CSC 222: Object-Oriented Programming

Spring 2012

Lists, data storage & access

- ArrayList class
  - methods: add, get, size, remove, contains, set, indexOf, toString
  - generics, for-each loop
  - autoboxing & unboxing
- example: Dictionary
- Scanner class, file input, text processing
- example: LetterFreq
- ArrayLists vs. arrays

Composite data types

String is a composite data type

- each String object represents a collection of characters in sequence
- can access the individual components & also act upon the collection as a whole

many applications require a more general composite data type, e.g.,

- ✓ a dot race will keep track of a sequence/collection of dots
- ✓ a dictionary will keep track of a sequence/collection of words
- ✓ a payroll system will keep track of a sequence/collection of employee records

Java provides several library classes for storing/accessing collections of arbitrary items

ArrayList class

an `ArrayList` is a generic collection of *objects*, accessible via an index

- must specify the type of object to be stored in the list
- create an `ArrayList<?>` by calling the `ArrayList<?>` constructor (no inputs)

```
ArrayList<String> words = new ArrayList<String>();
```

- add items to the end of the `ArrayList` using `add`

```
words.add("Billy");           // adds "Billy" to end of list
words.add("Bluejay");         // adds "Bluejay" to end of list
```

- can access items in the `ArrayList` using `get`
  - similar to Strings, indices start at 0

```
String first = words.get(0);    // assigns "Billy"
String second = words.get(1);   // assigns "Bluejay"
```

- can determine the number of items in the `ArrayList` using `size`

```
int count = words.size();     // assigns 2
```

3

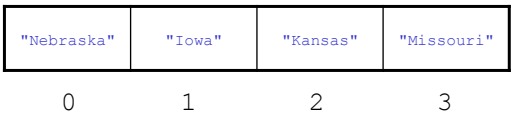
Simple example

```
ArrayList<String> words = new ArrayList<String>();

words.add("Nebraska");
words.add("Iowa");
words.add("Kansas");
words.add("Missouri");

for (int i = 0; i < words.size(); i++) {
    String entry = words.get(i);
    System.out.println(entry);
}
```

since an `ArrayList` is a composite object, we can envision its representation as a sequence of indexed memory cells



exercise:

- given an `ArrayList` of state names, output index where "Hawaii" is stored

4

For-each loop

traversing a list is such a common operation that a variant of for loops was introduced to make it easier/cleaner

```
for (TYPE value : ARRAYLIST) {
    PROCESS value
}
```

```
for (int i = 0; i < words.size(); i++) {
    String entry = words.get(i);
    System.out.println(entry);
}
```

```
for (String entry : words) {
    System.out.println(entry);
}
```

```
int count = 0;
for (int i = 0; i < words.size(); i++) {
    String nextWord = words.get(i);
    if (nextWord.length() > 5) {
        count++;
    }
}
```

```
int count = 0;
for (String nextWord : words) {
    if (nextWord.length() > 5) {
        count++;
    }
}
```

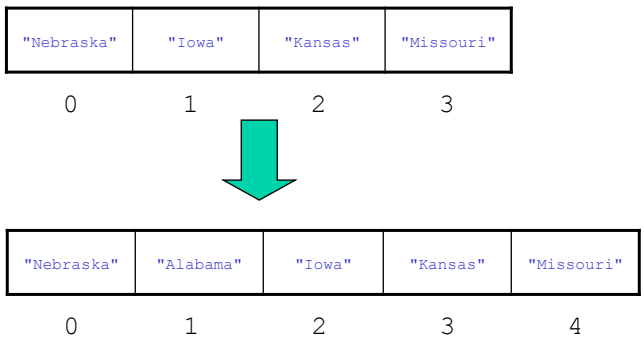
5

Other ArrayList methods: add at index

the general `add` method adds a new item at the end of the `ArrayList`

a 2-parameter version exists for adding at a specific index

```
words.add(1, "Alabama");   // adds "Alabama" at index 1, shifting
                           // all existing items to make room
```



