Introduction to ArrayLists

ArrayList concepts, syntax, operations, examples

List Interface

A **list** is a sequence of elements. As an abstract data type, a list should support several operations: create(), isEmpty(), size(), add(index, item), remove(index), and get(index).

`List<E>` (or simply `List`) is an interface in the Java Collections Framework that declares list operations.

```
interface List<E>
```

- **Java Collections Framework**: Part of the API that provides code for collections of objects.
- <E> notation. E stands for element type, which can be any object/ class type.
- Brace notation: List<E> is a **parameterized type**, where the parameter (inside the brace) stands for the element type.

ArrayList

`ArrayList<E>` is an implementation of the `List<E>` interface.

```
ArrayList<E> implements List<E>
```

- `ArrayList` implements a list in which the underlying data structure is an array.
  - `ArrayList` elements are contiguous in memory.
  - `ArrayLists` are indexed.
  - `ArrayLists` have capacity.
  - `ArrayLists` automatically expand their capacity when needed to hold more elements.

Example ArrayLists

All of the Java collection types (classes and interfaces) are generic with a parameter <E> that stands for the type of element:

- `interface List<E>`
- `class ArrayList<E>`
- `class LinkedList<E>`

Examples create lists of String elements. (Types are parameterized with String.)

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

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

List operations

`ArrayList` supports many operations, including these:

- `int size()` – number of elements in the list
- `boolean isEmpty()` – true if the list contains no elements
- `boolean contains(E item)` – true if `item` is a list element
- `int indexOf(Object item)` – the index of `item` in the list (0-based indexing)
- `void add(E item)` – adds `item` to the end of the list
- `void add(int index, E item)` – inserts `item` to the list at index. Subsequent elements shift to one index higher.
- `E remove(int index)` – removes and returns the element at `index`. Subsequent elements shift to one index lower.
- `void set(int index, E item)` Replaces the element at `index` with `item`
- `E get(int index)` – gets the element at the given index

ArrayList characteristics

The underlying data structure is an array

- ArrayList elements are contiguous.
  - Inserting a new element into an ArrayList causes a shift of subsequent elements toward the end.
  - Removing an element from an ArrayList cause a shift of subsequent elements to the front.
- ArrayLists are indexed. The first element is at index 0.
- Each ArrayList has a capacity.
- The capacity of an ArrayList increases as needed to add more elements.

Creating ArrayList instances

Declare an ArrayList with the element type in diamond braces, < >.

Three constructors:

1. Null constructor.
2. Constructor with initial capacity (integer parameter).
3. Constructor taking another collection with elements of the same type.

You can omit the type in the constructor call, but you should put in the empty diamond, <>.

```
// Create an empty list of Strings
ArrayList<String> guests = new ArrayList<String>();

// Create an empty list with an initial capacity of 100
ArrayList<Integer> measures = new ArrayList<Integer>(100);

// Create a list as a copy of another list
ArrayList<String> invited = new ArrayList<String>(guests);

// Declaration using the empty diamond <>
ArrayList<String> cities = new ArrayList<>();
```

Using ArrayList operations

```
ArrayList<String> guests = new ArrayList<>();
// guests.isEmpty() is true

guests.add("Paul");
guests.add("Sue");
guests.add("Pete");
guests.add("Carl");
// guests: ["Paul", "Sue", "Pete", "Carl"]
// guests.indexOf("Paul") is 0

String name = guests.remove(1);
// name is "Sue"
// guests: ["Paul", "Pete", "Carl"]
// guests.size() is 3

guests.set(0, "New Guy");
// guests: ["New Guy", "Pete", "Carl"]

guests.add(1, "Molly");
// guests: ["New Guy", "Molly", "Pete", "Carl"]
```

Parameters and return types

You can declare methods that have ArrayList return types and methods that have ArrayList formal parameters.

```
public void printStrings(ArrayList<String> s) {
   for (String x: s)
      System.out.println(x);
}

public ArrayList<String> takeHalf(ArrayList<String> p) {
   ArrayList<String> r = new ArrayList<String> ();
   for (int i = 0; i < p.size(); i += 2)
      r.add(p.get(i));
   return r;
}
```

Errors: out of bounds indexes

```
ArrayList<String> guests = new ArrayList<>();
guests.add("Paul");
guests.add("Sue");
guests.add("Pete");
// guests: ["Paul", "Sue", "Pete"]

guests.add(10, "Oops");
```

The last line generates an IndexOutOfBoundsException:

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 10, Size: 3
        at java.util.ArrayList.rangeCheckForAdd(ArrayList.java:661)
        at java.util.ArrayList.add(ArrayList.java:473)
```

Valid index range for operations that take an index argument:

- add operation: `0` through `size()`
- get and remove operations: `0` through `size() - 1`