Week 5: Arrays

4009 DATA STRUCTURES & ALGORITHM (DSA)

Learning Outcomes

- ❖Understand and implement Arrays in Java
- ❖Able to create, access and understand Array elements and errors
- ❖Discuss the out of bounds error in Java
- ❖Understand and implement one or more of Array sorting algorithms
- ❖Differentiate between Array sorting algorithms
- ❖Understand the concept of multi-dimensional arrays

Content Overview

- Arrays
- Java.util methods for Arrays and Random Numbers
- Creating an Array (declare an array)
- Accessing Array elements
- Out bound errors
- Array Initialisation
- Using Arrays – Storing Game Entries in an Array
  - Adding an Entry
  - Removing an Entry
- An example of using Pseudo-Random Numbers
- The Insertion-Sort Algorithm
- Other Array sorting algorithms
- Multi-dimensional arrays (two-dimensional arrays and Positional Games)

Data Structure Trade-offs

Requirement that is enforced:

- Arrays store data sequentially in memory.

Data structures	Advantages	Disadvantages
Array	Quick insertion, very fast access to index known	Slow search, slow deletion, fixed size
Ordered Array	Quicker Search than unsorted array	Slow insertion and deletion, fixed size
Stack	Provide Last-in First-out access	Slow access to other items
Queue	Provide First-in First out access	Slow access to other items
Linked List	Quick Insertion, Quick deletion	Slow search
Binary Tree	Quick search, insertion, deletion (if tree balanced)	Deletion algorithm is complex
Hash Table	Very fast access if key known, fast insertion	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion, deletion, access to large item	Slow access to other items
Graphs	Model real world situations	Some algorithms are slow and complex

Arrays

- The array is the most commonly used data storage structure; it’s built into most programming languages.
- A common programming task is to keep track of a numbered group of related objects.
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- An array is a collection of items stored at contiguous memory locations.
- For example, we want a video game to keep track of the top ten scores for that game. Rather than use ten different variables of this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.
- Therefore, an array is a numbered collection of variables all of the same type. Each variable or cell, in an array has an index which uniquely refer to the value stored in that cell.