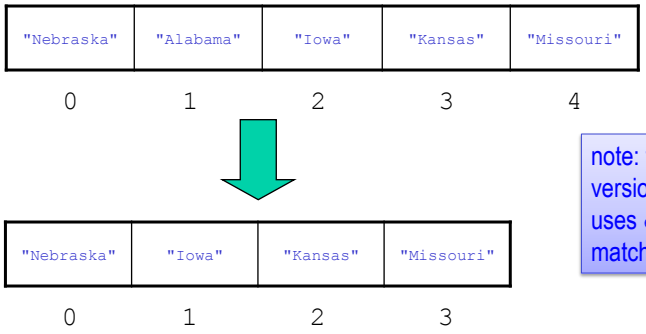
6

Other ArrayList methods: remove

in addition, you can remove an item using the `remove` method

- either specify the item itself or its index
- all items to the right of the removed item are shifted to the left

```
words.remove("Alabama");        words.remove(1);
```



note: the item version of remove uses `equals` to match the item

7

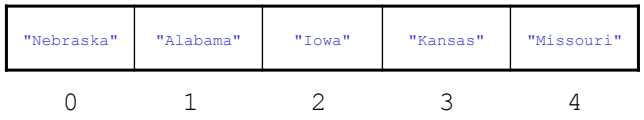
Other ArrayList methods: indexOf & toString

the `indexOf` method will search for and return the index of an item

- if the item occurs more than once, the first (smallest) index is returned
- if the item does not occur in the ArrayList, the method returns -1

```
words.indexOf("Kansas") → 3
words.indexOf("Alaska") → -1
```

similarly, `indexOf` uses `equals`



the `toString` method returns a String representation of the list

- items enclosed in [ ], separated by commas

```
words.toString() → "[Nebraska, Alabama, Iowa, Kansas, Missouri]"
```

- the `toString` method is automatically called when printing an ArrayList

```
System.out.println(words) = System.out.println(words.toString())
```

8

ArrayList<TYPE> methods

`TYPE get(int index)`	returns object at specified index
`TYPE set(int index, TYPE obj)`	sets entry at index to be obj
`boolean add(TYPE obj)`	adds obj to the end of the list
`void add(int index, TYPE obj)`	adds obj at index (shifts to right)
`TYPE remove(int index)`	removes object at index (shifts to left)
`boolean remove(TYPE obj)`	removes specified object (shifts to left) (assumes TYPE has an equals method)
`int size()`	returns number of entries in list
`boolean contains(TYPE obj)`	returns true if obj is in the list (assumes TYPE has an equals method)
`int indexOf(TYPE obj)`	returns index of obj in the list (assumes TYPE has an equals method)
`String toString()`	returns a String representation of the list e.g., "[foo, bar, biz, baz]"

9

Dot race revisited

we could modify the DotRace class to store a list of Dots

- the constructor adds multiple Dots to the list field
- the step method traverses the list, moving and displaying each Dot
- the reset method traverses the list, resetting and displaying each Dot

if we wanted to add more dots?

any class that uses an `ArrayList` must load the library file that defines it

```
import java.util.ArrayList;

public class DotRaceList {
    private ArrayList<Dot> dots;

    public DotRaceList(int maxStep) {
        this.dots = new ArrayList<Dot>();
        this.dots.add(new Dot("red", maxStep));
        this.dots.add(new Dot("blue", maxStep));
    }

    public void step() {
        for (Dot nextDot: this.dots) {
            nextDot.step();
            nextDot.showPosition();
        }
    }

    public void reset() {
        for (Dot nextDot: this.dots) {
            nextDot.reset();
            nextDot.showPosition();
        }
    }
}
```

10

HoopsScorer revisited

similarly, we could generalize HoopsScorer using lists

```
private int freeThrowsTaken;
private int twoPointersTaken;
private int threePointersTaken;
```

```
private ArrayList<int> taken;
```

- unfortunately, ArrayLists can only store object types (i.e., no primitives)
- fortunately, there exists a class named `Integer` that encapsulates an `int` value

```
private ArrayList<Integer> taken;
```

- the Java compiler will automatically
  - convert an `int` value into an `Integer` object when you want to store it in an ArrayList (called *autoboxing*)
  - convert an `Integer` value back into an `int` when need to apply an arithmetic operation on it (called *unboxing*)

11

HoopsScorer revisited

need a list of Integers for # of shots taken & made

- the constructor initializes the lists and adds 4 0's
  WHY 4?
- can remove redundant if-else cases in other methods

```
import java.util.ArrayList;

public class HoopsScorerList {
  private ArrayList<Integer> taken;
  private ArrayList<Integer> made;

  public HoopsScorerList() {
    taken = new ArrayList<Integer>();
    made = new ArrayList<Integer>();
    for (int i = 0; i <=3 ; i++) {
      this.taken.add(0);
      this.made.add(0);
    }
  }

  public void recordMissedShot(int numPoints) {
    if (1 <= numPoints && numPoints <= 3) {
      this.taken.set(numPoints,
                     this.taken.get(numPoints)+1);
    }
    else {
      System.out.println("ERROR");
    }
  }

  . . .
}
```

