CSE 143
Lecture 3

Implementing `ArrayIntList`

reading: 15.1 - 15.3

slides created by Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/

Exercise

- Pretend for a moment that there is **no `ArrayList` class**.
  - Write a program that reads a file `data.txt` (of unknown size) full of integers and prints them in reverse order.

```
17
932085
-32053278
100
3
```

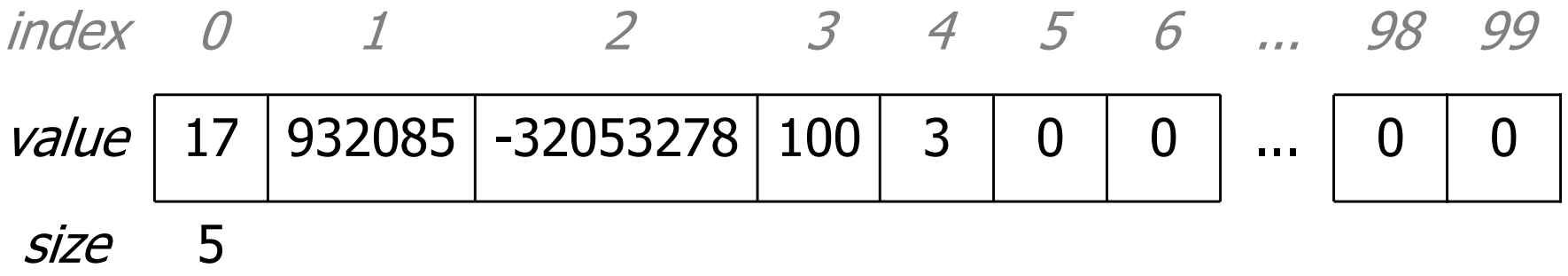
  - Output:

```
3
100
-32053278
932085
17
```

"Unfilled array" solution

- We are using an array to store a *list* of values.
  - Only the values at indexes [0, *size* - 1] are relevant.

```
int[] nums = new int[100];              // make a big array
int size = 0;
Scanner input = new Scanner(new File("data.txt"));
while (input.hasNextInt()) {
    nums[size] = input.nextInt();       // read each number
    size++;                             // into the array
}
for (int i = size - 1; i >= 0; i--) {
    System.out.println(nums[i]);        // print reversed
}
```



Possible list operations

```
public static void add(int[] list, int size, int value, int index)
public static void remove(int[] list, int size, int index)
public static void find(int[] list, int size, int value)
public static void print(int[] list, int size)
...
```

- We could write methods that accept a *list* array and its *size* .
  - But since this data and this behavior are so closely related, it makes more sense to put them together into an object.
  - A list object can store an array of elements and a size.
    - It can also have methods for manipulating the list of elements.
    - This will give us **abstraction** (hide the details of how the list works)

Exercise

- Let's write a class that implements a list using an `int[]`
  - We'll call it `ArrayIntList`
  - its behavior:
    - `add(`**value**`),`
    - `get(`**index**`),`
    - `size(),`
    - `remove(`**index**`),`
    - `indexOf(`**value**`),`
    - `toString(),`

      ...

    `add(`**index**`,` **value**`),`
    `set(`**index**`,` **value**`),`
    `isEmpty(),`
    `contains(`**value**`),`
  - The list's *size* will be the number of elements added to it so far.
    - The actual array length ("capacity") in the object may be larger. We'll start with an array of **length 10** by default.

Implementing add

- How do we add to the end of a list?

```
public void add(int value) {   // just put the element
    list[size] = value;        // in the last slot,
    size++;                    // and increase the size
}
```

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	0	0	0	0
size	6									

- `list.add(42);`

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	**42**	0	0	0
size	**7**									

Implementing add #2

- How do we add to the middle or end of the list?
  - must *shift* elements to make room for the value *(see book 7.3)*

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	12	0	0	0	0
size	6									

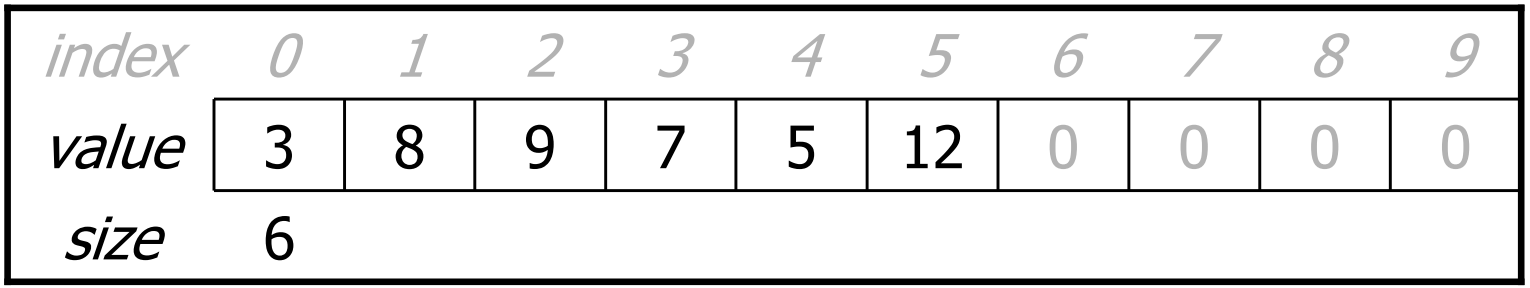
- `list.add(3, 42);` `// insert 42 at index 3`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	**42**	7	5	12	0	0	0
size	**7**									

