Introduction to Programming - Concepts and Tools

Lecture 6

Arrays and Searching

Overview

- What is an array?
- Declaration, creation, initialization, and indexation
- Arrays as parameters
- Copying arrays (side effects)
- Multi-dimensional arrays
- Arrays of non-primitive types
- Linear and binary search
- Invariants
- Complexity and efficiency

Why Arrays?

Consider the example:

```
int a0 = 0, a1 = 1, a2 = 2, a3 = 3, a4 = 4;

int sum = 0;
sum = sum + a0;
sum = sum + a1;
sum = sum + a2;
sum = sum + a3;
sum = sum + a4;

System.out.println("The sum is: " + sum);
```

But what if to add say 1000 variables?

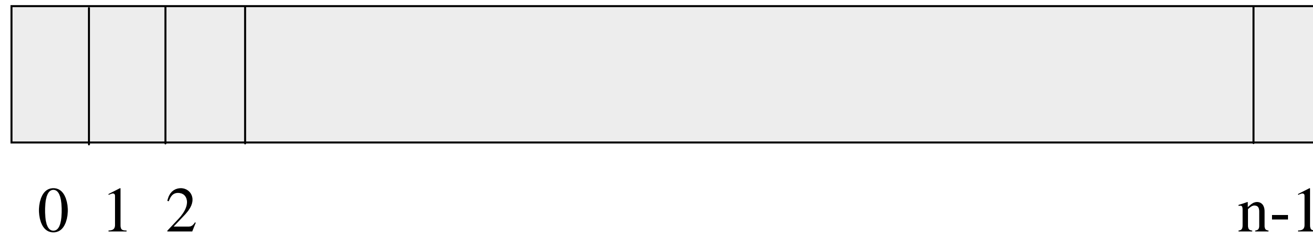
Why Arrays?

Use an *array* of length 1000!

```
int[] a = new int[1000];

for (int i = 0; i < 1000; i++)
   a[i] = i;

int sum = 0;
for (int i = 0; i < 1000; i++)
   sum += a[i];

System.out.println("The sum is: " + sum);
```

What is an array?

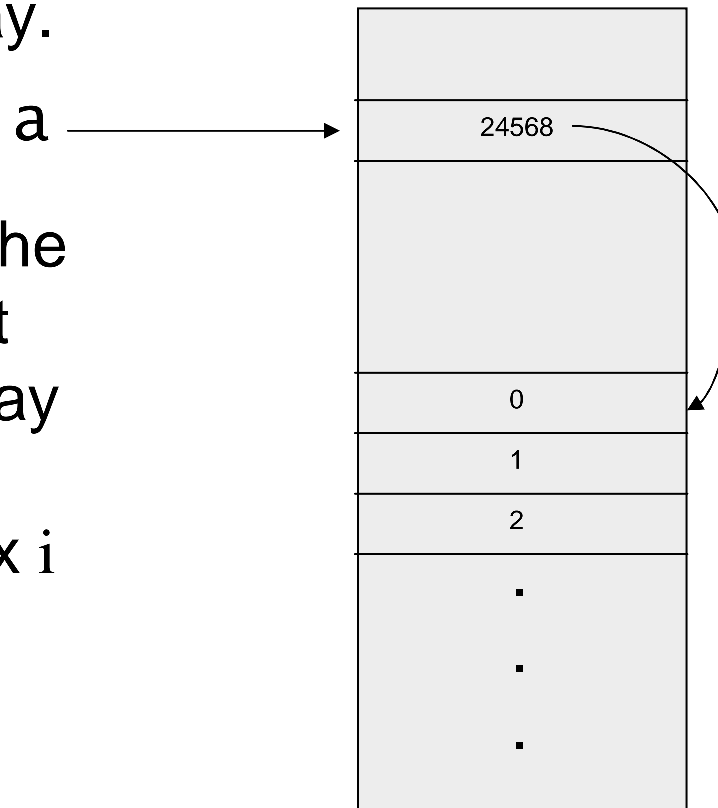
An **array** is a *container* that holds a group of values of the same (primitive or reference) type.



The elements in an array are *ordered*, the position of an element is its *index*. An array of size n is indexed from 0 to n-1.

Representing arrays in memory

An array variable (a reference variable) refers to the first element of the array.



The value of a is the location of the first element of the array named a. The element with index i is located i steps from the first element.

Array Declarations

An array variable is *declared* like any other variable by a type and a name, e.g.

```
int[] arrInt; //reference to array of
  ints
String[] arrStr; //reference to array of
  strings
```

`int[]` is the type "array of int's", and `String[]` is the type "array of strings".

`int[] arrInt;` allocates space in memory for a reference to an array.

Array Creation

An array variable is given an *initial value* (a reference to its first elements) by:

```
arrayVar = new Type[length];
```

For instance

```
arrInt = new int[i];
```

evaluates the *integer expression* `i`, creates an array of (allocates space in memory for) `i` elements of type `int`, and assign `arrInt` the value of the location of the first element.

Indexing an array

```
int[] a = new int[1000];
for (int i = 0; i < a.length; i++)
   a[i] = i;
```

- Elements of `a` may be accessed by *indexing* in the range `0` to `a.length - 1`. `a.length` is the number of elements in `a`.
- `IndexOutOfBoundException` if subscript outside `0` to `a.length - 1`.
- Each `a[i]` may be considered an `int` variable.

Array Creation and Initialization

Array elements are automatically initialized: primitive nummeric types to 0, booleans to `false`, and all other types are initialized to `null`.

```
int[] i = new int[2];          // i[0]=i[1]=0
boolean[] b = new boolean[2]; // b[0]=b[1]=false
String[] s = new String[2];   // s[0]=s[1]=null
```

Explicit initialization by comma separated list:

```
int[] i = { 11, 12, 13 }; // i[0]=11, i[1]=12, i[2]=13
String[] s = { ”one”, ”two”, ”three” };
```

Array Parameters

```
int max(int[] a) {
   int max = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i] > max) max = a[i];
   return max;
}
```

Parameters are passed by value. Because an array variable is a ***reference variable*** not the hole array of values but only the reference to the (first element) of the array is passed.

The advantage is that only a single reference and not all array values need to be copied.