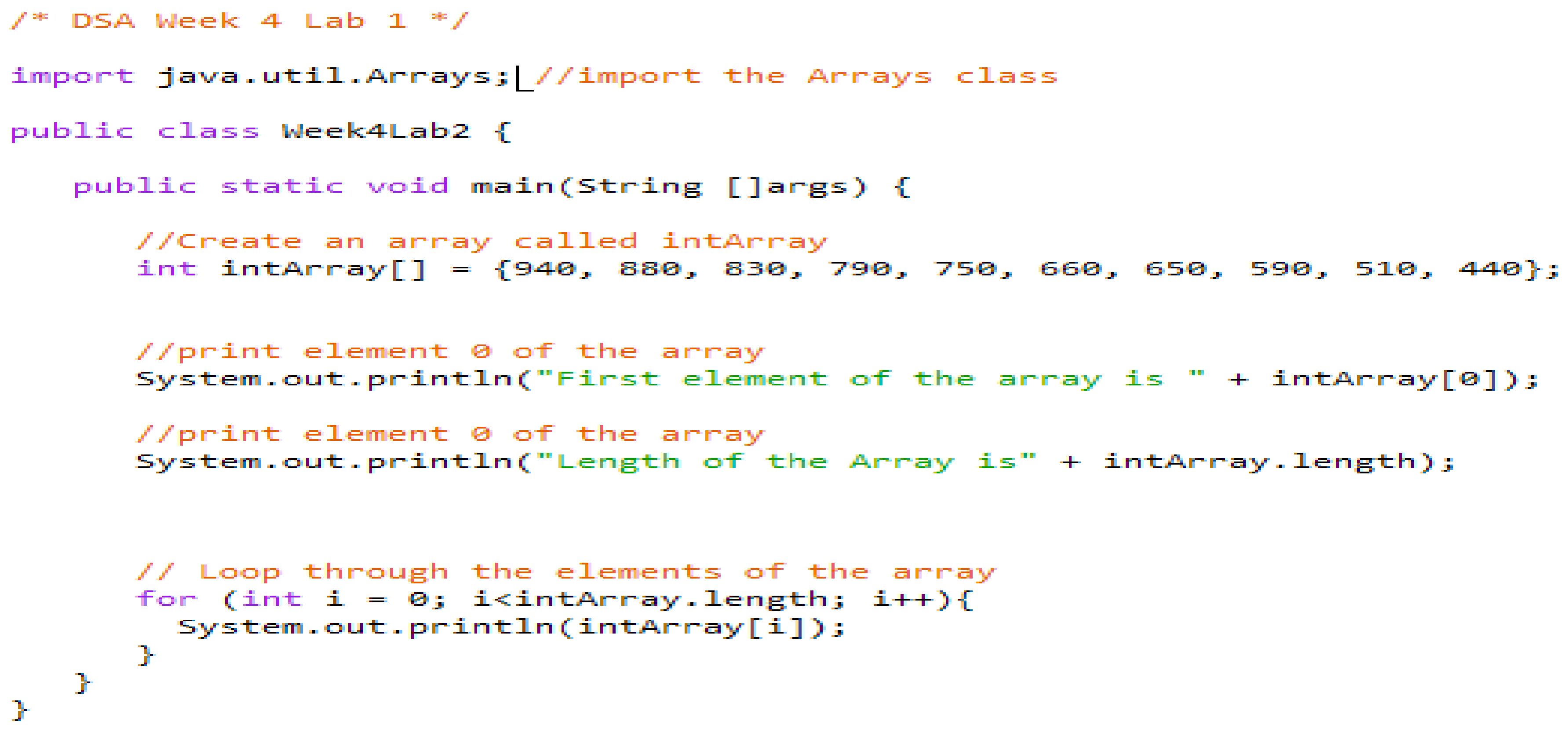
High Scores	940	880	830	790	750	660	650	590	510	440
	0	1	2	3	4	5	6	7	8	9

indices

Arrays

High Scores	940	880	830	790	750	660	650	590	510	440
	0	1	2	3	4	5	6	7	8	9

indices



```
/* DSA Week 4 Lab 1 */

import java.util.Arrays; //import the Arrays class

public class Week4Lab2 {

    public static void main(String []args) {

        //Create an array called intArray
        int intArray[] = {940, 880, 830, 790, 750, 660, 650, 590, 510, 440};


        //print element 0 of the array
        System.out.println("First element of the array is " + intArray[0]);

        //print element 0 of the array
        System.out.println("Length of the Array is" + intArray.length);



        // Loop through the elements of the array
        for (int i = 0; i<intArray.length; i++){
          System.out.println(intArray[i]);
        }
    }
}
```

Java.util methods for Arrays and Random Numbers

- Java provides a number of built-in methods for performing common tasks on arrays.
- This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.
- NOTE: before creating the class, you must import java.util.Arrays; class in order to use and manipulate Arrays in Java.

Creating an Array (declare an array)

Accordingly, you must use the new operator to create an array:

`dataType[] arrayName = new dataType[arraySize];`

```
int[] intArray; // defines a reference to an array
intArray = new int[10]; // creates the array, and // sets intArray to refer to it
```

Or you can use the equivalent single-statement approach:

The [] operator is the sign to the compiler we're naming an array object and not an ordinary variable. You can also use an alternative syntax for this operator, placing it after the name instead of the type:

```
int intArray[] = new int[10]; // alternative syntax
```

Note: placing the [] after the int makes it clear that the [] is part of the type, not the name.

Because an array is an object, its name—intArray in the preceding code—is a reference to an array; it's not the array itself. The array is stored at an address elsewhere in memory, and intArray holds only this address.

