CSE 143
Lecture 2

Collections and `ArrayIntList`

slides created by Marty Stepp

http://www.cs.washington.edu/143/

Collections

- **collection**: an object that stores data; a.k.a. a "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (find), get *size*
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `Stack`, `Queue`, `PriorityQueue`
  - *Why should we want to use collections?*

Exercise

- Write a program that reads a file (of unknown size) full of integers and prints the integers in the reverse order to how they occurred in the file. Consider example file `data.txt`:

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

  - When run with this file, your program's output would be:

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

Solution using arrays

```
int[] nums = new int[100];    // make a really 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
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*...*	*98*	*99*
value	17	932085	-32053278	100	3	0	0	...	0	0
size	5									

Unfilled arrays

```
int[] nums = new int[100];
int size = 0;
```

- We often need to store an unknown number of values.
  - Arrays can be used for this, but we must count the values.
  - Only the values at indexes [0, *size* - 1] are relevant.

- We are using an array to store a *list* of values.
  - What other operations might we want to run on lists of values?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*...*	*98*	*99*
value	17	932085	-32053278	100	3	0	0	...	0	0
size	**5**									

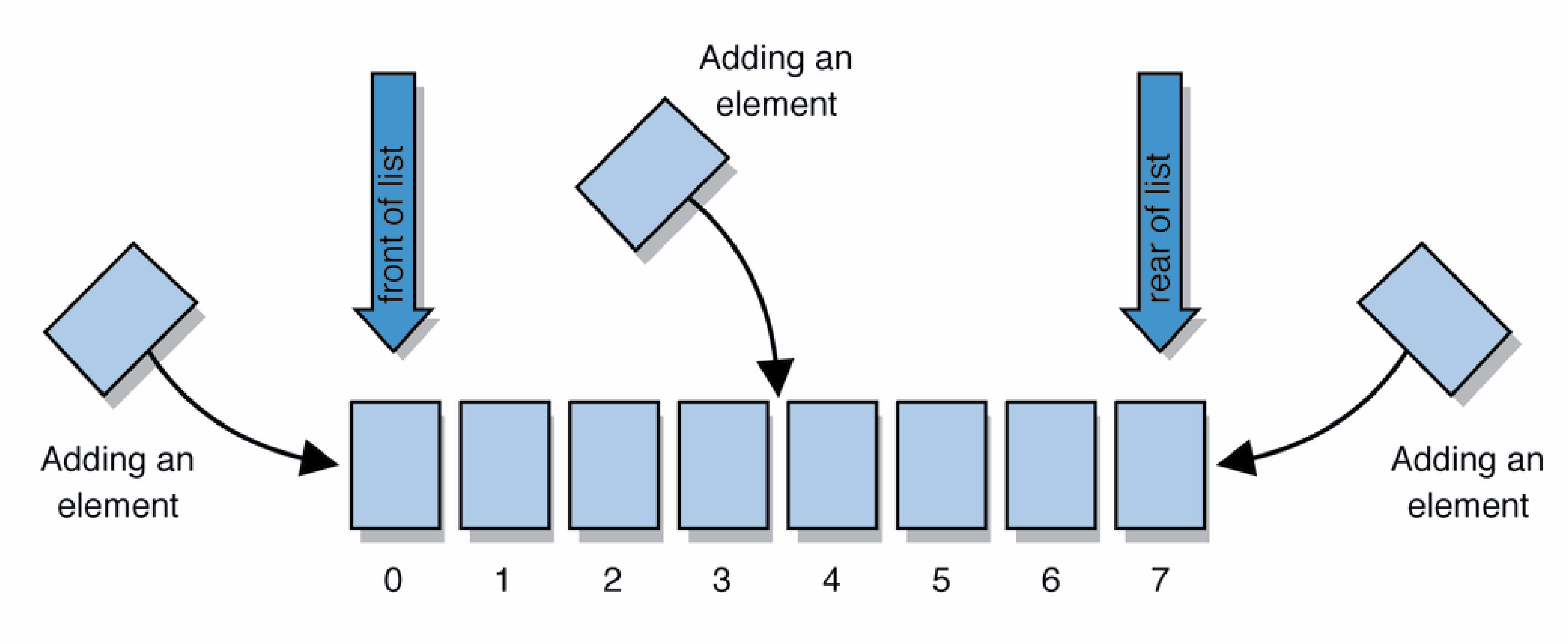
Other possible operations

```
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 implement these operations as methods that accept a *list* array and its *size* along with other parameters.
  - But since the behavior and data 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, and can have methods for manipulating the list of elements.
    - Promotes **abstraction** (hides details of how the list works)

Lists

- **list**: a collection storing an ordered sequence of elements, each accessible by a 0-based index
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere



