Building Java Programs

Chapter 15

`ArrayIntList`

reading: 15.1

Welcome to CSE 143!

Context for CSE 143

CSE 142

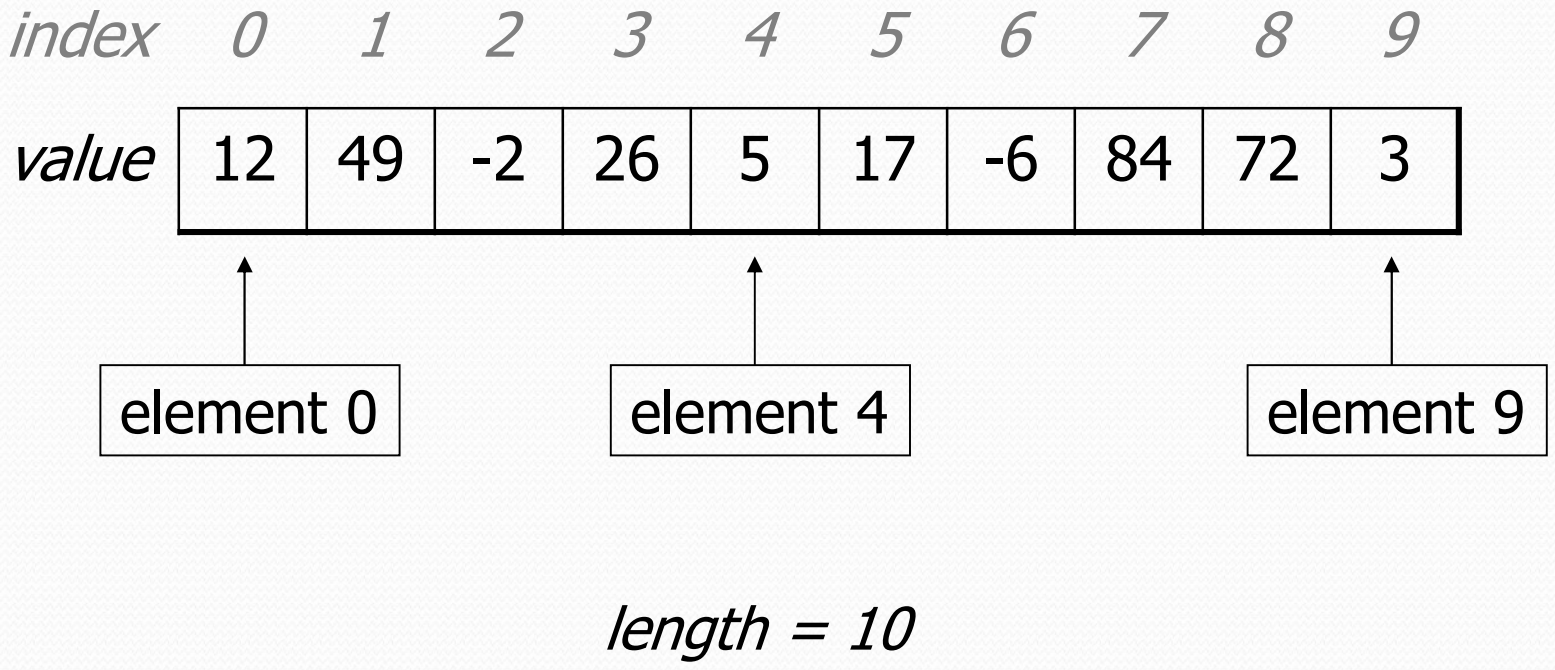
- Control: loops, if/else, methods, parameters, returns
- I/O: Scanners, user input, files
- Data: primitive types (int, double, etc.), *arrays*, *classes*

CSE 143

- Control: recursion
- Data
  - Java collections
  - Classes + Object Oriented Programming
- Best of CS

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)

Collections

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*

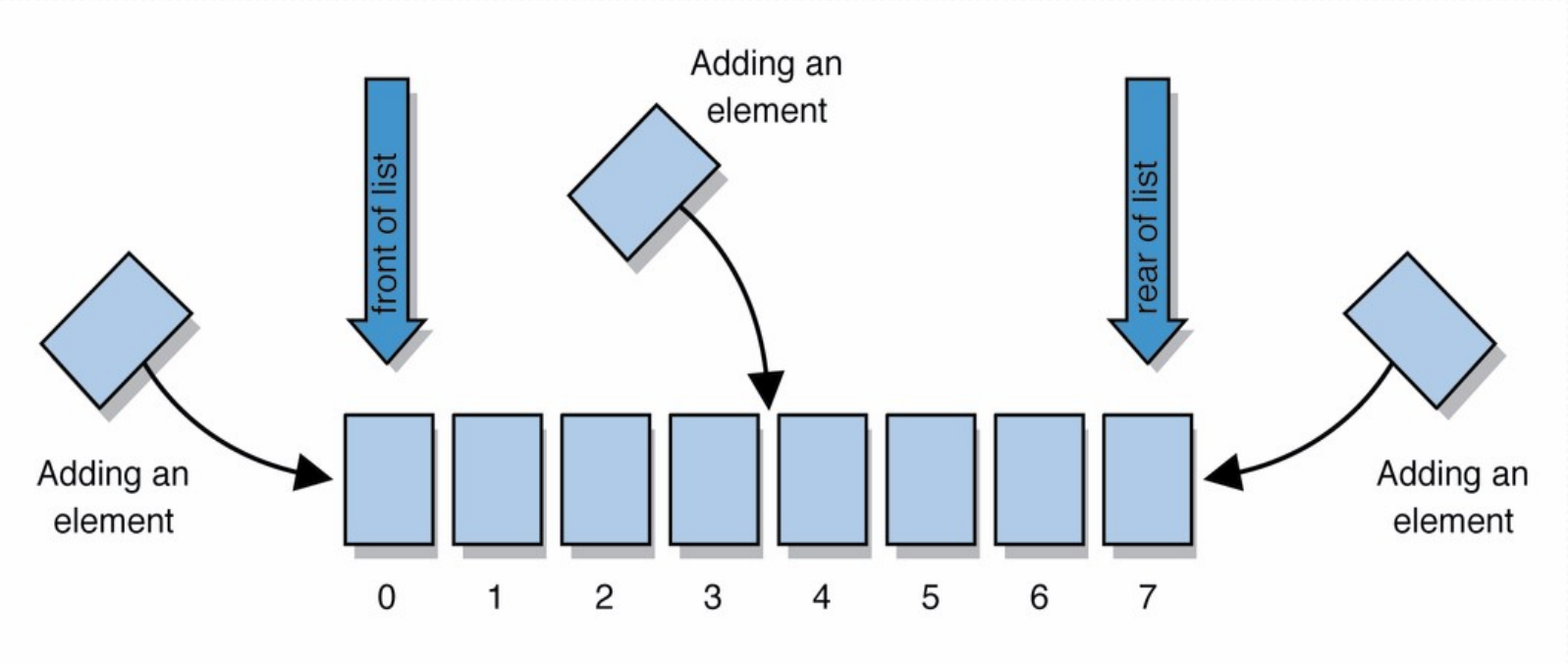
  - examples found in the Java class libraries:
    (covered in this course!)
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`