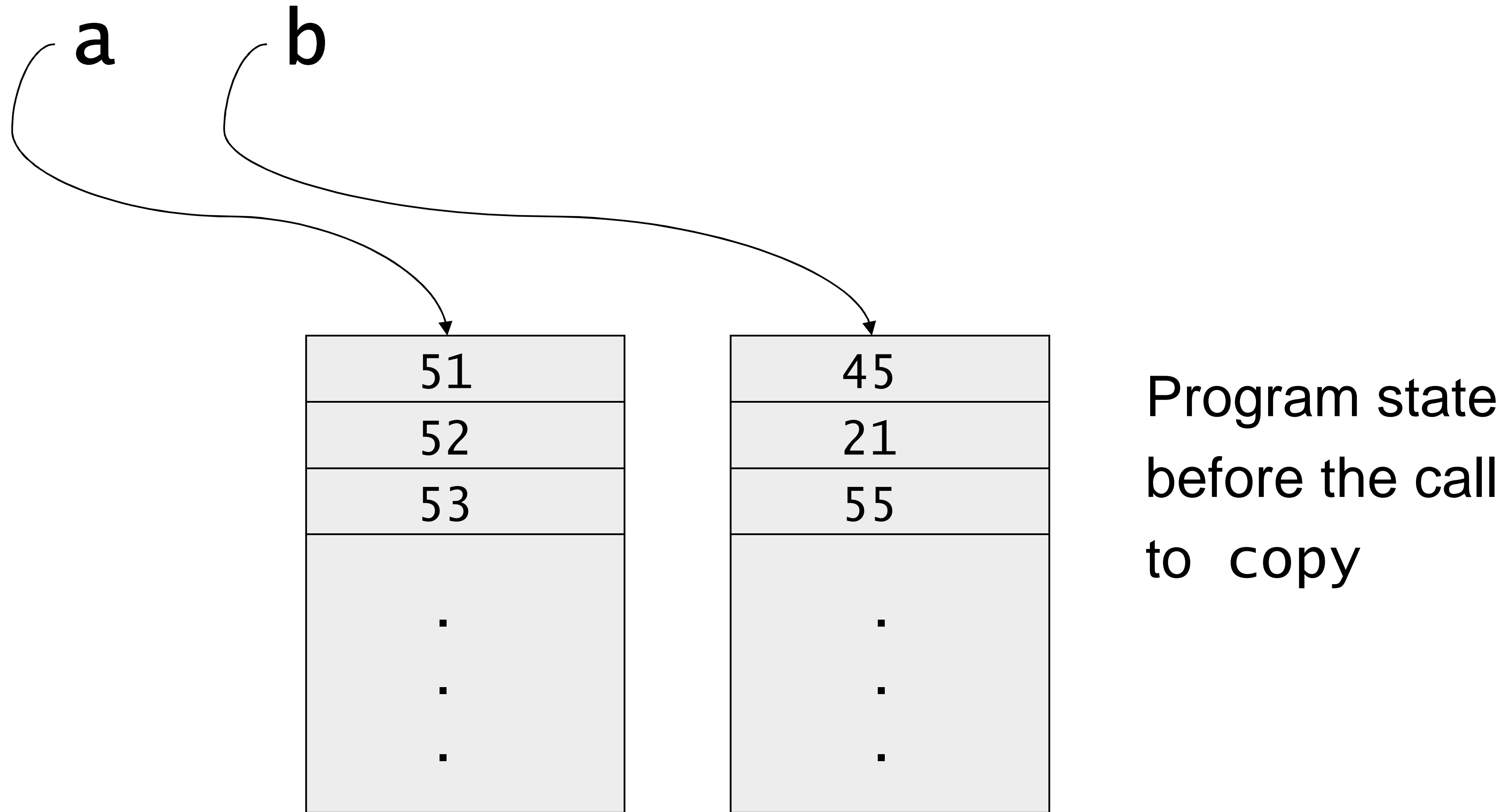
Copying Arrays

Suppose we want to copy the values from one array to another of equal length.

```
void copy(int[] from, int[] to) { to = from; }

int[] a = { 51, 52, 53, 54, 55, 56, 57, 58};
int[] b = { 45, 21, 55, 67, 91, 19, 81, 78};
copy(a,b);
```

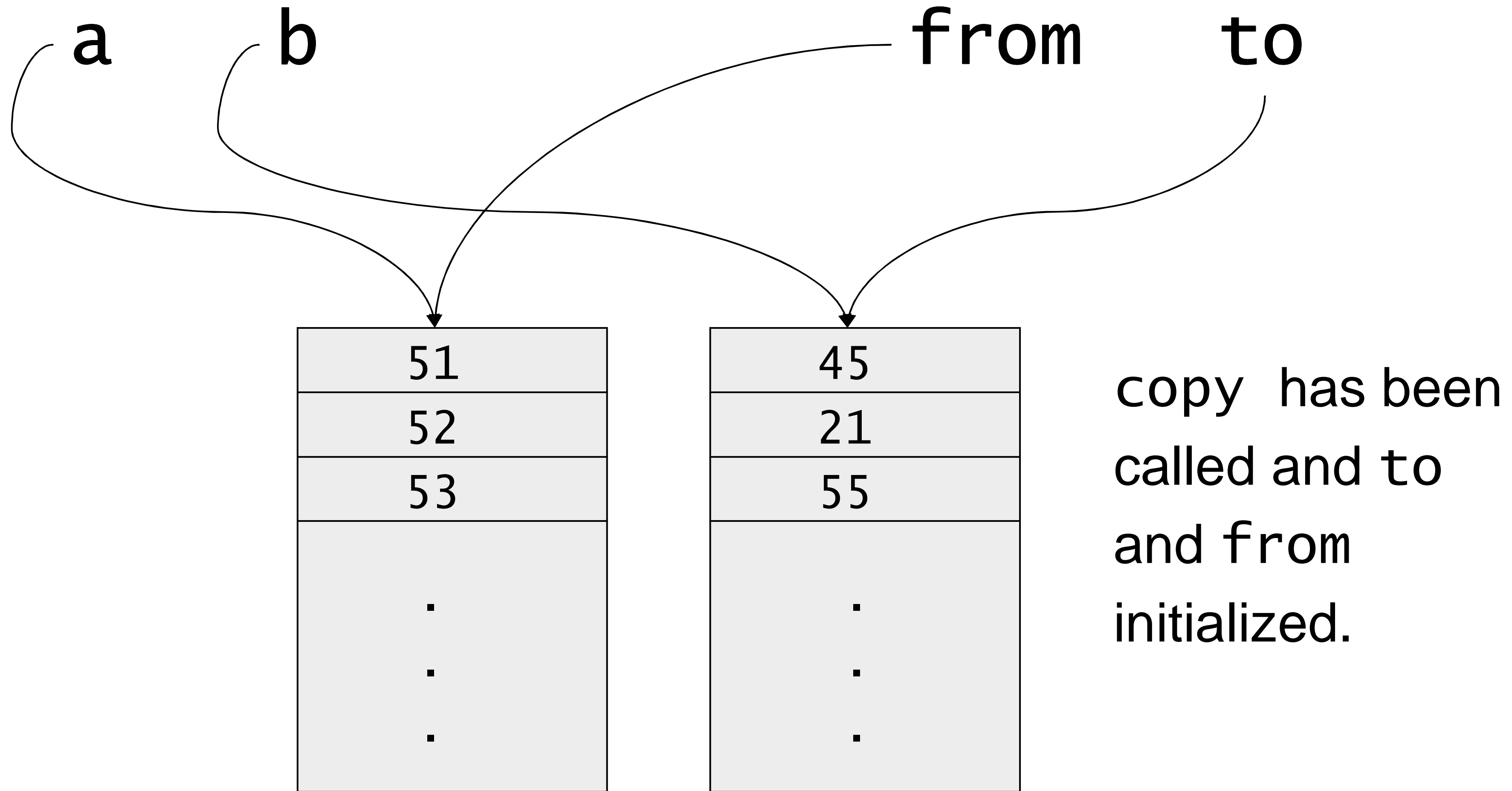
What is the effect of the call to `copy`?

