Structured Data in Java
ArrayLists

The `ArrayList` Class

- `ArrayList` is a class in the Java Collections Framework
  - It implements the `List` interface

- An `ArrayList` serves the same purpose as an array, except that `ArrayList` supports dynamic arrays that can grow as needed while the program is running
  - Unlike arrays, which have a fixed length once they have been created

The `ArrayList` Class

- The class `ArrayList` is implemented using an array as a private instance variable
  - Every `ArrayList` object has its own array hidden inside it
  - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array
    - The `ArrayList` object takes care of all the details for you
  - An `ArrayList` does have indices and the positions of the elements range from index 0 to index `size()`-1

`ArrayList` Properties

- An `ArrayList` tracks its *logical size* and *physical size*
- The `size()` method is used to get the logical size; i.e., the number of elements actually stored in the list
- We don’t necessarily know the physical size and we don’t need to
  - If more memory locations are needed then it will automatically resize itself
- The logical size is 0 when an `ArrayList` is constructed, and its logical size is automatically adjusted as elements are added or deleted

Advantages of `ArrayList`

- An `ArrayList` tracks its own logical size and grows or shrinks automatically depending on the number of elements it has

- When compared to arrays, operations are much easier and less complex for:
  - Insertions anywhere in the `ArrayList`
  - Removals anywhere in the `ArrayList`
  - Searching an `ArrayList`
  - [Traversing an `ArrayList` vs. an array is about the same]

Disadvantages of `ArrayList`

- Why not always use an **`ArrayList`** instead of an array?

1. An **`ArrayList`** is slightly less efficient than an array
2. It does not have the convenient square bracket notation
   - You use method calls instead
3. Multi-dimensional **`ArrayList`** is non-trivial
4. The base type of an **`ArrayList`** cannot be a primitive type, rather it must be a class type (or other reference type)
   - This is less of a problem now that Java provides automatic boxing and unboxing of primitives with wrapper classes

Methods in the Class `ArrayList`

- The tools for manipulating arrays consist only of the square brackets and the instance variable `length`

- However, `ArrayLists` come with a selection of powerful methods that can do many things for you
  - This is code which you would have been required to write yourself in order to do the same thing with arrays

Adding to an `ArrayList`

- The `add` method is usually used to place an element in an `ArrayList` for the first time
  - There are two versions

- The `add` method with a single parameter, for the element to be added, adds the element at the next unused index (i.e., at the end of the list)

- The `add` method with two parameters also specifies the index of where to add the element
  - All subsequent elements are shifted up one spot

Deleting from an `ArrayList`

- The `remove` method is used to delete an element in an `ArrayList`
  - There are two versions
- One version specifies the object to be deleted, and deletes the first occurrence found of that object
- The other version specifies the index of the element to be deleted
- For both, all subsequent elements are shifted down one spot

Other Common `ArrayList` Methods

- The `get` & `set` methods can retrieve or change any individual element based on an index
  - However, the index must be for an element that already exists

- The method `size` can be used to determine how many elements are stored in an `ArrayList`

- Here’s a quick summary of the main methods of the class; see the reference documents on the web for details

ArrayList methods

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
get(index)	returns the value at given index
set(index, value)	replaces value at given index with given value
size()	returns the number of elements in list
isEmpty()	returns true if the list is empty, otherwise false
toString()	returns a string representation of the list such as "[43, 6, -13, 272]"

`ArrayList` methods

`remove(`index`)`	removes/returns value at given index, shifting subsequent values to the left
`remove(`value`)`	finds and removes the given value from this list, shifting subsequent values to the left
`indexOf(`value`)`	returns first index where given value is found in list (-1 if not found)
`lastIndexOf(`value`)`	returns last index where value is found in list (-1 if not found)
`contains(`value`)`	returns true if given value is found somewhere in this list
`containsAll(`list`)`	returns true if this list contains every element from given list

`ArrayList` methods

`equals(`list`)`	returns true if given other list contains the same elements in same order

`iterator()`
`listIterator()`
	returns an object used to examine the contents of the list

