Topic 4: Arrays and ArrayLists

Reading: JBD Chapter 5
Arrays

- An array is an ordered collection of zero or more elements
- All the elements have the same type
- Creating an array has three steps:
  1. Declare the array
  2. Allocate storage for the array
  3. Populate the array with values
- These steps can sometimes be combined in a single statement, but first we will consider them separately.
Step 1: Declaring an Array

- Specify the name of the array and the type of its elements.

- Examples:
  ```java
  int[] a;       // array of ints
  boolean[] b;  // array of booleans
  String[] c;   // array of Strings
  ```

- An array declaration does NOT specify the size of the array or allocate storage for it (unlike in C)

  ```java
  int[5] a;       // OK in C but not in Java
  ```
Step 2: Allocating Storage for an Array

- Use the keyword `new` and specify the size of the array (the size need not be a constant)

- Examples:
  ```java
  a = new int[5];           // array of ints
  b = new boolean[mySize];  // array of booleans
  c = new String[big * 2];  // array of Strings
  ```

- By default, all the elements of the array are initialized to a value that depends on the type
  - For numeric types: 0 or 0.0
  - For booleans: `false`
  - For char: `'\u0000'`
  - For String and other reference types: `null`
Step 3: Populating the array with values

- Use "index" notation to access the elements of the array: \texttt{a[0]}, \texttt{a[1]}, \texttt{a[2]}, etc.
- Index must be an integer expression.
- Array elements are numbered starting with 0.
- Every array has a length field that evaluates to the number of elements in the array: \texttt{a.length}
  - Remember: \texttt{length} is not a method, so don't use ( )
- Examples:
  \begin{verbatim}
  a[0] = 47;
  c[0] = "Hello";
  for (int i = 0; i < a.length; i++)
    a[i] = 2 * i;
  \end{verbatim}
Shortcuts for Creating Arrays

- **Combining Steps 1 and 2:**
  Declare and allocate storage in one statement
  ```java
  int[] a = new int[5];
  ```

- **Combining Steps 1, 2, and 3:**
  Declare, allocate, and initialize using `{ }` notation
  ```java
  int[] a = {0, 1, 2, 3};
  String[] b = {"Hello", "Goodbye"};
  ```

  ![Indexing](b[0] b[1])
Array Literals

- Wherever an array is expected, an "array literal" can be used.
- Combines `new` keyword with `{ }` notation, similar to an array initializer
- Examples:
  
  ```java
  new int[] {1, 2, 3, 4, 5}
  new String[] {"Hello", "Goodbye"}
  ```
Arrays in Storage

```c
int n = 47;
int[] a = {61, 95, 32};
```

Stack

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>47</td>
<td>int</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>int[]</td>
</tr>
</tbody>
</table>

Heap

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>61</td>
<td>a[0]</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>a[1]</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>a[2]</td>
</tr>
</tbody>
</table>
Managing Array Storage (a closer look)

```java
int[] a;
a = new int[3];
a[1] = 47;
a = new int[4];
a[2] = 63;
```

Stack

- **Name**: `a`
- **Value**: `null`
- **Type**: `int[]`

Heap

- **Value 0**: `0`
- **Value 1**: `47`
- **Value 2**: `63`

Garbage Collection
Printing an Array

- Passing an array to `System.out.println` is not an error but it is not very useful.
- To print the content of an array, use a loop.
- Here's a method that prints an array of ints:

```java
static void print(int[] a) {
    System.out.print("{");
    for (int i = 0; i < a.length - 1; i++) {
        System.out.print(a[i] + ", ");
    }
    System.out.print(a[a.length - 1]);
    System.out.println("}");
}
```
Strings and char Arrays

- In Java, String is not the same type as char[].
- But you can convert between these types.
- To convert from String to char[], use
  ```java
  String s = "Hello";
  char[] a;
  a = s.toCharArray();  // a[0] is 'H' etc.
  ```
- To convert from char[] to String, use
  ```java
  String s;
  char[] a = {'H', 'e', 'l', 'l', 'o'};
  s = new String(a);    // s is now "Hello"
  ```
Indexing out of Bounds

- Unlike C, Java checks every array reference at run time.

  \[ a[\text{index}] \]

If \text{index} is less than 0 or greater than \( a.length-1 \), here's what you get:

  \text{ArrayIndexOutOfBoundsException}
Passing Arrays to Methods

- A method can take an array as a parameter:
  ```java
  static void triple(int[] a) {
      for (int i = 0; i < a.length; i++)
          a[i] *= 3;
  }
  ```

- What is the result of this experiment?
  ```java
  int[] scores = {1, 2, 3, 4};
  triple(scores);
  for (int i = 0; i < scores.length; i++)
      System.out.println(scores[i]);
  ```

- Summary:
  - The method is passed a *reference* to the argument array
  - Unlike Strings, arrays are mutable types and can be modified by passing them to methods
Iterating over an Array

- For-loops are often used to iterate over the elements of an array:

  ```java
  for (int i = 0; i < a.length; i++) {
    // do something to a[i]
  }
  ```

- Java Version 5 introduced a better way to iterate over the elements of an array:

  ```java
  for (int n: a) {
    // do something to n
  }
  ```

- **Advantages:**
  - Saves writing lots of subscripting expressions
  - No worries about subscripts out of bounds
Iterating over an Array (a closer look)

for elementType elementVariableName : arrayExpression {
    // loop over the elements, binding elementVariableName to each
}

Examples:

for (double d: myArrayOfDoubles) {
    // do something to d
}

for (String s: myArrayOfStrings) {
    // do something to s
}
Some useful static methods in \texttt{java.util}:

- Remember:
  
  \begin{verbatim}
  import java.util.*;
  \end{verbatim}

- Filling an array:
  
  \begin{verbatim}
  static void Arrays.fill(int[] a, int value)
  \end{verbatim}
  
  (Similar methods for other primitive types)

- Sorting an array:
  
  \begin{verbatim}
  static void Arrays.sort(int[] a)
  \end{verbatim}
  
  (Similar methods for other primitive types)

- Comparing two arrays for equal content:
  
  \begin{verbatim}
  static boolean Arrays.equals(int[] a1, int[] a2)
  \end{verbatim}
  
  (Similar methods for other primitive types)
Arrays of Strings

String[] words;
words = new String[5];
words[0] = "Hello";
words[4] = "Goodbye";
Two-dimensional arrays

- **Declaring a 2D array (example):**
  ```
  int[][] table;  // choose any name you like
  ```

- **Declaring and initializing a 2D array:**
  ```
  int[][] table = {{1, 2}, {3, 4}, {5, 6}};
  ```

- **Accessing the elements of a 2D array:**
  ```
  table[row][col]  // indexes are int expr's
  ```

  *NOT* `table[row, col]`!

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
2D Arrays in Storage

```c
int[][] grid;  // choose any name you like
grid = {{1, 2}, {3, 4}, {5, 6}};
```

**Stack**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td></td>
<td>int[][]</td>
</tr>
</tbody>
</table>

**Heap**

```
1  2
3  4
5  6
```

What is the type and value of `grid[1]`?
Explicit Storage Allocation for a 2D Array

```java
int[][] matrix = new int[3][2];
matrix[0][0] = 47;
matrix[1][1] = 63;
```

What happens if you write this?
```java
matrix = new int[2][5];
```
2D Arrays do not need to be rectangular!

```java
final int SIZE = 4;
int[][] pyramid = new int[SIZE][];
for (int i = 0; i < SIZE; i++)
    pyramid[i] = new int[i+1];
for (int i = 0; i < SIZE; i++)
    for (int j = 0; j < i + 1; j++)
        pyramid[i][j] = 10 * i + j;
```

What is the value of `pyramid[1][3]`?

What is the value of `pyramid[2].length`?
An array can have any number of dimensions

```java
int[][][] solid = new int[4][2][3];
```

```
a[0][0][0]  a[3][1][2]
```

```
a[3][0][0]  a[3][1][0]
```
Allocating storage for a subset of dimensions

```java
int[][][] solid = new int[4][2][3];
```

- Must allocate dimensions from left to right

```java
int[][][] solid = new int[4][2][ ]; // OK
int[][][] solid = new int[4][ ][ ]; // OK
int[][][] solid = new int[ ][ ][3]; // NOT OK
```
ArrayList

- A limitation of arrays:
  The dimension(s) of an array are fixed (once declared, they never change).