BE CAREFUL: Java will not unbox an Integer for comparison

```
if (this.taken.get(1) == this.made.get(1)) {
    …
}
```

== will test to see if they are the same Integer objects

12

Dictionary class

consider designing a simple class to store a list of words

- will store words in an `ArrayList<String>` field
- constructor initializes the field to be an empty list
- `addWord` method adds the word to the list, returns true
- `addWordNoDupes` method adds the word if it is not already stored, returns true if added
- `findWord` method determines if the word is already stored
- `display` method displays each word, one-per-line

```
import java.util.ArrayList;

public class Dictionary {
  private ArrayList<String> words;

  public Dictionary() {
    this.words = new ArrayList<String>();
  }

  public boolean addWord(String newWord) {
    this.words.add(newWord);
    return true;
  }

   public boolean addWordNoDupes(String newWord) {
    if (!this.findWord(newWord)) {
      return this.addWord(newWord);
    }
    return false;
  }

  public boolean findWord(String desiredWord) {
    return this.words.contains(desiredWord);
  }

  public int numWords() {
    return this.words.size();
  }

  public void display() {
    for (String nextWord : words) {
      System.out.println(nextWord);
  }
}
```


In-class exercises

download Dictionary.java and try it out

✓ make it so that words are stored in lower-case
- which method(s) need to be updated?

✓ add a method for removing a word

```
/**
 * Removes a word from the Dictionary.
 *   @param desiredWord the word to be removed
 *   @return true if the word was found and removed; otherwise, false
 */
public boolean removeWord(String desiredWord)
```

✓ add a method for finding & returning all partial matches

```
/**
 * Returns an ArrayList of words that contain the specified substring.
 *   @param substr the substring to match
 *   @return an ArrayList containing all words that contain the substring
*/
public ArrayList<String> findMatches(String substr)
```

Input files

adding dictionary words one-at-a-time is tedious

- better option would be reading words directly from a file

- `java.io.File` class defines properties & behaviors of text files
- `java.util.Scanner` class provides methods for easily reading from files

```
import java.io.File;
import java.util.Scanner;

. . .

public Dictionary(String fileName) throws java.io.FileNotFoundException {
  this.words = new ArrayList<String>();

  Scanner infile = new Scanner(new File(fileName));
  while (infile.hasNext()) {
    String nextWord = infile.next();
    this.addWord(nextWord);
  }
  infile.close();
}
```

this addition to the constructor header acknowledges that an error could occur if the input file is not found

opens a text file with the specified name for input

while there are still words to be read from the file, read a word and store it in the Dictionary

15

User input

the Scanner class is useful for reading user input as well

- can create a Scanner object connected to the keyboard (`System.in`)

```
Scanner input = new Scanner(System.in);
```

- can then use Scanner methods to read input entered by the user

`input.next()`	will read the next String (delineated by whitespace)
`input.nextLine()`	will read the next line of text
`input.nextInt()`	will read the next integer
`input.nextDouble()`	will read the next double

```
Scanner input = new Scanner(System.in);

System.out.println("Enter your first name: ");
String firstName= input.next();

System.out.println("Enter your age: ");
int age = input.nextInt();
```

16

Output files

for outputting text to a file, the FileWriter class is easy to use

- `java.io.FileWriter` class provides methods for easily writing to files

```
import java.io.FileWriter;

. . .

public void saveToFile(String filename) throws java.io.IOException {
  FileWriter outfile = new FileWriter(new File(filename));
  for (String nextWord : this.words) {
    outfile.write(nextWord + "\n");
  }
  outfile.close();
}
```

this addition to the method header acknowledges that an error could occur if the file cannot be opened

opens a text file with the specified name for output, writes each word to the file (followed by the end-of-line char)

17

HW4: spell checker

you will implement a class that serves as a simple spell checker

- specify a dictionary file when constructing a `SpellChecker`
- `dictionary.txt` in class code folder contains 117K word dictionary
- can utilize the `Dictionary` class to store the words

- can then call a `checkFile` method to process a file with a given name
  will need to be able to trim non-letters off the front/end of words

  `"He yelled 'Hi.' to her."` → `"He" "yelled" "Hi" "to" "her"`

- for each misspelled word, user is given the option to
  1. add that word to the dictionary,
  2. ignore that occurrence of the word, or
  3. ignore every occurrence of the word in that file.