Copying Arrays



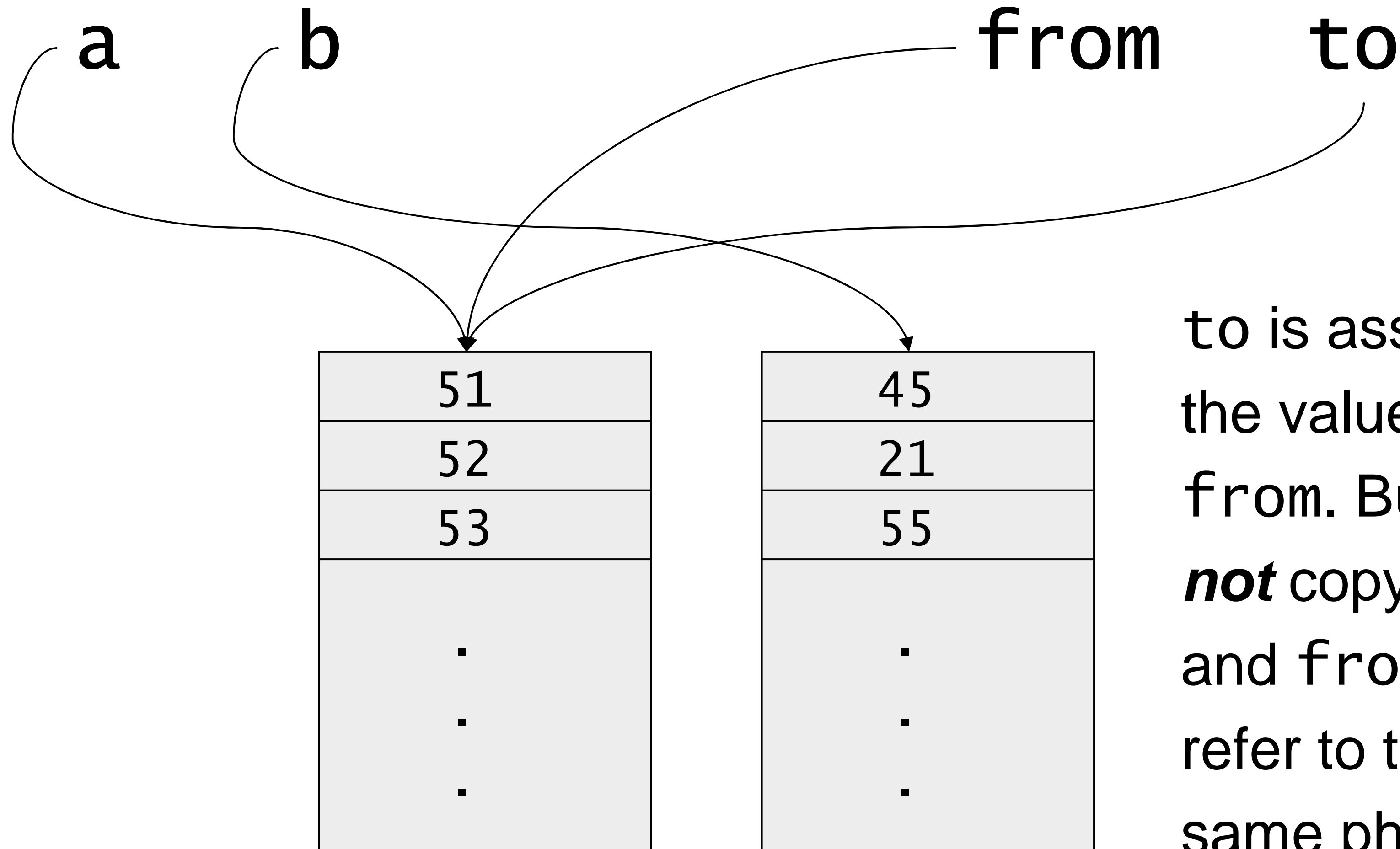
Program state before the call to `copy`

Copying Arrays



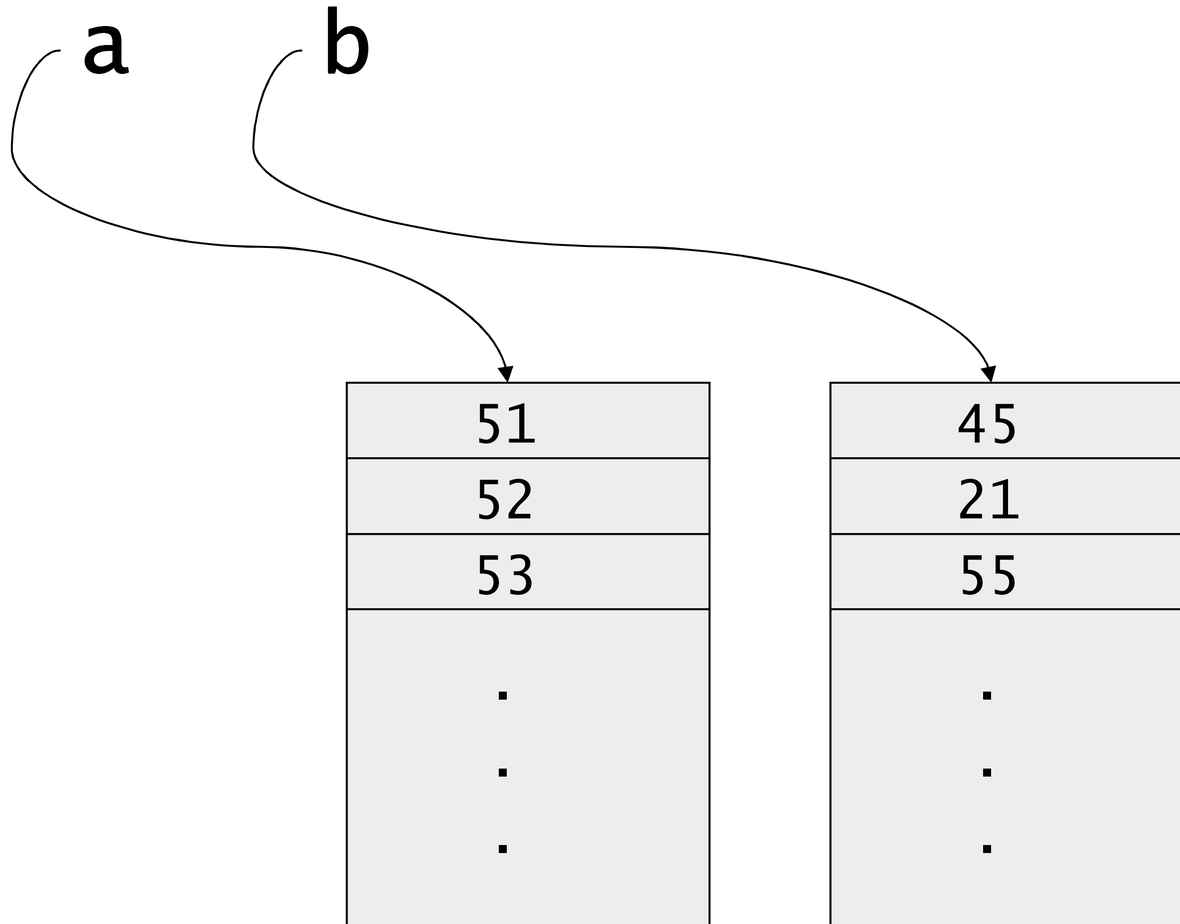
`copy` has been called and `to` and `from` initialized.

Copying Arrays



`to` is assigned the value of `from`. But this is ***not*** copying, `to` and `from` just refer to the same physical array!

Copying Arrays



When returning from the call to copy there is no effect on the program state because the parameters are local variables in copy.

