CSE 143
Lecture 3

`ArrayIntList`

slides created by Ethan Apter and Marty Stepp
http://www.cs.washington.edu/143/

`remove`

- **`ArrayIntList`** has an **`add`**, so it should also have a **`remove`**
- **`remove`** will take an index as a parameter
- But how do we remove from **`ArrayIntList`**?
  - Is it enough to just set the value to 0 or -1?
- No! 0 and -1 can represent real, valid data
- Instead we need to:
  - shift all remaining valid data, so there is no "hole" in our data
  - decrement size, so there's one less piece of data

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**	12	0	0	0	0
size	**5**			←						

remove

- `remove` code:

Be careful with loop boundaries!

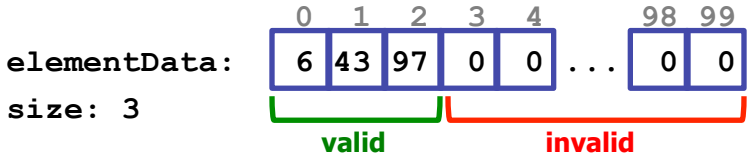
```
public void remove(int index) {
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    size--;
}
```

- We didn't "reset" any value to 0. Why not?

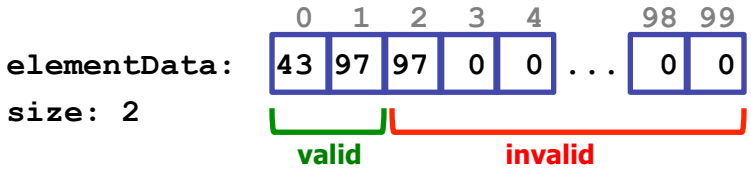
5

remove

- If we made an `ArrayIntList` and `add`ed the values 6, 43, and 97, it would have the following state:



- After a call of `remove(0)` it has this state:



- We don't care what values are in the invalid data

6

Is `ArrayIntList` Finished?

- What we've done so far:
  - Made an **`ArrayIntList`** class
  - Gave it enough variables to maintain its state
  - Gave it three methods: **`add`**, **`remove`**, and **`toString`**
- Sure we could add more methods...
- But what if our client is malicious?

```
ArrayIntList list = new ArrayIntList();
list.add(6);
list.size = -1
list.size = 9000;
list.elementData = null;
```

This can really mess up our `ArrayIntList`!

7

`private`

- **`private`** is a keyword in Java
- **`private`** is used just like **`public`**, but has the opposite effect
- When something is made **`private`**, it can be accessed only by the class in which it is declared
- Some things that can be **`private`**:
  - methods
  - fields

8

ArrayIntList

- Now we'll update **`ArrayIntList`** to use **`private`** fields:

```
public class ArrayIntList {
    private int[] elementData = new int[100];
    private int size = 0;
    ...
}
```

- Now the malicious code won't work!
  - If the client tries to access **`elementData`** or **`size`**, he'll get a compiler error

9

A Problem!

- What if the client wants to know the current size of **`ArrayIntList`**?
- This seems like a reasonable request...
- But we've completely blocked all access to **`size`**
- We don't mind telling the client the current size, we just don't what them to change it
- How can we solve this problem?

10

Accessor Methods

- We can write a method that returns the current size:

```
public int size() {
    return size;
}
```

- Because size is an int, this returns a copy of size
- Our size method is an accessor method
- **Accessor method**: a method that returns information about an object without modifying the object

11

get

- We should also provide a way for the client to read the values in `elementData`
- This should also be an accessor method. We'll call it `get`:
- `get` will return the value of `elementData` at a given index
- Code for `get`:

```
public int get(int index) {
    return elementData[index];
}
```

12

Preconditions

- What happens if someone passes an illegal index to **`get`**?
  - possible illegal indexes: -100, 9999
- Our code will break! This means **`get`** has a precondition
- **Precondition**: a condition that must be true before a method is called. If it is not true, the method may not work properly
- So, a precondition for **`get`** is that the index be valid
  - The index must be greater than or equal to zero
  - And the index must be less than **`size`**
- At the very least, we should record this precondition in a comment