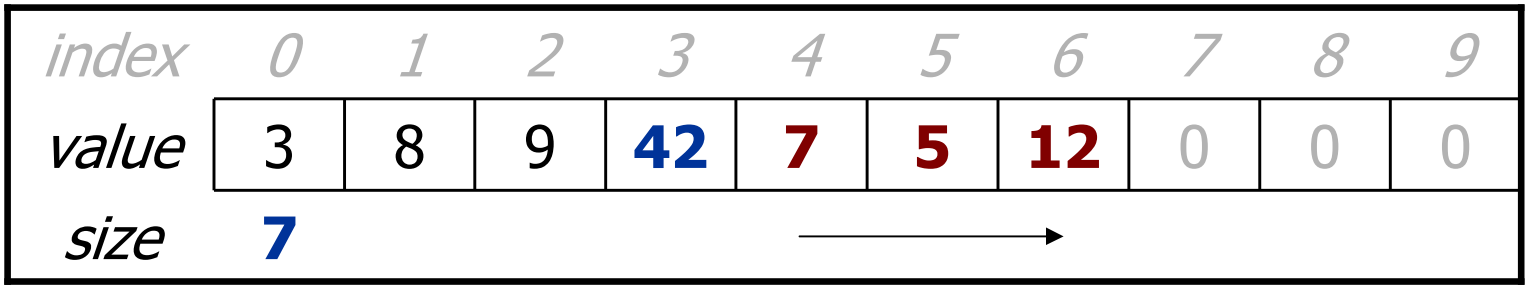
  - Note: The order in which you traverse the array matters!

add #2 code

```
public void add(int index, int value) {
    for (int i = size; i > index; i--) {
        list[i] = list[i - 1];
    }
    list[index] = value;
}
```



- `list.add(3, 42);`



Other methods

- Let's implement the following methods in our list:
  - `get(`**index**`)`
    Returns the element value at a given index.
  - `set(`**index**`, `**value**`)`
    Sets the list to store the given value at the given index.
  - `size()`
    Returns the number of elements in the list.
  - `isEmpty()`
    Returns `true` if the list contains no elements; else `false`.

Implementing `remove`

- How can we remove an element from the list?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	12	0	0	0	0
size	6									

- `list.remove(2);` `// delete 9 from index 2`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	7	5	12	0	0	0	0	0
size	**5**									

Implementing `remove`, cont.

- Again, we need to shift elements in the array
  - this time, it's a left-shift
  - in what order should we process the elements?
  - what indexes should we process?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	12	0	0	0	0
size	6									

- `list.remove(2);  // delete 9 from index 2`

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	**7**	**5**	**12**	0	0	0	0	0
size	**5**									

Implementing `remove` code

```
public void remove(int index) {
    for (int i = index; i < size; i++) {
        list[i] = list[i + 1];
    }
    size--;
    list[size] = 0;      // optional (why?)
}
```

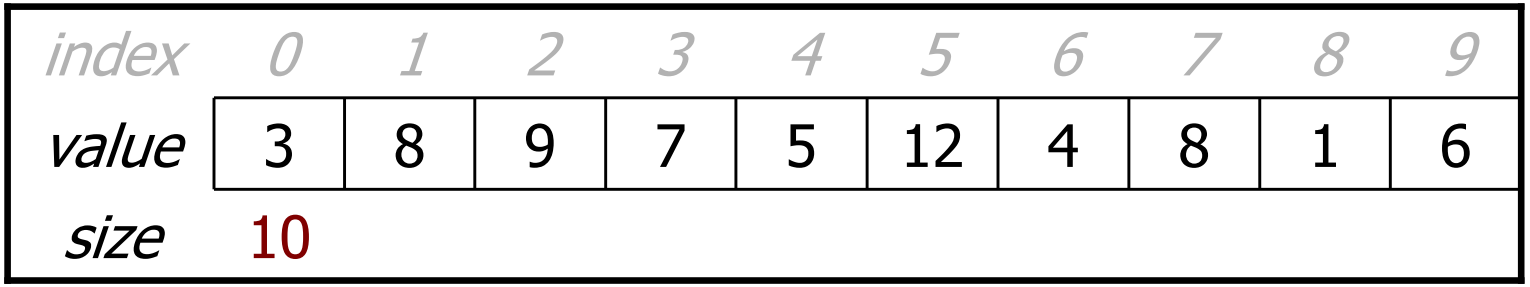
index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	0	0	0	0
size	6									