Arrays have a length field, which you can use to find the size (the number of elements) of an array:

```
int arrayLength = intArray.length; // find array size
```

As in most programming languages, you can't change the size of an array after it's been created.

Accessing Array elements

Array elements are accessed using an index number in square brackets. This is similar to how other languages work:

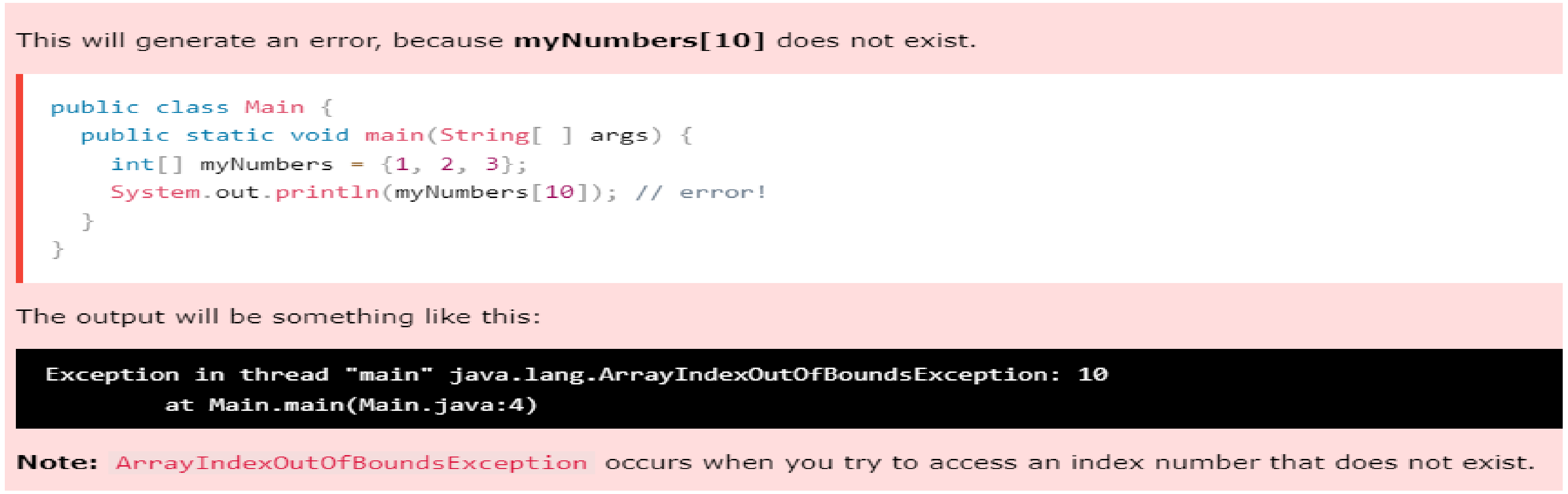
```
intArray[7] = 660; // insert 660 into the sixth cell

System.out.println(intArray[0]); //prints the element[0] in the array [940]

System.out.println(intArray.length); //prints the length of the array (10)

for (int i = 0; i < intArray.length; i++) {

  System.out.println(intArray[i]);

} //prints all the array elements
```

Remember that in Java, the first element is numbered 0, so that the indices in an array of 10 elements run from 0 to 9.

If you use an index that’s less than 0 or greater than the size of the array less 1, you’ll get the Array Index Out of Bounds runtime error.

Out bound errors

- A dangerous mistake to attempt to index into an array intArray using a number outside of the range from 0 to intArray.length & -1. Such a reference is said to be **out of bounds.**
- Out of bounds references have been exploited numerous times by hackers using a method called the buffer overflow attack to compromise the security of computer systems written in languages other than Java.
- Java has a safety feature to check and avoid out bound errors in Arrays called ArrayIndexOutOfBoundsException.


This will generate an error, because **myNumbers[10]** does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

Note: `ArrayIndexOutOfBoundsException` occurs when you try to access an index number that does not exist.