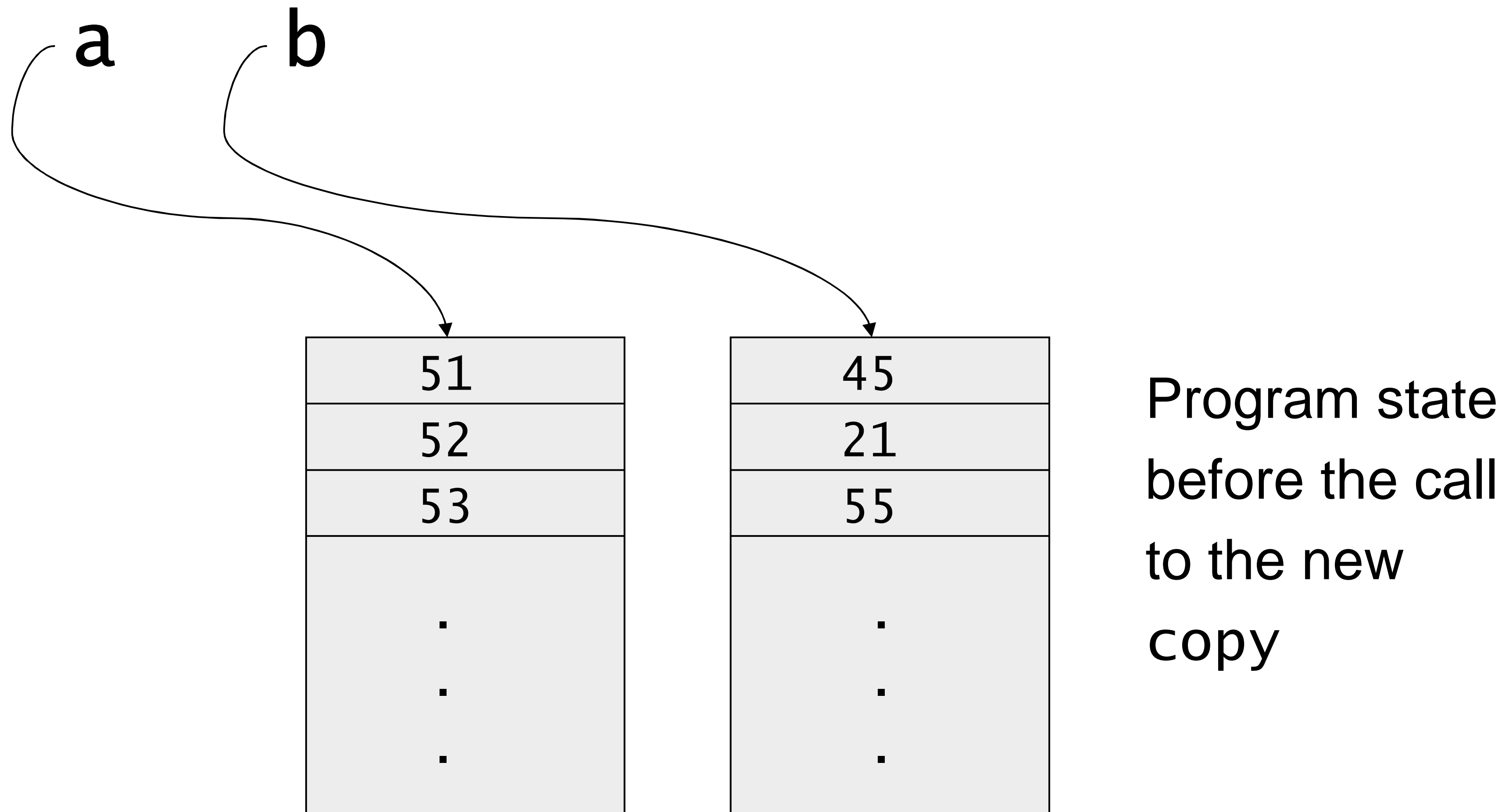
Copying by side effects

Ensure that the contents pointed at by the reference `from` is element by element copied to the array refered to by `to`.

```
void copy(int[] from, int[] to) {
   for (int i = 0; i < from.length; i++)
      to[i] = from[i];
}

int[] a = { 51, 52, 53, 54, 55, 56, 57, 58};
int[] b = { 45, 21, 55, 67, 91, 19, 81, 78};
copy(a,b);
```

