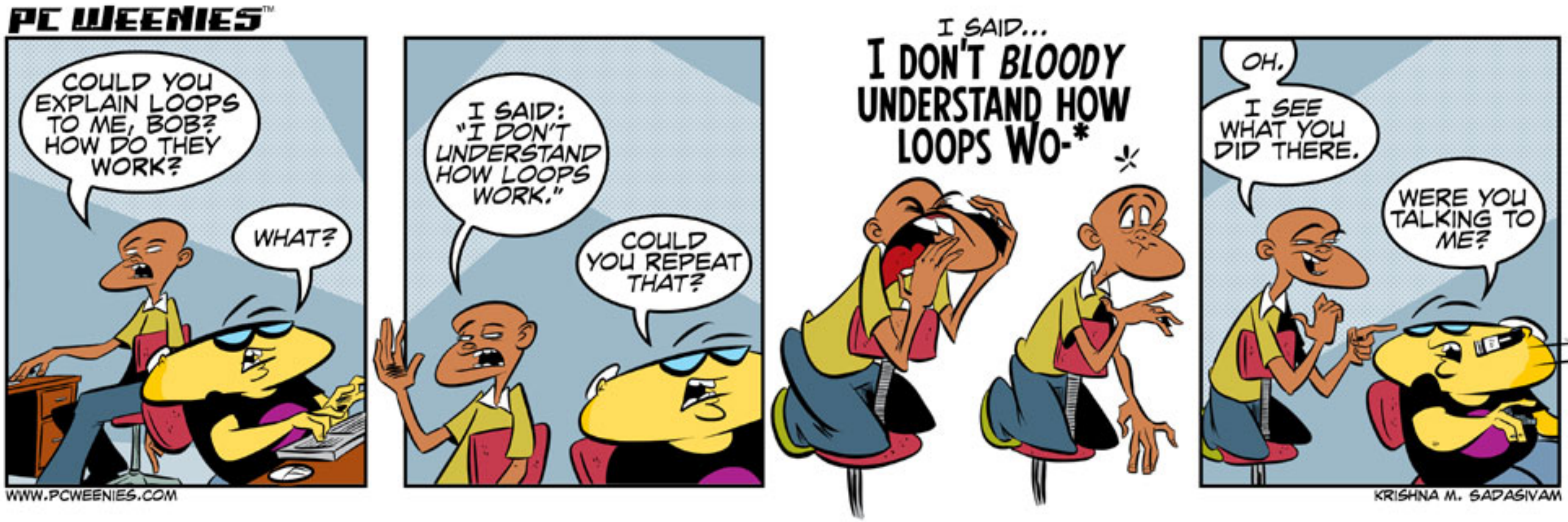
Building Java Programs

Chapter 10

Lecture 10-1: `ArrayList`

reading: 10.1


PC WEENIES™
COULD YOU EXPLAIN LOOPS TO ME, BOB? HOW DO THEY WORK?
WHAT?
I SAID: "I DON'T UNDERSTAND HOW LOOPS WORK."
COULD YOU REPEAT THAT?
I SAID... I DON'T BLOODY UNDERSTAND HOW LOOPS WO-*
OH.
I SEE WHAT YOU DID THERE.
WERE YOU TALKING TO ME?
WWW.PCWEENIES.COM
KRISHNA M. SADASIVAM

Words exercise

- Write code to read a file and display its words in reverse order.

- A solution that uses an array:

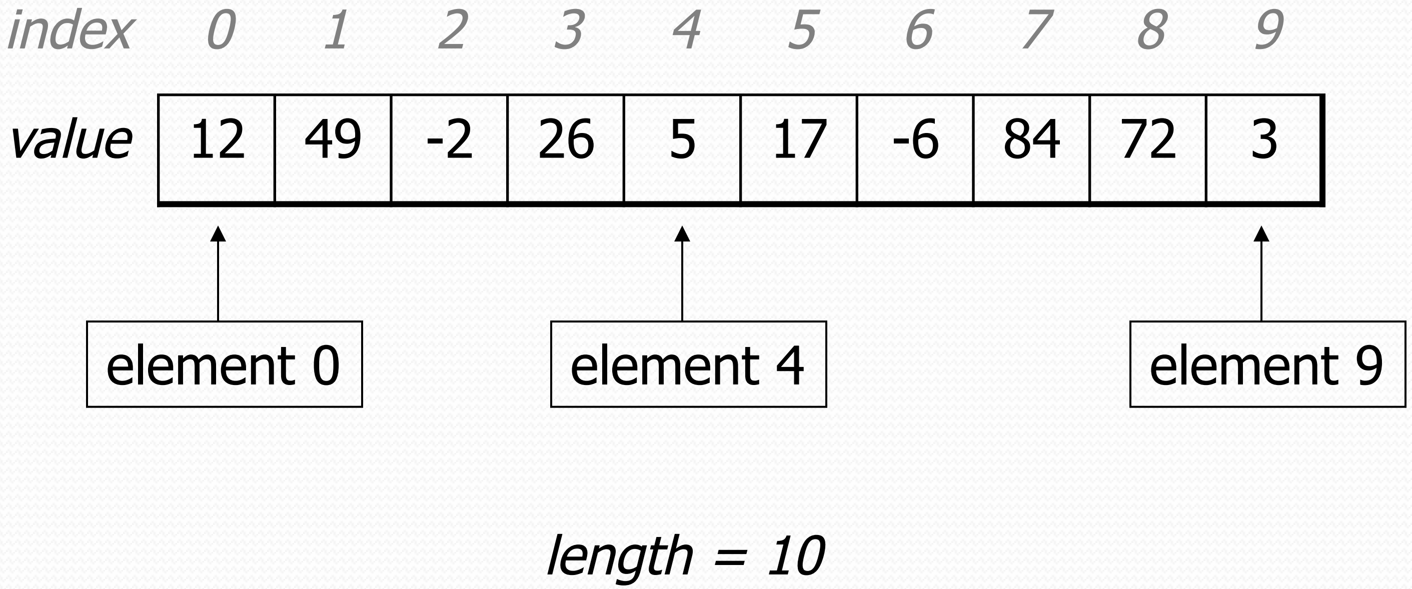
```
String[] allWords = new String[1000];
int wordCount = 0;

Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords[wordCount] = word;
    wordCount++;
}
```

- What's wrong with this?

Recall: Arrays (7.1)

- **array**: object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: 0-based integer to access an element from an array.
  - **length**: Number of elements in the array.



Array Limitations

- Fixed-size
- Adding or removing from middle is hard
- Not much built-in functionality (need Arrays class)

List Abstraction

- Like an array that resizes to fit its contents.

- When a list is created, it is initially empty.

```
[]
```

- Use `add` methods to add to different locations in list

```
[hello, ABC, goodbye, okay]
```

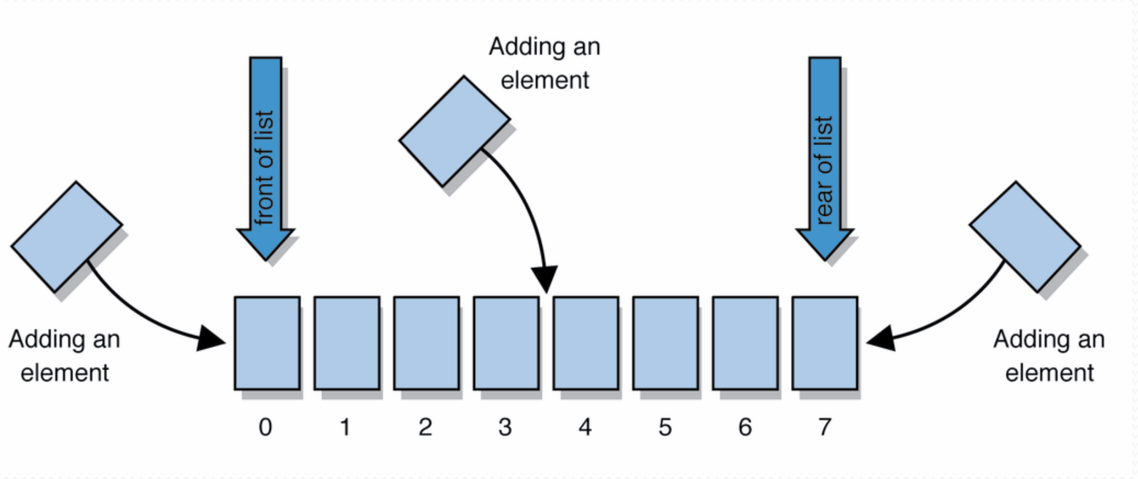
  - The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - You can add, remove, get, set, ... any index at any time.