Array Initialisation

- Unless you specify, an array of integers is automatically initialized to 0 when it's created. Create an array of objects like this:

```
autoData[] carArray = new autoData[4000];
```

- Until the array elements are given explicit values, they contain the special null object. If you attempt to access an array element that contains null, you'll get the runtime error Null Pointer Assignment. The moral is to make sure you assign something to an element before attempting to access it.

- You can initialize an array of a primitive type to something besides 0 using this syntax:

```
Element_type[] array_name = {init_value0, init_value1, …., init_valueN-1}
int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
```

- This single statement takes the place of both the reference declaration and the use of new to create the array. The numbers within the curly brackets are called the initialization list. The size of the array is determined by the number of values in this list.

Using Arrays – Storing Game Entries in an Array (1/2)

The first application we study is storing a sequence of high score entries for a video game in an array. This is representative of many applications in which a sequence of objects must be stored. We could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data-structuring concepts.

To begin, we consider what information to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we identify as score. Another useful thing to include is the name of the person earning this score, which we identify as name.

A Java class, GameEntry, representing a game entry, is given in Code Fragment 3.1.

```
public class GameEntry {
  private String name;                    // name of the person earning this score
  private int score;                      // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName() { return name; }
  /** Returns the score field. */
  public int getScore() { return score; }
  /** Returns a string representation of this entry. */
  public String toString() {
    return "(" + name + ", " + score + ")";
  }
}
```

Code Fragment 3.1: Java code for a simple GameEntry class. Note that we include methods for returning the name and score for a game entry object, as well as a method for returning a string representation of this entry.

Using Arrays – Storing Game Entries in an Array (2/2)

A Class for High Scores To maintain a sequence of high scores, we develop a class named Scoreboard. A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest "high score" on the board. The length of the desired scoreboard may depend on the game, perhaps 10, 50, or 500. Since that limit may vary, we allow it to be specified as a parameter to our Scoreboard constructor.

Internally, we will use an array named board to manage the GameEntry instances that represent the high scores. The array is allocated with the specified maximum capacity, but all entries are initially null. As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the array.

We illustrate a typical state of the data structure in Figure 3.1, and give Java code to construct such a data structure in Code Fragment 3.2.