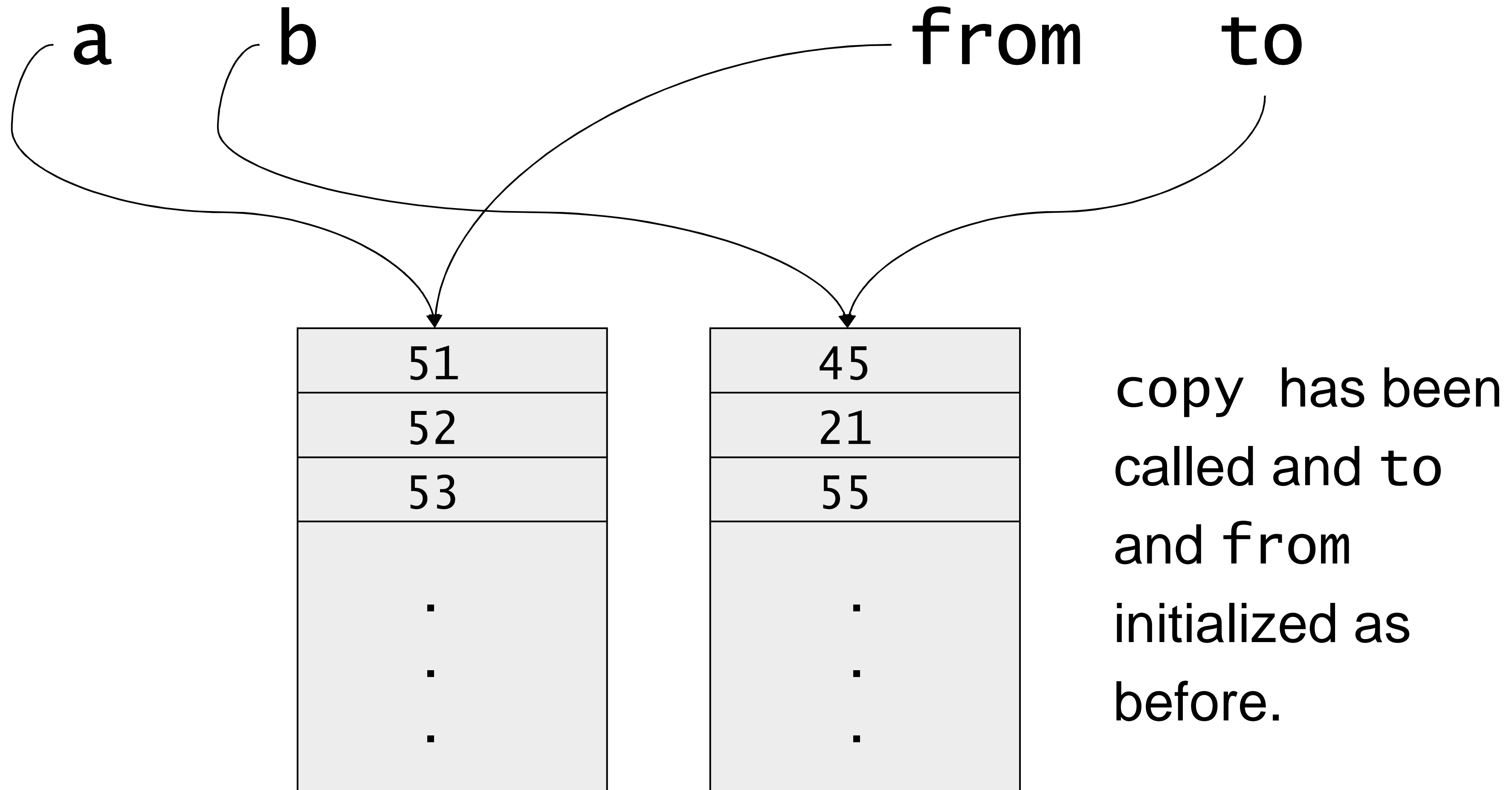
What is now the effect of the call to `copy`?

Copying by side effects



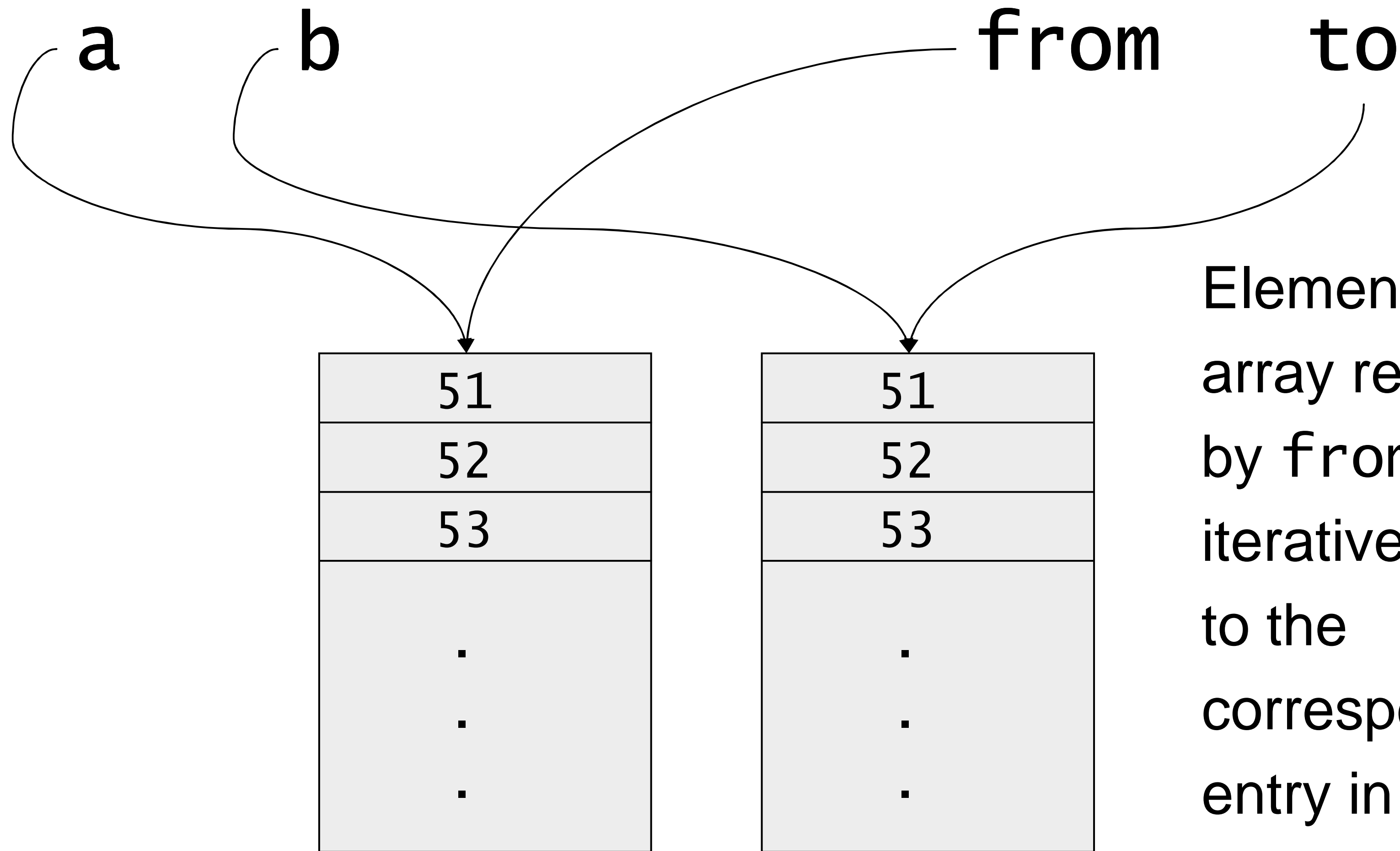
Program state before the call to the new copy