  - all collections are in the `java.util` package

```
import java.util.*;
```

Lists

- **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)
  - This is just a high level idea, haven't said how to do it in Java



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.

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]"`

ArrayList vs. array

- construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];
String s = list.get(0);
```

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;
}
```

Client - Radio

Implementer - Radio

Client – ArrayList

ArrayList<String> list:

["a", "b", "c"]

Implementer - ArrayList

String[] elementData:

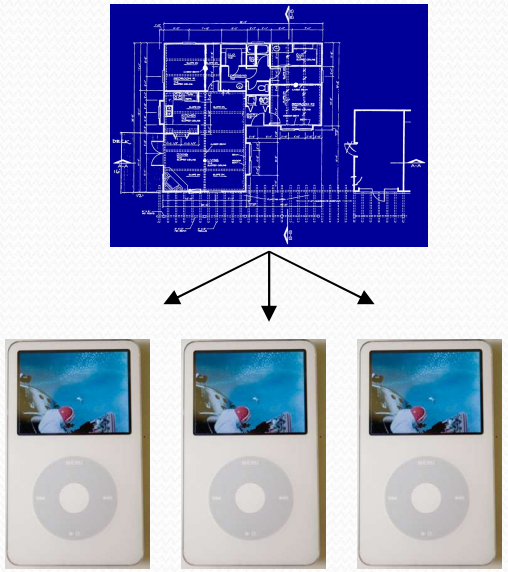
["a", "b", "c", null, null, null, null, null, null]

int size:

3

Recall: classes and objects

- **class**: A program entity that represents:
  - A complete program or module, or
  - A template for a type of objects.
  - (`ArrayList` is a class that defines a type.)



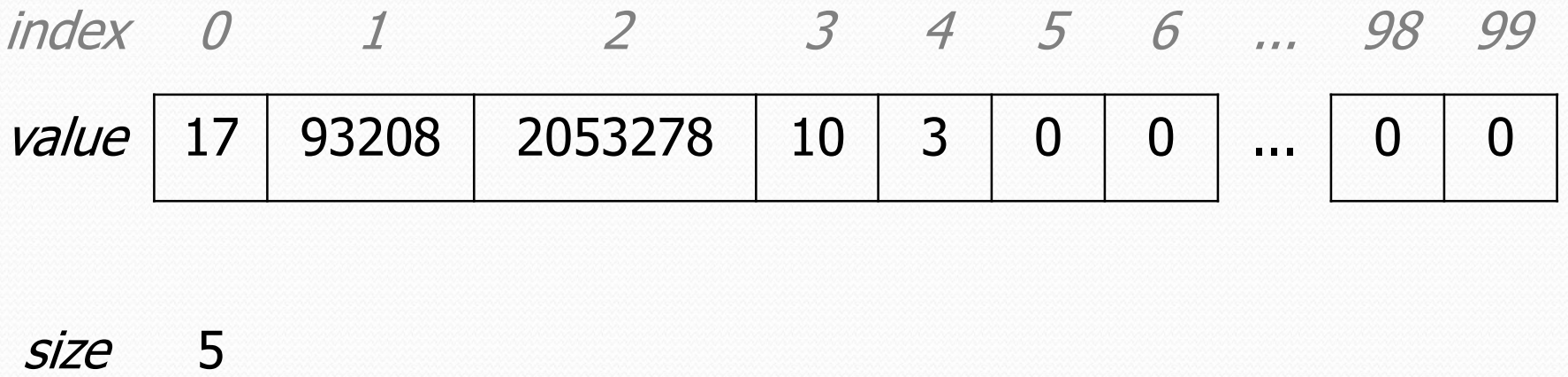
- **object**: An entity that combines **state** and **behavior**.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details. Objects provide abstraction in programming.

Elements of a class

```
public class BankAccount {
    private String name;
    private int id;
    private double balance;
  object

    public BankAccount(String
        this.name = name;
        this.id = id;
        this.balance = 0.0;
    }

    public void deposit(double

  method
    }
    ...
}
```

`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



`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;
    }
}
```