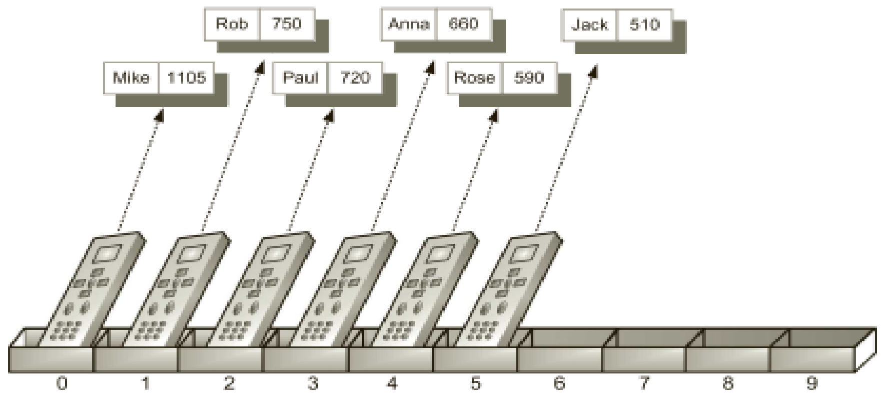


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

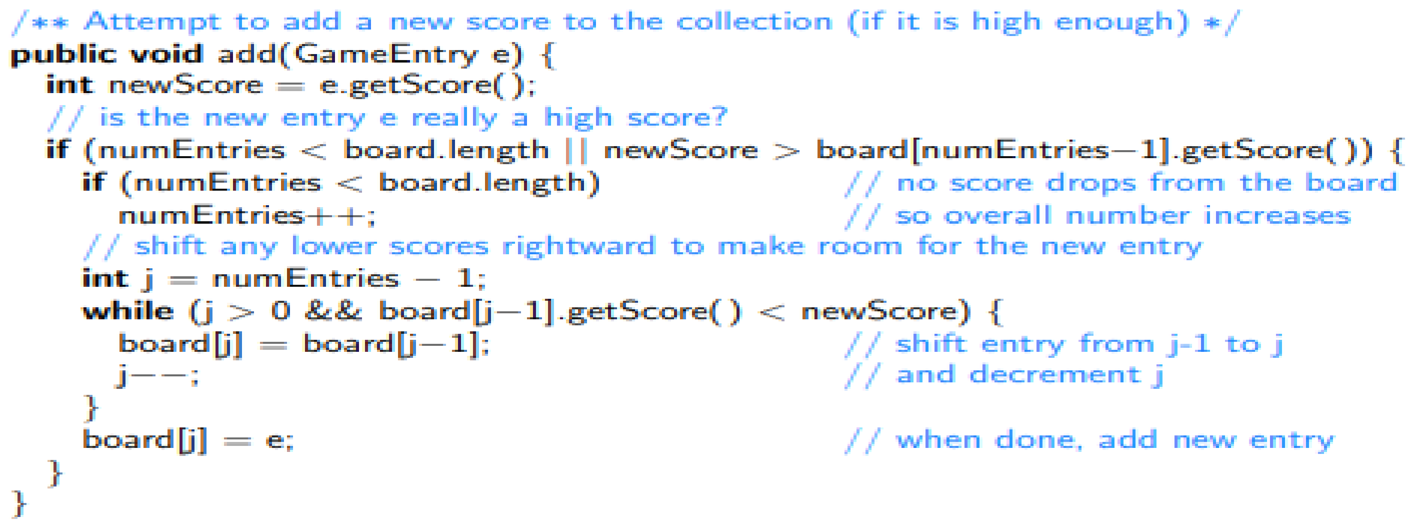
```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;                  // number of actual entries
  private GameEntry[ ] board;                  // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity for storing entries. */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
  // more methods will go here
}
```

Code Fragment 3.2: The beginning of a Scoreboard class for maintaining a set of scores as GameEntry objects. (Completed in Code Fragments 3.3 and 3.4.)

Adding an Entry (1/3)

One of the most common updates we might want to make to a Scoreboard is to add a new entry. Keep in mind that not every entry will necessarily qualify as a high score. If the board is not yet full, any new entry will be retained. Once the board is full, a new entry is only retained if it is strictly better than one of the other scores, in particular, the last entry of the scoreboard, which is the lowest of the high scores.

Code Fragment 3.3 provides an implementation of an update method for the Scoreboard class that considers the addition of a new game entry.



```java
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
  int newScore = e.getScore();
  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
    if (numEntries < board.length)                  // no score drops from the board
      numEntries++;                                 // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries - 1;
    while (j > 0 && board[j-1].getScore() < newScore) {
      board[j] = board[j-1];                        // shift entry from j-1 to j
      j--;                                          // and decrement j
    }
    board[j] = e;                                   // when done, add new entry
  }
}
```

Code Fragment 3.3: Java code for inserting a GameEntry object into a Scoreboard.

Adding an Entry (2/3)

When a new score is considered, the first goal is to determine whether it qualifies as a high score. This will be the case (see line 13) if the scoreboard is below its capacity, or if the new score is strictly higher than the lowest score on the board. Once it has been determined that a new entry should be kept, there are two remaining tasks: (1) properly update the number of entries, and (2) place the new entry in the appropriate location, shifting entries with inferior scores as needed.

The first of these tasks is easily handled at lines 14 and 15, as the total number of entries can only be increased if the board is not yet at full capacity. (When full, the addition of a new entry will be counteracted by the removal of the entry with lowest score.) The placement of the new entry is implemented by lines 17–22. Index j is initially set to numEntries − 1, which is the index at which the last GameEntry will reside after completing the operation. Either j is the correct index for the newest entry, or one or more immediately before it will have lesser scores. The while loop checks the compound condition, shifting entries rightward and decrementing j, as long as there is another entry at index j −1 with a score less than the new score.