Copying by side effects



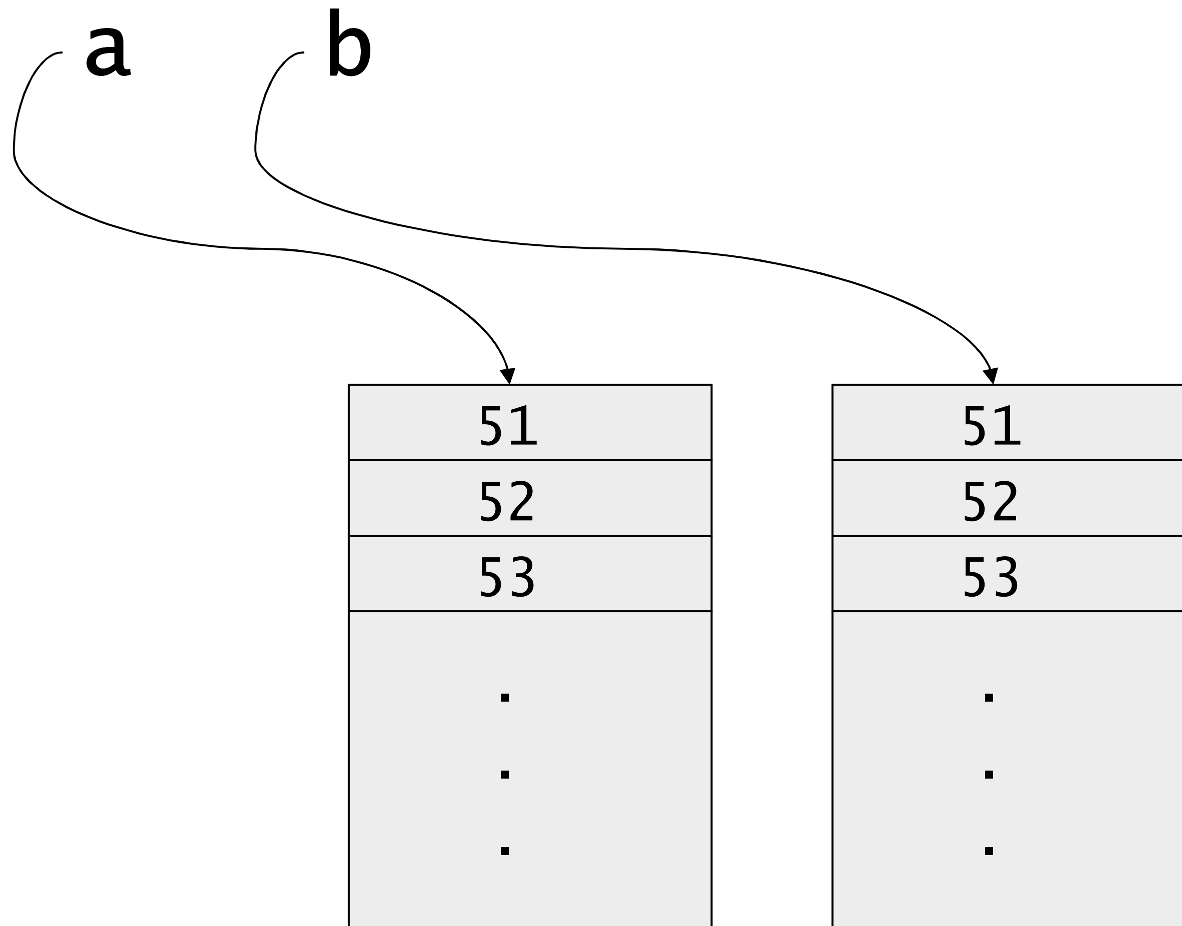
`copy` has been called and `to` and `from` initialized as before.

Copying by side effects



Elements in the array refered to by `from` are iteratively copied to the corresponding entry in the array refered to by `to`

Copying by side effects



Returning from the call to copy there is a **side effect** on the program state because the contents of the array refered to by the **output parameter** b has changed.

Array size and doublication

Array dublication (by side effect):

```
int[] dublicate(int[] a) {
   int[] tmp = new int[a.length];
   for (int i = 0; i < a.length; i++) tmp[i] = a[i];
   return tmp; }

int[] a = {12, 34, 45, 21, 55, 67, 91, 19, 81, 78};
int[] b = dublicate(a);
```

The *size of an array is fixed*, but an array variable can be assigned to a new array

```
int[] dublicateIncrease(int[] a) {
   int[] tmp = new int[2*a.length];
   for (int i = 0; i < a.length; i++) tmp[i] = a[i];
   return tmp; }

a = dublicateIncrease(a);
```

Sieve of Eratosthenes

```
static void crossOut(boolean[] b, int from, int interval) {
      for (int i = from; i < b.length; i += interval)
        b[i] = false;
  }

public static void main(String[] args) {

      // create boolean array for integers 0,1,...,args[0]
      boolean[] a = new boolean[Integer.parseInt(args[0])+1];

      // all but 0 and 1 are potential primes
      for (int i = 2; i < a.length; i++)
        a[i] = true;

      // sieve from 2 and onwards
      for (int j = 2; j <= Math.sqrt(Integer.parseInt(args[0])); j++)
        if (a[j]) crossOut(a,2*j,j);

      for (int k = 0; k < a.length; k++)
        if (a[k]) System.out.print(" " + k);

  }
```