- when `checkFile` is done, it should update the dictionary file

18

Letter frequency

one common tool for identifying the author of an unknown work is letter frequency

- i.e., count how frequently each of the letters is used in the work
- analysis has shown that an author will tend to have a consistent pattern of letter usage

will need 26 counters, one for each letter

- traverse each word and add to the corresponding counter for each character
- having a separate variable for each counter is not feasible
- instead have an ArrayList of 26 counters

  `this.counts.get(0)` is the counter for 'a'
  `this.counts.get(1)` is the counter for 'b'
  ...
  `this.counts.get(25)` is the counter for 'z'

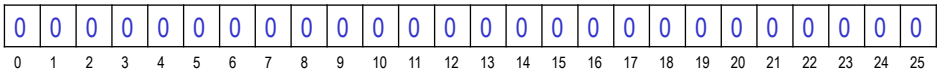
letter frequencies from the Gettysburg address

```
a:    93 ( 9.2%)
b:    12 ( 1.2%)
c:    28 ( 2.8%)
d:    49 ( 4.9%)
e:   150 (14.9%)
f:    21 ( 2.1%)
g:    23 ( 2.3%)
h:    65 ( 6.4%)
i:    59 ( 5.8%)
j:     0 ( 0.0%)
k:     2 ( 0.2%)
l:    39 ( 3.9%)
m:    14 ( 1.4%)
n:    71 ( 7.0%)
o:    81 ( 8.0%)
p:    15 ( 1.5%)
q:     1 ( 0.1%)
r:    70 ( 6.9%)
s:    36 ( 3.6%)
t:   109 (10.8%)
u:    15 ( 1.5%)
v:    20 ( 2.0%)
w:    26 ( 2.6%)
x:     0 ( 0.0%)
y:    10 ( 1.0%)
z:     0 ( 0.0%)
```

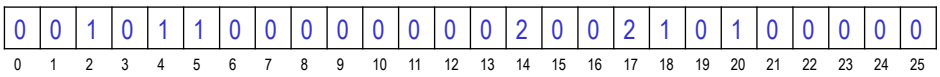
19

Letter frequency example

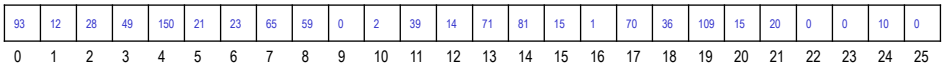
initially, have ArrayList of 26 zeros:



after processing `"Fourscore" :`



after processing the entire Gettysburg address



20

LetterFreq1 design

```
public class LetterFreq1 {
  private ArrayList<Integer> counts;
  private int numLetters;

  public LetterFreq1(String fileName) throws java.io.FileNotFoundException {
    INITIALIZE this.counts AND this.numLetters

    FOR EACH WORD IN THE FILE
      FOR EACH CHARACTER IN THE WORD
        IF THE CHARACTER IS A LETTER
           DETERMINE ITS POSITION IN THE ALPHABET
           INCREMENT THE CORRESPONDING COUNT IN this.counts
           INCREMENT  this.numLetters
  }

  public int getCount(char ch) {
    IF ch IS A LETTER
      DETERMINE ITS POSITION IN THE ALPHABET
      ACCESS & RETURN THE CORRESPONDING COUNT IN this.counts
    OTHERWISE
      RETURN 0
  }

  public double getPercentage(char ch) {
    IF ch IS A LETTER
      DETERMINE ITS POSITION IN THE ALPHABET
      ACCESS THE CORRESPONDING COUNT IN this.counts
      CALCULATE & RETURN THE PERCENTAGE
    OTHERWISE
      RETURN 0.0
  }

  public void showCounts() {
    FOR EACH LETTER IN THE ALPHABET
      DISPLAY THE LETTER, ITS COUNT & PERCENTAGE
  }
}
```