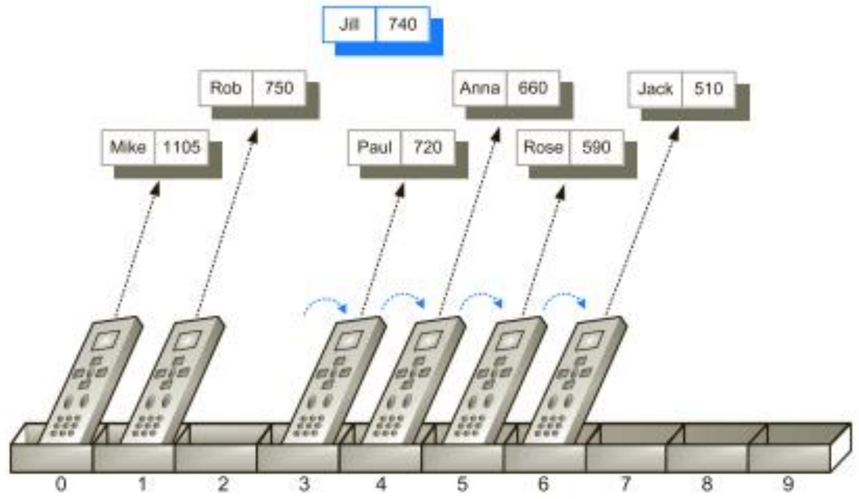


Figure 3.2: Preparing to add Jill's GameEntry object to the board array. In order to make room for the new reference, we have to shift any references to game entries with smaller scores than the new one to the right by one cell.

Adding an Entry (3/3)

Figure 3.2 shows an example of the process, just after the shifting of existing entries, but before adding the new entry. When the loop completes, j will be the correct index for the new entry. Figure 3.3 shows the result of a complete operation, after the assignment of board[j] = e, accomplished by line 22 of the code.

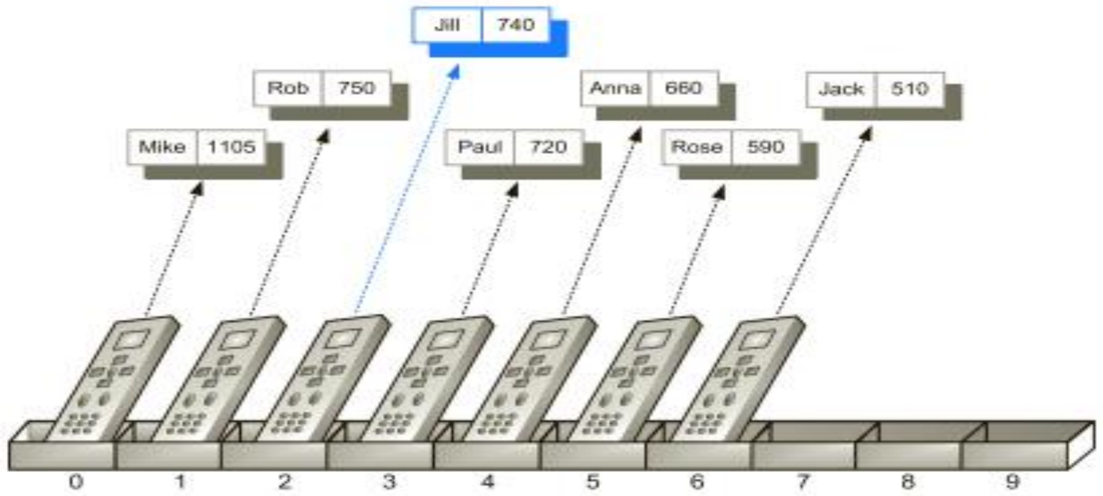


Figure 3.3: Adding a reference to Jill's GameEntry object to the board array. The reference can now be inserted at index 2, since we have shifted all references to GameEntry objects with scores less than the new one to the right.

