the array
  - E.g., to sum the elements of an array
  - [15, 10, 8] → 0 + 15 + 10 + 8 → 33

  reduce(*callback[, initialValue]*)

  ***callback*(*previous*, *elt*, index, *array*)**
- Examples

```
let sum = (a, b) => a + b;
let acc = (a, b) => a + 2 * b;
[2, 3, 7, 1].reduce(sum)      //=> ?
[2, 3, 7, 1].reduce(sum, 0)   //=> ?
[2, 3, "7", 1].reduce(sum)    //=> ?
[2, 3, 7, 1].reduce(acc)      //=> ?
[2, 3, 7, 1].reduce(acc, 0)   //=> ?
```

Recall: Ruby's Reduction Chain



Iteration: Reduce

- Examples with anonymous functions

```
[2, 3].reduce((a, b) => a + b);
        //=> ?


[
  [0, 1],
  [2, 3],
  [4, 5]
].reduce((a, b) => a.concat(b));
        //=> ?
```

Your Turn

Given: roster of students (an array)

Write a JavaScript function that returns an html list of students (name and midterm score) whose gpa is > 3.0, such that the list is sorted by midterm score

1. Xi Chen (85)
2. Mary Smith (80)
3. Alessandro Reis (74)

Example Input


```
let roster =
[    { name: "Mary Smith",
       gpa: 3.7,
       midterm: 80 },
     { name: "Xi Chen",
       gpa: 3.5,
       midterm: 85 },
     { name: "Alessandro Reis",
       gpa: 3.2,
       midterm: 74 },
     { name: "Erin Senda",
       gpa: 3.0,
       midterm: 68 }          ];
```

Summary

- Array accessors and mutators
  - Accessors: indexOf, slice
  - Mutators for extraction: push/pop, unshift/shift, splice
  - Mutators for rearranging: reverse, sort
- Array iteration
  - Quantification: every, some, filter
  - Map (foreach for side-effects & mutating)
  - Reduce