13

Postconditions

- While we're writing a comment for **`get`**, we should also say what it action it performs
- **Postcondition**: a condition a method guarantees to be true when it finishes executing, as long as the method's preconditions were met
- What is get's postcondition?
  - it has returned the current value located at the given index

14

Pre/Post for `get`

- One way to record preconditions and postconditions is with a pre/post style comment:

```
// pre:  0 <= index < size()
// post: returns the value at the given index
public int get(int index) {
    return elementData[index];
}
```

15

Constructors

- Whenever you use the keyword **`new`**, Java calls a special method called the constructor
- Constructors have special syntax
  - they have the same name as the class
  - they do not have a return type
- Here's how to write a simple constructor for ArrayIntList:

```
public ArrayIntList() {
    // constructor code
    ...
}
```

16

Default Constructor

- But didn't we already use **`new`** on our **`ArrayIntList`**? How does that work when we hadn't yet written a constructor?
- If a class does not have any constructors, Java provides a default constructor
- The default constructor is often known as the zero-argument constructor, because it takes no parameters/arguments
- However, as soon as you define a single constructor, Java no longer provides the default constructor

17

`ArrayIntList` Constructor

- Here's the updated code for **`ArrayIntList`**, now with a constructor:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   public ArrayIntList() {
      elementData = new int[100];
      size = 0;
   }
   ...
}
```

- Notice that I moved the initialization of the fields into the constructor. This is considered better style in Java, and we will look for it when grading.

18

Automatic/Implicit Initialization

- What happens if the fields are never initialized?
- If you don't initialize your fields, Java will automatically initialize them to their zero-equivalents
- Some zero-equivalents, by type:
  - `int`: 0
  - `double`: 0.0
  - `boolean`: false
  - objects (like arrays or Strings): null
- This means we did not have to initialize size to 0 beforehand.

19

Multiple Constructors

- You can have more than one constructor
- Just like when overloading other methods, all constructors for the same class must have different parameters
- An `ArrayIntList` constructor that takes a capacity as a parameter:

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

20

this

- Now we have the following two constructors:

```
public ArrayIntList() {
   elementData = new int[100];
   size = 0;
}
```

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

- We can use the keyword **this** to fix our redundancy. Using **this** with parameters will call the constructor in the same class that requires those parameters.

- Updated constructor code:

```
public ArrayIntList() {
   this(100);
}
```

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

21

Constants

- Our default value of 100 for capacity is arbitrary

- We should make it a class constant instead

- Code to declare a class constant:

```
public static final int DEFAULT_CAPACITY = 100;
```

- Updated zero-argument constructor:

```
public ArrayIntList() {
   this(DEFAULT_CAPACITY);
}
```

22

Completed `ArrayIntList`

- Has two fields
  - **`elementData`** and **`size`**
- Has one constant
  - **`DEFAULT_CAPACITY`**
- Has two constructors
  - **`ArrayIntList()`** and **`ArrayIntList(int capacity)`**
- Has seven methods (some not covered in lecture)
  - **`size()`**, **`get(int index)`**, **`toString()`**, **`indexOf(int value)`**, **`add(int value)`**, **`add(int index, int value)`**, and **`remove(int index)`**

23

Quick Discussion: `static`

- **`static`** is hard to understand
  - Many of you will pass CSE 143 without understanding **`static`**
- When something is declared **`static`**, it is shared by all instances of a class
- What would happen if we made **`size`** a **`static`** field?
  - All instances of **`ArrayIntList`** would use and update the same **`size`** variable!
  - We do **not** want to do this...
  - But what would happen if we tried it?

24

Quick Discussion: `static`

- Making `size` a `static` field:

```
private static int size;
```

- Consider the following code

```
ArrayIntList list1 = new ArrayIntList();
ArrayIntList list2 = new ArrayIntList();
list1.add(6);
list1.add(9);
System.out.println("sizes: " + list1.size() + ", " + list2.size());
System.out.println("toStrings: " + list1 + ", " + list2);
```

- What is printed?

```
sizes: 2, 2
toStrings: [6, 9], [0, 0]
```

Making `size` `static` affects more than just `size()`!

25