- `list.remove(2);` `// delete 9 from index 2`

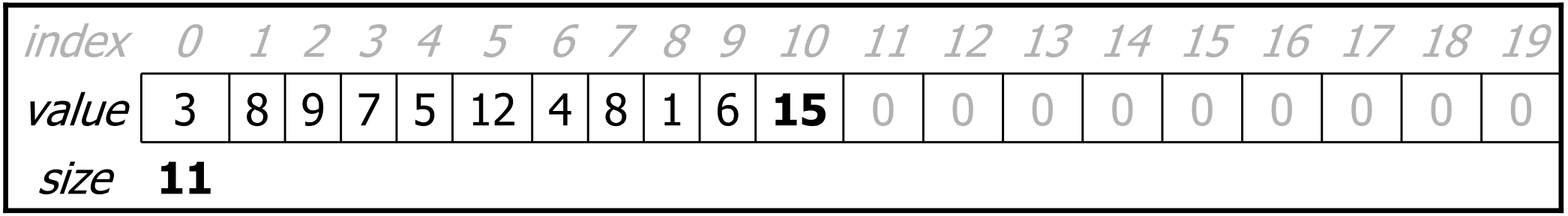
index	0	1	2	3	4	5	6	7	8	9
value	3	8	**7**	**5**	**12**	0	0	0	0	0
size	**5**			←						

Running out of space

- What should we do if the client adds more than 10 elements?



- `list.add(15);` `// add an 11th element`



The `Arrays` class

- Class `Arrays` in `java.util` has many useful array methods:

Method name	Description
`binarySearch(`**array**`, `**value**`)`	returns the index of the given value in a *sorted* array (or < 0 if not found)
`binarySearch(`**array**`, `**minIndex**`, `**maxIndex**`, `**value**`)`	returns index of given value in a *sorted* array between indexes *min* / *max* - 1 (< 0 if not found)
`copyOf(`**array**`, `**length**`)`	returns a new resized copy of an array
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element to the given value
`sort(`**array**`)`	arranges the elements into sorted order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, -25, 17]"`

- Syntax: `Arrays.`**methodName**`(`**parameters**`)`

Printing an `ArrayIntList`

- Let's add a method that allows clients to print a list's elements.
  - You may be tempted to write a `print` method:

```
// client code
ArrayIntList list = new ArrayIntList();
...
list.print();
```

    - Why is this a bad idea? What would be better?

The `toString` method

- Tells Java how to convert an object into a `String`

```
ArrayIntList list = new ArrayIntList();
System.out.println("list is " + list);
                // ("list is " + list.toString());
```

- Syntax:

```
public String toString() {
    code that returns a suitable String;
}
```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:

```
ArrayIntList@9e8c34
```

`toString` solution

```
// Returns a String representation of the list.
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "]";
        return result;
    }
}
```

Searching methods

- Implement the following methods:
  - `indexOf` - returns the first index an element is found, or -1 if not
  - `contains` - returns true if the list contains the given int value

- Why do we need `isEmpty` and `contains` when we already have `indexOf` and `size` ?
  - Adds convenience to the client of our class:

```
// less elegant
if (myList.size() == 0) {
if (myList.indexOf(42) >= 0) {
```

```
// more elegant
if (myList.isEmpty()) {
if (myList.contains(42)) {
```

Multiple constructors

- Our list class has the following constructor:

```
public ArrayIntList() {
    elementData = new int[10];
    size = 0;
}
```

- Let's add a new constructor that takes a capacity parameter:

```
public ArrayIntList(int capacity) {
    elementData = new int[capacity];
    size = 0;
}
```

  - The constructors are very similar. Can we avoid redundancy?

`this` keyword

- **`this`** : A reference to the *implicit parameter*
  (the object on which a method/constructor is called)

- Syntax:
  - To refer to a field: `this.`**field**
  - To call a method: `this.`**method**`(`**parameters**`);`
  - To call a constructor from another constructor: `this(`**parameters**`);`

Revised constructors

```
// Constructs a list with the given capacity.
public ArrayIntList(int capacity) {
    elementData = new int[capacity];
    size = 0;
}


// Constructs a list with a default capacity of 10.
public ArrayIntList() {
    this(10);   // calls (int) constructor
}
```

Class constants

```
public static final type name = value;
```

- **class constant**: a global, unchangeable value in a class
  - used to store and give names to important values used in code
  - documents an important value; easier to find and change later

- classes will often store constants related to that type
  - `Math.PI`
  - `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
  - `Color.GREEN`

```
// default array length for new ArrayIntLists
public static final int DEFAULT_CAPACITY = 10;
```