Removing an Entry (1/2)

Suppose some hot shot plays our video game and gets his or her name on our high score list, but we later learn that cheating occurred. In this case, we might want to have a method that lets us remove a game entry from the list of high scores.

Therefore, let us consider how we might remove a reference to a GameEntry object from a Scoreboard. We choose to add a method to the Scoreboard class, with signature remove(i), where i designates the current index of the entry that should be removed and returned. When a score is removed, any lower scores will be shifted upward, to fill in for the removed entry.

If index i is outside the range of current entries, the method will throw an IndexOutOfBoundsException. Our implementation for remove will involve a loop for shifting entries, much like our algorithm for addition, but in reverse. To remove the reference to the object at index i, we start at index i and move all the references at indices higher than i one cell to the left. (See Figure 3.4.)

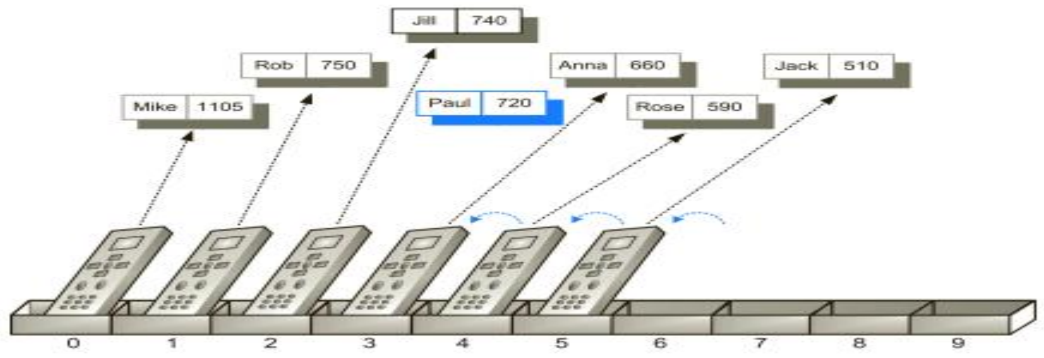
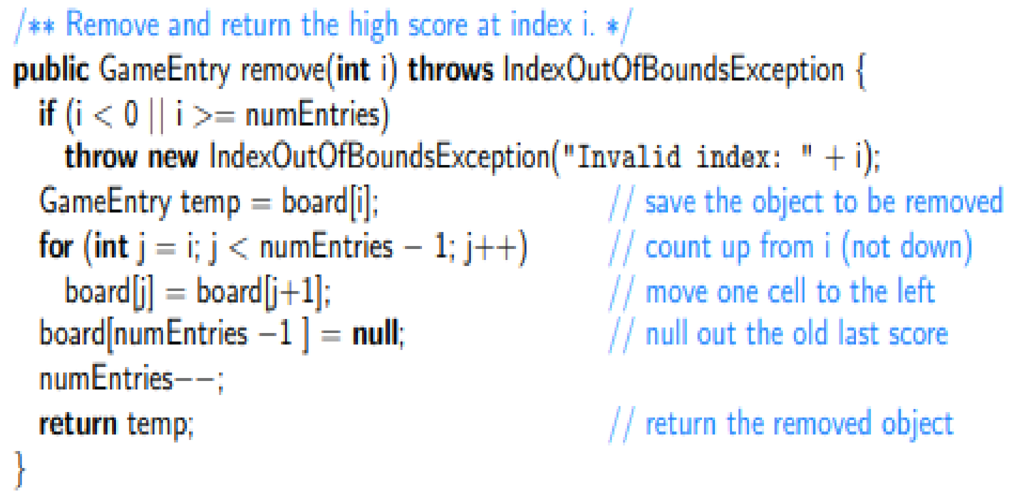


Figure 3.4: An illustration of the removal of Paul's score from index 3 of an array storing references to GameEntry objects.