`clear()`	removes all elements of the list

Plus more…

See the reference documents on the web for a complete list

Choosing the datatype

- When you declare an ArrayList object, you *should* use the List interface as the datatype instead of ArrayList
  - That will allow you to change to a LinkedList implementation in the future, if needed, by changing a single line of code!

```
List<String> aList = new ArrayList<String>();
```

👍

🚫

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

- The latter is not necessary wrong, just not recommended

Alternate Constructor

- If you know approximately the number of data elements you will add to the ArrayList, you can specify a desired size on the constructor

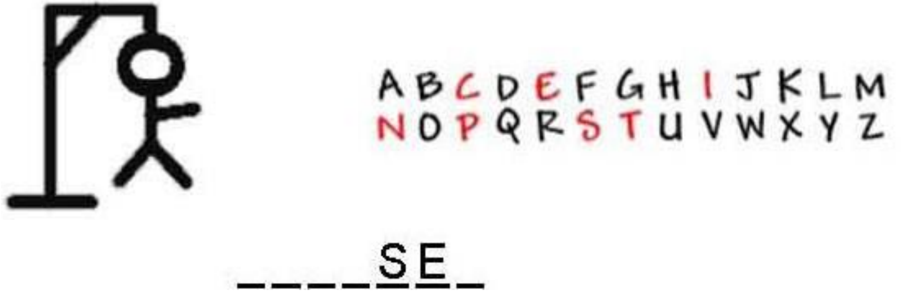
- The constructor will allocate a physical array of the specified size

- The constructed ArrayList will still be empty with a logical **`size()`** of zero

```
List<String> aList = new ArrayList<String>(250);
```

Extended Example

- Let’s consider the game of Hangman
- One player thinks of a word and shares its length
- The other player tries to guess it by suggesting letters
- If a letter is correct, the first player informs the second which positions in the word hold that letter
- If the letter is incorrect, then that is a miss



Extended Example

- Let’s consider the game of Hangman
- One player thinks of a word and shares its length
- The other player tries to guess it by suggesting letters
- If a letter is correct, the first player informs the second which positions in the word held that letter
- If the letter is incorrect, then that is a miss
- The game is over when the word is guessed, and the second player wins, or the number of misses exceeds a limit, and the first player wins

Extended Example

- We will not develop a complete Hangman game in this example
- We will develop some methods that could assist a player in guessing the hidden word
- We will use an ArrayList of strings to hold all possible words
- We will then write methods that will filter out words that hold incorrect letters or are missing required letters in given positions

Extended Example

- Let’s start by writing a method that will create and return an ArrayList that holds the initial set of valid words
    - We will load the Official Scrabble Players Dictionary
    - Since reading a file on Android is non-trivial, we will assume someone has read all the words and created an array of strings
- We will only load the ArrayList with words that meet the length requirement
    - We don’t know how many words will meet the length requirement, so using an ArrayList is better than an array

Extended Example

```
public static List<String> loadWords(int len,
                                     String[] ospd)
{
  List<String> words = new ArrayList<String>(1000);
  for (String word : ospd) {
    if (word.length()==len) {
      words.add(word);
    }
  }
  return words;
}
```

Extended Example

- Next, let’s write a method that will filter out words that do not have a specified letter at a given index

- The method will take three parameters
  - The specified letter
  - The position where the letter is expected
  - The ArrayList containing words

Extended Example

```
public static void mustHaveAt(char ch, int position,
                            List<String> aList)
{
  for (int i=aList.size()-1; i>=0; i--) {
    String word = aList.get(i);
    if (position >= word.length() ||
        word.charAt(position)!=ch)
    {
      aList.remove(i);
    }
  }
}
```

Extended Example

- Finally, let’s write a method that will filter out words that contain an invalid letter

- The method will take two parameters
  - The invalid letter
  - The ArrayList containing words

- In this case we will use an iterator to process the list

Extended Example

```
public static void mustNotHave(char ch,
                               List<String> aList)
{
  Iterator<String> itr = aList.iterator();
  while (itr.hasNext()) {
    String word = itr.next();
    if (word.indexOf(ch)>=0) {
      itr.remove();
    }
  }
}
```