Collections and lists

- **collection**: an object that stores data ("**elements**")

```
import java.util.*;    // to use Java's collections
```

- **list**: a collection of elements with 0-based **indexes**
  - elements can be added to the front, back, or elsewhere
  - a list has a **size** (number of elements that have been added)
  - in Java, a list can be represented as an **`ArrayList`** object



Type parameters (generics)

`ArrayList<`**`Type`**`>` **`name`** `= new ArrayList<`**`Type`**`>();`

- When constructing an `ArrayList`, you must specify the type of its elements in `< >`
  - This is called a *type parameter* ; `ArrayList` is a *generic* class.
  - Allows the `ArrayList` class to store lists of different types.
  - Arrays use a similar idea with **Type**`[]`

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

`ArrayList` methods (10.1)*

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

* *(a partial list; see 10.1 for other methods)*

ArrayList vs. array

```
String[] names = new String[5];                 // construct
names[0] = "Jessica";                           // store
String s = names[0];                            // retrieve
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}                                               // iterate
```

```
ArrayList<String> list = new ArrayList<String>();
list.add("Jessica");                            // store
String s = list.get(0);                         // retrieve
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}                                               // iterate
```

ArrayList as param/return

```
public static void name(ArrayList<Type> name) {// param
public static ArrayList<Type> name(params)     // return
```

- Example:

```
// Returns count of plural words in the given list.
public static int countPlural(ArrayList<String> list) {
    int count = 0;
    for (int i = 0; i < list.size(); i++) {
        String str = list.get(i);
        if (str.endsWith("s")) {
            count++;
        }
    }
    return count;
}
```

Words exercise, revisited

- Write a program that reads a file and displays the words of that file as a list.
  - Then display the words in reverse order.
  - Then display them with all plurals (ending in "s") capitalized.
  - Then display them with all plural words removed.

Exercise solution (partial)

```
ArrayList<String> allWords = new ArrayList<String>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords.add(word);
}

// display in reverse order
for (int i = allWords.size() - 1; i >= 0; i--) {
    System.out.println(allWords.get(i));
}

// remove all plural words
for (int i = 0; i < allWords.size(); i++) {
    String word = allWords.get(i);
    if (word.endsWith("s")) {
        allWords.remove(i);
        i--;
    }
}
```

`ArrayList` implementation

- What is an `ArrayList`'s behavior?
  - add, remove, indexOf, etc
- What is an `ArrayList`'s state?
  - Many elements of the same type
  - For example, unfilled array

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

size 5

`ArrayIntList` implementation

- Simpler than `ArrayList<E>`
  - No generics (only stores `int`s)
  - Fewer methods: `add(`**value**`)`, `add(`**index**`, `**value**`)`, `get(`**index**`)`, `set(`**index**`, `**value**`)`, `size()`, `isEmpty()`, `remove(`**index**`)`, `indexOf(`**value**`)`, `contains(`**value**`)`, `toString()`,
- Fields?
  - `int[]`
  - `int` to keep track of the number of elements added
  - The default capacity (array length) will be 10

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**									

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;
    }
}
```

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 7.4)*

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;
    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(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**					$\longrightarrow$				

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`.
    (Why write this if we already have the `size` method?)

Implementing `remove`

- 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**				$\longleftarrow$					

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**									