21

LetterFreq1 implementation

```
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class LetterFreq1 {
  private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
  private ArrayList<Integer> counts;
  private int numLetters;

  public LetterFreq1(String fileName) throws java.io.FileNotFoundException {
    this.counts = new ArrayList<Integer>();
    for (int i = 0; i < LetterFreq1.LETTERS.length(); i++) {
      this.counts.add(0);
    }
    this.numLetters = 0;

    Scanner infile = new Scanner(new File(fileName));
    while (infile.hasNext()) {
      String nextWord = infile.next();

      for (int c = 0; c < nextWord.length(); c++) {
        char ch = nextWord.charAt(c);
        if (Character.isLetter(ch)) {
          int index = LetterFreq1.LETTERS.indexOf(Character.toLowerCase(ch));
          this.counts.set(index, this.counts.get(index)+1);
          this.numLetters++;
        }
      }
    }
  }

  . . .
```

will use a constant to store the alphabet

initialize the letter counts

for each word, process each letter …

… get letter's index, increment its count

22

LetterFreq1 implementation (cont.)

```
  . . .

  public int getCount(char ch) {
    if (Character.isLetter(ch)) {
      int index = LetterFreq1.LETTERS.indexOf(Character.toLowerCase(ch));
      return this.counts.get(index);
    }
    else {
      return 0;
    }
  }

  public double getPercentage(char ch) {
    if (Character.isLetter(ch) && this.numLetters > 0) {
      int index = LetterFreq1.LETTERS.indexOf(Character.toLowerCase(ch));
      return Math.round(1000.0*this.counts.get(index)/this.numLetters)/10.0;
    }
    else {
      return 0.0;
    }
  }

  public void showCounts() {
    for (int i = 0; i < LetterFreq1.LETTERS.length(); i++) {
      char ch = LetterFreq1.LETTERS.charAt(i);
      System.out.println(ch + ": " + this.getCount(ch) + "\t(" +
                         this.getPercentage(ch) + "%)");
    }
  }
}
```

if it is a letter, access & return its count

if it is a letter, calculate & return its percentage

display all letters, counts and percentages

23

Interesting comparisons

letter frequencies from the Gettysburg address	letter frequencies from Alice in Wonderland	letter frequencies from Theory of Relativity
a: 93 (9.2%)	a: 8791 (8.2%)	a: 10936 (7.6%)
b: 12 (1.2%)	b: 1475 (1.4%)	b: 1956 (1.4%)
c: 28 (2.8%)	c: 2398 (2.2%)	c: 5272 (3.7%)
d: 49 (4.9%)	d: 4930 (4.6%)	d: 4392 (3.1%)
e: 150 (14.9%)	e: 13572 (12.6%)	e: 18579 (12.9%)
f: 21 (2.1%)	f: 2000 (1.9%)	f: 4228 (2.9%)
g: 23 (2.3%)	g: 2531 (2.4%)	g: 2114 (1.5%)
h: 65 (6.4%)	h: 7373 (6.8%)	h: 7607 (5.3%)
i: 59 (5.8%)	i: 7510 (7.0%)	i: 11937 (8.3%)
j: 0 (0.0%)	j: 146 (0.1%)	j: 106 (0.1%)
k: 2 (0.2%)	k: 1158 (1.1%)	k: 568 (0.4%)
l: 39 (3.9%)	l: 4713 (4.4%)	l: 5697 (4.0%)
m: 14 (1.4%)	m: 2104 (2.0%)	m: 3253 (2.3%)
n: 71 (7.0%)	n: 7013 (6.5%)	n: 9983 (6.9%)
o: 81 (8.0%)	o: 8145 (7.6%)	o: 11181 (7.8%)
p: 15 (1.5%)	p: 1524 (1.4%)	p: 2678 (1.9%)
q: 1 (0.1%)	q: 209 (0.2%)	q: 344 (0.2%)
r: 70 (6.9%)	r: 5437 (5.0%)	r: 8337 (5.8%)
s: 36 (3.6%)	s: 6500 (6.0%)	s: 8982 (6.2%)
t: 109 (10.8%)	t: 10686 (9.9%)	t: 15042 (10.5%)
u: 15 (1.5%)	u: 3465 (3.2%)	u: 3394 (2.4%)
v: 20 (2.0%)	v: 846 (0.8%)	v: 1737 (1.2%)
w: 26 (2.6%)	w: 2675 (2.5%)	w: 2506 (1.7%)
x: 0 (0.0%)	x: 148 (0.1%)	x: 537 (0.4%)
y: 10 (1.0%)	y: 2262 (2.1%)	y: 2446 (1.7%)
z: 0 (0.0%)	z: 78 (0.1%)	z: 115 (0.1%)