- What if you need a collection of objects that can grow indefinitely?

- ArrayList is a "container class" provided in java.util
  - Remember: import java.util.*;
  - Other container classes: LinkedList, HashSet, Stack, etc.

- An ArrayList does not have a fixed size.
  - It grows and shrinks according to its content.
  - Java manages the memory allocation automatically.
Some things you can do with an ArrayList

- Add an object at the end
- Insert an object in the middle
- Get the object at a given position
- Replace an object with a new object
- Remove an object
- Remove all the objects
- Find out whether an object is in the ArrayList
- Find the current size of the ArrayList
- etc.
Parameterized Types

- All the objects in an ArrayList have a common type
- The ArrayList is said to be "parameterized" by the type of object it contains
- Example: `ArrayList<String>` is an ArrayList that contains Strings
- You can use this "parameterized type" in a declaration:
  ```java
  ArrayList<String> planets;
  ```
  - Means: `planets` is an ArrayList that contains Strings
- Purpose of parameterized types: type safety
  - Compiler knows what type is inside the ArrayList
ArrayLists: A Limitation on Content

- The bad news: ArrayLists may not contain primitive types.
  - `ArrayList<int>` is *not OK* because `int` is primitive
  - `ArrayList<String>` is OK because `String` is not primitive

- The good news: Every primitive type has a nonprimitive counterpart called a "wrapper"

<table>
<thead>
<tr>
<th>Primitive Types:</th>
<th>Wrapper Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

- Example: Use `ArrayList<Integer>`, not `ArrayList<int>`
Creating an ArrayList

- Use a "constructor"

  ```java
  ArrayList<String> a = new ArrayList<String>();
  ```

- Optional: You can give the compiler a non-binding "estimate" about how big your ArrayList might grow

  ```java
  ArrayList<String> a = new ArrayList<String>(100);
  ```

- The ArrayList starts out empty (even if you gave it an estimated size.)
Adding items to an ArrayList

- Invoke the `add()` method, using "dot" notation
- By default, new items go at the end

```java
ArrayList<String> planets = new ArrayList<String>();
planets.add("Earth");
planets.add("Venus");
planets.add("Jupiter");
planets.add("Saturn");
```

```
<table>
<thead>
<tr>
<th>index</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Earth</td>
</tr>
<tr>
<td>1</td>
<td>Venus</td>
</tr>
<tr>
<td>2</td>
<td>Jupiter</td>
</tr>
<tr>
<td>3</td>
<td>Saturn</td>
</tr>
</tbody>
</table>
```
Accessing items in an ArrayList

- \texttt{get(n)} method returns the nth item in the ArrayList
- Indexes start with 0, like an array
- ArrayLists don't use bracket notation like \texttt{a[i]}
- If the type of \texttt{a} is \texttt{ArrayList<T>}, the type of \texttt{a.get(n)} is \texttt{T}

```java
planets.get(0); // Earth
planets.get(3); // Saturn
planets.get(4); // Error: subscript out of range
```

<table>
<thead>
<tr>
<th>planets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>Earth</td>
</tr>
<tr>
<td>Venus</td>
</tr>
<tr>
<td>Jupiter</td>
</tr>
<tr>
<td>Saturn</td>
</tr>
</tbody>
</table>
Modifying an ArrayList

- Adding items "in the middle": use `add(index, item)` (causes renumbering)

- Removing items: use `remove(index)` (causes renumbering)

- Setting (replacing) a specific item: use `set(index, item)`

```java
planets.add(2, "Mars");
planets.remove(3);
planets.set(3, "Neptune");
```
More ArrayList Methods

- `a.size( )`
  - Returns the current size of `a` (number of items)

- `a.clear( )`
  - Removes all items in the ArrayList `a`

- `a.isEmpty( )`
  - Returns true if ArrayList `a` is empty, otherwise false

- `a.indexOf("Neptune")`
  - Returns the index of the first matching item, based on the `equals` method
  - Returns -1 if no matching item is found in `a`
Things to remember about ArrayLists

- They can grow and shrink
- Their items are always numbered compactly, starting with zero
- The index of a given item can change
- Use methods: `a.add()`, `a.get()`, `a.delete()`, etc.
- Don't use array notation: `a[i]` doesn't work