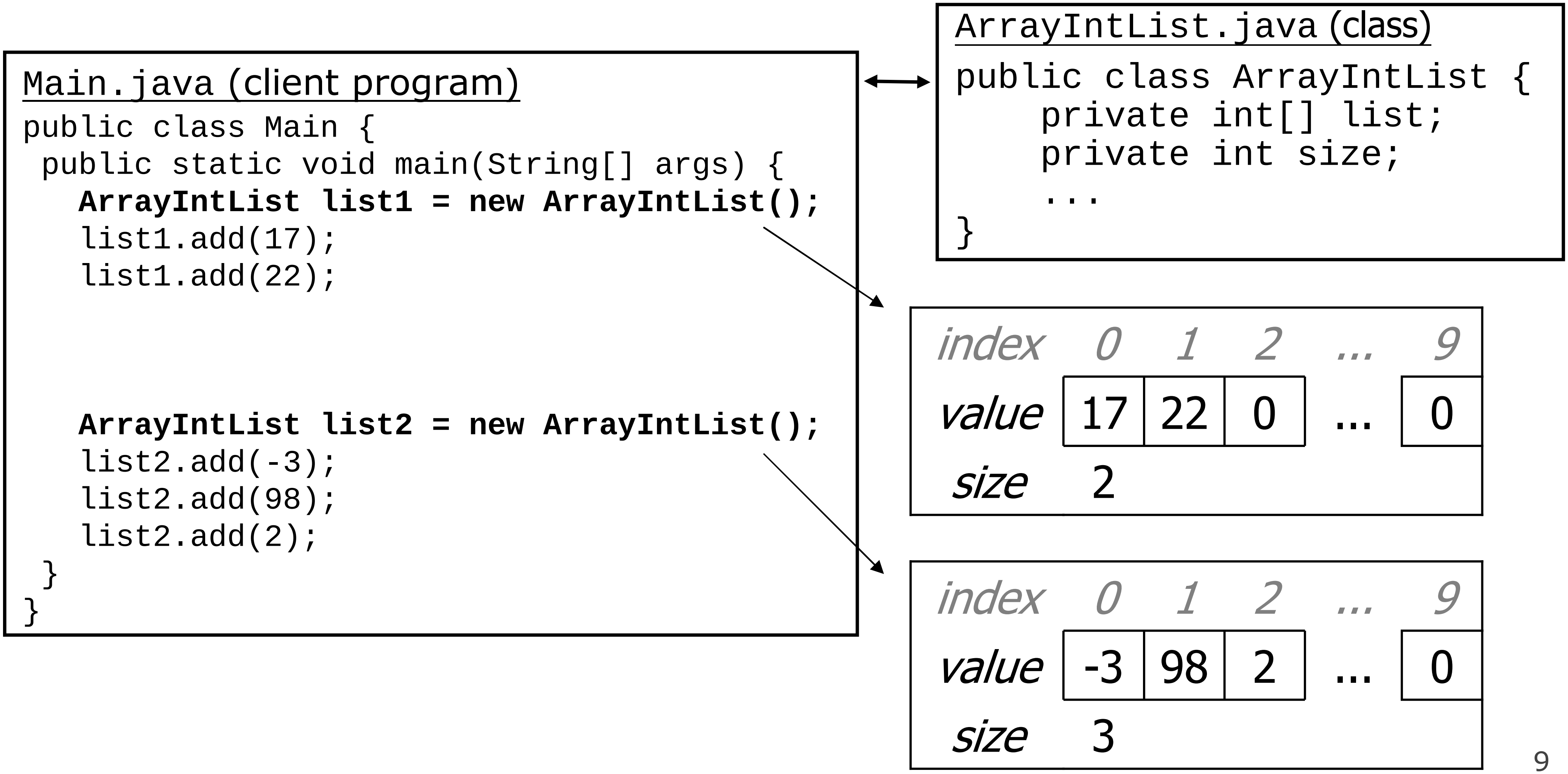
Exercise

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

  - The list's *size* will be the number of elements added to it so far
  - How will the list be used?...

Client programs

- `ArrayIntList.java` is not, by itself, a runnable program.
  - A class can be used by **client programs**.



Main.java (client program)

```
public class Main {
 public static void main(String[] args) {
   ArrayIntList list1 = new ArrayIntList();
   list1.add(17);
   list1.add(22);



   ArrayIntList list2 = new ArrayIntList();
   list2.add(-3);
   list2.add(98);
   list2.add(2);
 }
}
```

ArrayIntList.java (class)

```
public class ArrayIntList {
    private int[] list;
    private int size;
    ...
}
```

index	*0*	*1*	*2*	*...*	*9*
value	17	22	0	...	0
size	2				

index	*0*	*1*	*2*	*...*	*9*
value	-3	98	2	...	0
size	3				

Using ArrayIntList

- construction

```
int[] numbers = new int[5];
ArrayIntList list = new ArrayIntList();
```

- storing a value

```
numbers[0] = 42;
list.add(42);
```

retrieving a value

```
int n = numbers[0];
int n = list.get(0);
```

- searching for the value 27

```
for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] == 27) { ... }
}

if (list.indexOf(27) >= 0) { ... }
```

Pros/cons of `ArrayIntList`

- pro (benefits)
  - simple syntax
  - don't have to keep track of array size and capacity
  - has powerful methods (`indexOf`, `add`, `remove`, `toString`)

- con (drawbacks)
  - `ArrayIntList` only works for `ints` (arrays can be any type)
  - syntax is different to learn and use

Implementing add

- Add to end of list is easy; just store element and increase size

```
public void add(int index, int value) {
    list[size] = value;
    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**									

Printing a list

- You may be tempted to write a method that prints a list:

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

- But the better way is to make a `toString` method in the list:

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

// client code
System.out.println(list);    // calls toString
```

Implementing add (2)

- Adding to the middle or front is hard *(see book ch 7.3)*
  - must *shift* nearby elements to make room for the new value

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);`

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!

Implementing add (2)

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

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**);

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

Implementing remove

```
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);`

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

Preconditions

- What happens if the client tries to access an element that is past the size but within the bounds of the array?
  - Example: `list.get(11);` on a list of 5 elements, capacity 100
  - We have not addressed this case yet, and currently we just choose to assume that the user will not do such a thing.
- **precondition**: Something your method assumes is true at the start of its execution.
  - Often documented as a comment on the method's header:

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public void remove(int index) {
    return elementData[index];
}
```