24

ArrayLists and arrays

ArrayList enables storing a collection of objects under one name

- can easily access and update items using `get` and `set`
- can easily `add` and `remove` items, and shifting occurs automatically
- can pass the collection to a method as a single object

ArrayList is built on top of a more fundamental Java data structure: the *array*

- an array is a *contiguous*, *homogeneous* collection of items, accessible via an index
- arrays are much less flexible than ArrayLists
  - ✓ *the size of an array is fixed at creation, so you can't add items indefinitely*
  - ✓ *when you add/remove from the middle, it is up to you to shift items*
  - ✓ *you have to manually keep track of how many items are stored*
- for fixed size lists, arrays can be simpler

Arrays

to declare an array, designate the type of value stored followed by []

```
String[] words;                    int[] counters;
```

to create an array, must use `new` (an array is an object)

- specify the type and size inside brackets following `new`

```
words = new String[100];           counters = new int[26];
```

- or, if you know what the initial contents of the array should be, use shorthand:

```
int[] years = {2001, 2002, 2003, 2004, 2005};
```

to access or assign an item in an array, use brackets with the desired index

- similar to the `get` and `set` methods of ArrayList

```
String str = word[0];              // note: index starts at 0
                                   // (similar to ArrayLists)
for (int i = 0; i < 26, i++) {
    counters[i] = 0;
}
```

LetterFreq2 implementation

```
import java.util.Scanner;
import java.io.File;

public class LetterFreq2 {
  private static final String LETTERS = "abcdefghijklmnopqrstuvwxyz";
  private int[] counts;
  private int numLetters;

  public LetterFreq2(String fileName) throws java.io.FileNotFoundException {
    this.counts = new int[LetterFreq2.LETTERS.length()];
    for (int i = 0; i < LetterFreq2.LETTERS.length(); i++) {
      this.counts[i] = 0;
    }
    this.numLetters = 0;

    Scanner infile = new Scanner(new File(fileName));
    while (infile.hasNext()) {
      String nextWord = infile.next();

      for (int c = 0; c < nextWord.length(); c++) {
        char ch = nextWord.charAt(c);
        if (Character.isLetter(ch)) {
          int index = LetterFreq2.LETTERS.indexOf(Character.toLowerCase(ch));
          this.counts[index]++;
          this.numLetters++;
        }
      }
    }
  }

  . . .
```

could instead make the field an array

initialize array to desired size

access/assign an entry using [ ]

increment is simpler (no need to get then set)

LetterFreq2 implementation (cont.)

```
  . . .

  public int getCount(char ch) {
    if (Character.isLetter(ch)) {
      int index = LetterFreq2.LETTERS.indexOf(Character.toLowerCase(ch));
      return this.counts[index];
    }
    else {
      return 0;
    }
  }

  public double getPercentage(char ch) {
    if (Character.isLetter(ch) && this.numLetters > 0) {
      int index = LetterFreq2.LETTERS.indexOf(Character.toLowerCase(ch));
      return Math.round(1000.0*this.counts[index]/this.numLetters)/10.0;
    }
    else {
      return 0.0;
    }
  }

  public void showCounts() {
    for (int i = 0; i < LetterFreq1.LETTERS.length(); i++) {
      char ch = LetterFreq2.LETTERS.charAt(i);
      System.out.println(ch + ": " + this.getCount(ch) + "\t(" +
                         this.getPercentage(ch) + "%)");
    }
  }
}
```

other method essentially the same (array access uses [ ] instead of the get method for ArrayLists)

Why arrays?

general rule: ArrayLists are better, more abstract – USE THEM!

- they provide the basic array structure with many useful methods provided for free

```
get, set, add, size, contains, indexOf, remove, …
```

- plus, the size of an ArrayList automatically adjusts as you add/remove items

when *might* you want to use an array?

- if the size of the list will never change and you merely want to access/assign items, then the advantages of arrays may be sufficient to warrant their use
  - ✓ if the initial contents are known, they can be assigned when the array is created

```
String[] answers = { "yes", "no", "maybe" };
```

  - ✓ the [] notation allows for both access and assignment (instead of get & set)

```
int[] counts = new int[11];
. . .
counts[die1.roll() + die2.roll()]++;
```

  - ✓ you can store primitive types directly, so no autoboxing/unboxing

29