Multi-Dimensional Arrays

Java allows arrays of any (primitive or reference) type, e.g. `int[]` is a reference type so `int[][]` is an array of arrays of int's

```
int[][] aa = { {0, 1, 1}, {1, 0, 1} };
int[][][] aaa = { { {0}, {1}, {1} },
                  { {1}, {0}, {1} },
                  { {1}, {1}, {0} } };
```

The indexing of array `aa` and the values it contains an be illustrated as follows:

```
aa[0][0] aa[0][1] aa[0][2]
aa[1][0] aa[1][1] aa[1][2]
```

```
0 1 1
1 0 1
```

Multi-Dimensional Arrays

`int[][][]` is an array of arrays of arrays of int's

```
int[][][] aaa = { { {0}, {1}, {2} },
                  { {1}, {0} },
                  { {2}, {1}, {0} } };
```

Notice `aaa` is *irregular*. The indexing of `aaa` and the values it contains an be illustrated as follows:

```
aa[0][0][0] aa[0][1][0] aa[0][2][0]
aa[1][0][0] aa[1][1][0]
aa[2][0][0] aa[2][1][0] aa[2][2][0]
```

```
0 1 2
1 0
2 1 0
```

Two-Dimensional Arrays

Consider the two dimensional array with 3 rows and 4 columns:

```
int[][] aa = new int[3][4];
```

We can fill it up with e.g.

```
for (int i = 0; i < aa.length; i++)
      for (int j = 0; j < aa[i].length; j++)
           aa[i][j] = i+j;
```

Arrays of Non-primitive Types

```
Class Card { … } // playing card

Class Deck { // deck of 52 playing card
     Card[] deck;

     Deck() { … }; // initializes all the playing cards

     void shuffle() {
       for (int i = 0; i < deck.length; i++)
       // select randomly a card k and swap with card i
              int k = (int)(Math.random()*52);
              Card tmp = new Card(deck[i]);
              deck[i] = deck[k];
              deck[k] = t;
     }
}
```

Linear Search

Suppose we want to **search** for an element in an *unordered array* A of length n.

```
A = { 3, 6, 89, 45, 11, 34, 90, 65,
      55, 44, 21, 7, 91, 64, 71, 12, 18,
      41, 87, 66, 100 };
```

Because A is unordered we must search from one end to the other **comparing** in the

- *best case* with only 1 element,
- *worst case* with all n elements, and
- *on avarage* with $n/2$ elements.

Linear Search

```
int linearSearch(int[] a, int x) {
      int i = 0;
      while (i < a.length)
           if (a[i] == x) return i;
           else i++;
      // if x is found its index in a is
      // returned, otherwise -1 is returned
      return -1;
 }
```

Binary Search

Suppose we want to **search** for an element in an *ordered array* `A` of length n.

```
A = { 3, 6, 7, 11, 12, 18, 21, 34, 41, 44,
    45, 55, 64, 65, 66, 71, 87, 89, 90, 91,
    100 };
```

Because `A` is ordered we may start search from the middle comparing in the

- *best case* with only 1 element, and
- *worst (average) case* with $\log2(n)$ (base 2 logarithm) elements.

Complexity

Binary search is much more **efficient** than linear search,

- linear seach has **linear** complexity, and
- binary search has **logaritmic** complexity

It takes on the average 500.000 comparisons to seach 1.000.000 elements by linear search, by binary seach it takes on average only 20 comparisons.

How to order an array?

```
int min(int[] a, int start) {
   // returns index (>= start) of the smallest element in a
   int min = start;
   for (int i = start+1; i < a.length; i++)
      if (a[i] < a[min]) min = i;
   return min;
}

void selectionSort(int[] a) {
   int tmp, indexToSwap;
   // pick iteratively the smallest element and place it at i
   for(int i = 0; i < a.length - 1; i++) {
      indexToSwap = min(a,i);
      tmp = a[i];
      a[i] = a[indexToSwap];
      a[indexToSwap] = tmp;
   }
}
```

Binary Search

```
boolean binarySearch(int[] a, int x) {
    int i = 0, j = 0, k = a.length-1;
    while (j <= k) {
       i = (j+k)/2;
       if (x == a[i]) return true;
       else if (x > a[i]) j = i+1;
           else k = i-1;
    }
    return false;
  }
```

How to be sure that the code is *correct*? If true is returned it should be that `x == a[i]` for some `i` in [0,…,a.length], and if false is returned then x is not in a[0,…,a.length].

Binary Search

```
boolean binarySearch(int[] a, int x) {
    int c = 0, l = 0, r = a.length-1;
    // Invariant:
    // 1) a[0],…,a[l-1] < x,
    //    x < a[r+1],…,a[a.length-1], and
    // 2) x == a[i] for some i in [j,…,k], or
    //    x not in a[l,…,r]
    while (l <= r) {
       c = (l+r)/2;
       if (x == a[c]) return true;
       else if (x > a[c]) l = c+1;
           else r = c-1;
    }
    return false;
  }
```

Correctness

Before entering the while loop the invariant holds (letting `n = a.length -1`) since:

`l = 0`   `r = n`

?

Part 1 of the invariant holds vacuously, and we don't know anyting about the area contents!

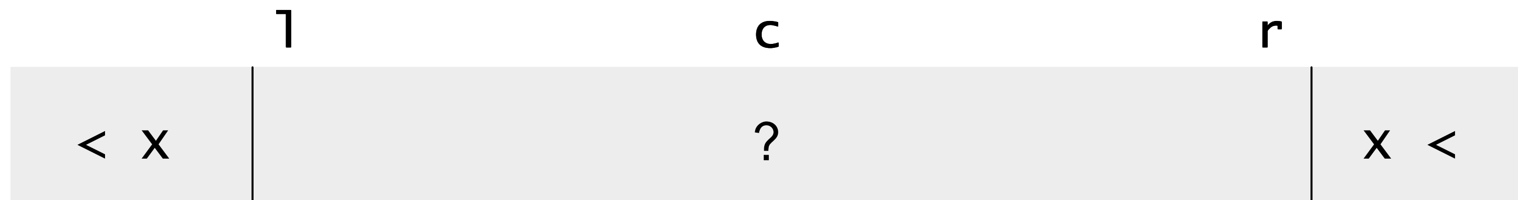
Correctness

Suppose the invariant holds after a number of iterations through the while loop. Then we have:

Before starting a next iteration, either

- `l > r`, in which case the loop terminats and false is returned. `x` is not in `a` `(l >= r+1` so `a[l-1] < x < a[l]`) so the invariant holds, or
- `l <= r`, in which case `c = (l+r)/2` is computed

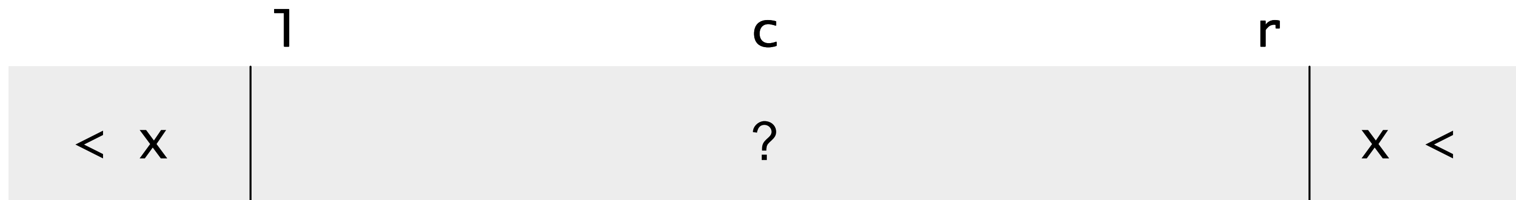
Correctness



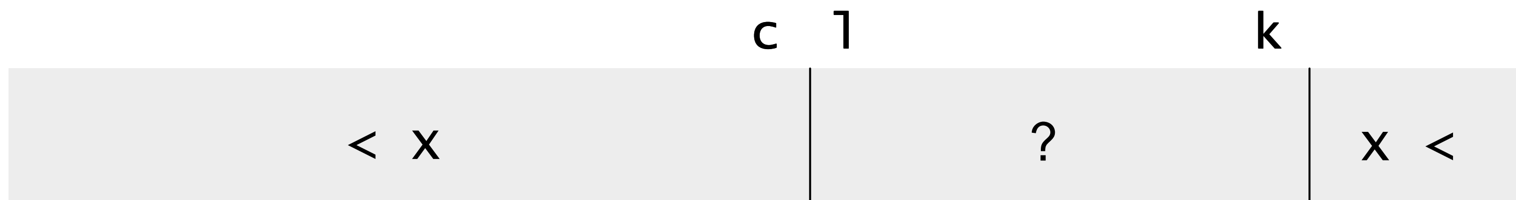
We consider three cases:

1) x == a[c], in which case true is returned, the invariant holds since x belongs to a.
2) `x > a[c]`, we split the array and proceed
3) `x < a[c]`, we split the array and proceed

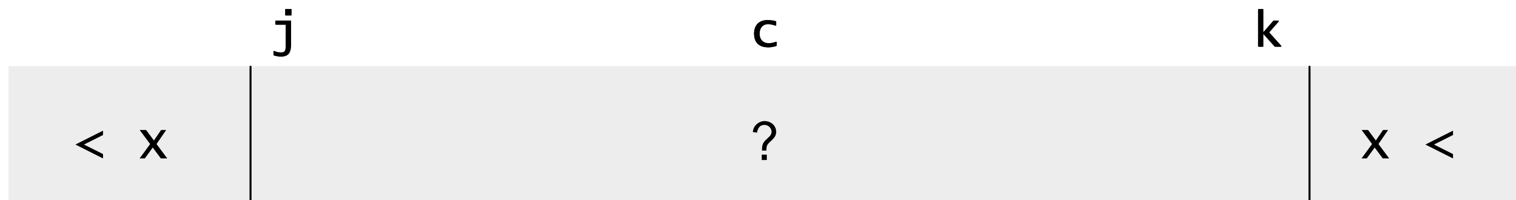
`x > a[c]`



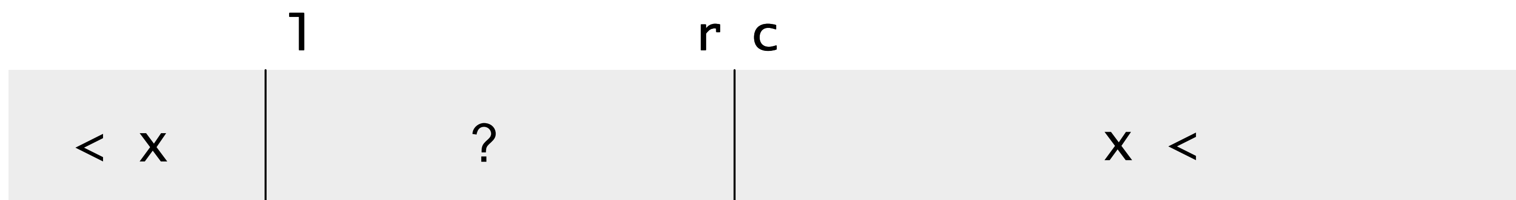
If `x > a[c]`, then also all elements to the left of `c` are less than `x`, so we set `l = c + 1` and the invariant holds again.



`x < a[c]`



If `x < a[c]`, then all elements to the right of `c` are greater than `x`, so we set `r = c - 1` and the invariant holds again.



Efficiency

```
int max(int[] a) {
   int max = a[0];
   for (int i = 1; i < a.length; i++)
        if (a[i] > max) max = a[i];
   return max;
}

int min(int[] a) {
   int min = a[0];
   for (int i = 1; i < a.length; i++)
        if (a[i] < min) min = a[i];
   return min;
}
```

Both `max` and `min` uses $n-1$ comparisons where n is `a.length`, hence the total **cost** of

```
System.out.println("Max is " + max(a) +
                   ", and min is " + min(a));
```

is $2(n-1)$ comparisons. We can do better using $3n/2-1$ comparisons!

Efficiency

```
int[] minMaxArray(int[] a) {
    int[] minMax = new int[2];    //store min and max
    int midPoint = a.length/2;

    // loop puts the min somewhere in the first half
    // and the max somewhere in the second half,
    for (int i = 0; i < midPoint; i++)
      if (a[i] > a[a.length - (i+1)])
        swap(i, a.length - (i+1), a);
    // loop finds the min which must be in first half
    minMax[0] = a[0];
    for (int i = 1; i <= midPoint; i++)
      if (a[i] < minMax[0])
        minMax[0] = a[i];
    // loop finds the max which must be in second half
    minMax[1] = a[midPoint];
    for (int i = midPoint+1; i < a.length; i++)
      if (a[i] > minMax[1])
        minMax[1] = a[i];
    return minMax;
  }
```

The End