Removing an Entry (1/2)

Our implementation of the remove method for the Scoreboard class is given in Code Fragment 3.4. The details for doing the remove operation contain a few subtle points. The first is that, in order to remove and return the game entry (let’s call it e) at index i in our array, we must first save e in a temporary variable. We will use this variable to return e when we are done removing it.

The second subtle point is that, in moving references higher than i one cell to the left, we don’t go all the way to the end of the array. First, we base our loop on the number of current entries, not the capacity of the array, because there is no reason for “shifting” a series of null references that may be at the end of the array. We also carefully define the loop condition, j < numEntries − 1, so that the last iteration of the loop assigns board[numEntries−2] = board[numEntries−1]. There is no entry to shift into cell board[numEntries−1], so we return that cell to null just after the loop. We conclude by returning a reference to the removed entry (which no longer has any reference pointing to it within the board array).



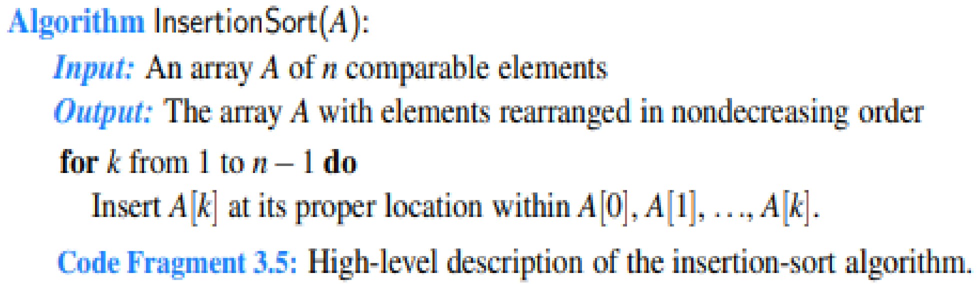
```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                    // save the object to be removed
  for (int j = i; j < numEntries − 1; j++)      // count up from i (not down)
    board[j] = board[j+1];                      // move one cell to the left
  board[numEntries −1 ] = null;                 // null out the old last score
  numEntries−−;
  return temp;                                  // return the removed object
}
```

Code Fragment 3.4: Java code for performing the Scoreboard.remove operation.

The Insertion-Sort Algorithm

The insertion-sort is an algorithm that proceeds by considering one element at a time, placing the element in the correct order relative to those before it. The first element in the array, which is trivially sorted by itself. When considering the next element in the array, if it is smaller than the first, they swap them. Next, the third element in the array is considered, swapping it leftward until it is in its proper order relative to the first two elements. This manner continues until the whole array is sorted.

We can express the insertion-sort algorithm in pseudocode, as shown in Code Fragment 3.5.



```
Algorithm InsertionSort($A$):
  Input: An array $A$ of $n$ comparable elements
  Output: The array $A$ with elements rearranged in nondecreasing order
  for $k$ from 1 to $n-1$ do
    Insert $A[k]$ at its proper location within $A[0], A[1], \ldots, A[k]$.
```

Code Fragment 3.5: High-level description of the insertion-sort algorithm.

A Java implementation of insertion-sort in Code Fragment 3.6, using an outer loop to consider each element in turn, and an inner loop that moves a newly considered element to its proper location relative to the (sorted) subarray of elements that are to its left. We illustrate an example run of the insertion-sort algorithm in Figure 3.5.

The Insertion-Sort Algorithm

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {                       // begin with second character
    char cur = data[k];                               // time to insert cur=data[k]
    int j = k;                                        // find correct index j for cur
    while (j > 0 && data[j−1] > cur) {                // thus, data[j-1] must go after cur
      data[j] = data[j−1];                            // slide data[j-1] rightward
      j−−;                                            // and consider previous j for cur
    }
    data[j] = cur;                                    // this is the proper place for cur
  }
}
```

Code Fragment 3.6: Java code for performing insertion-sort on a character array.

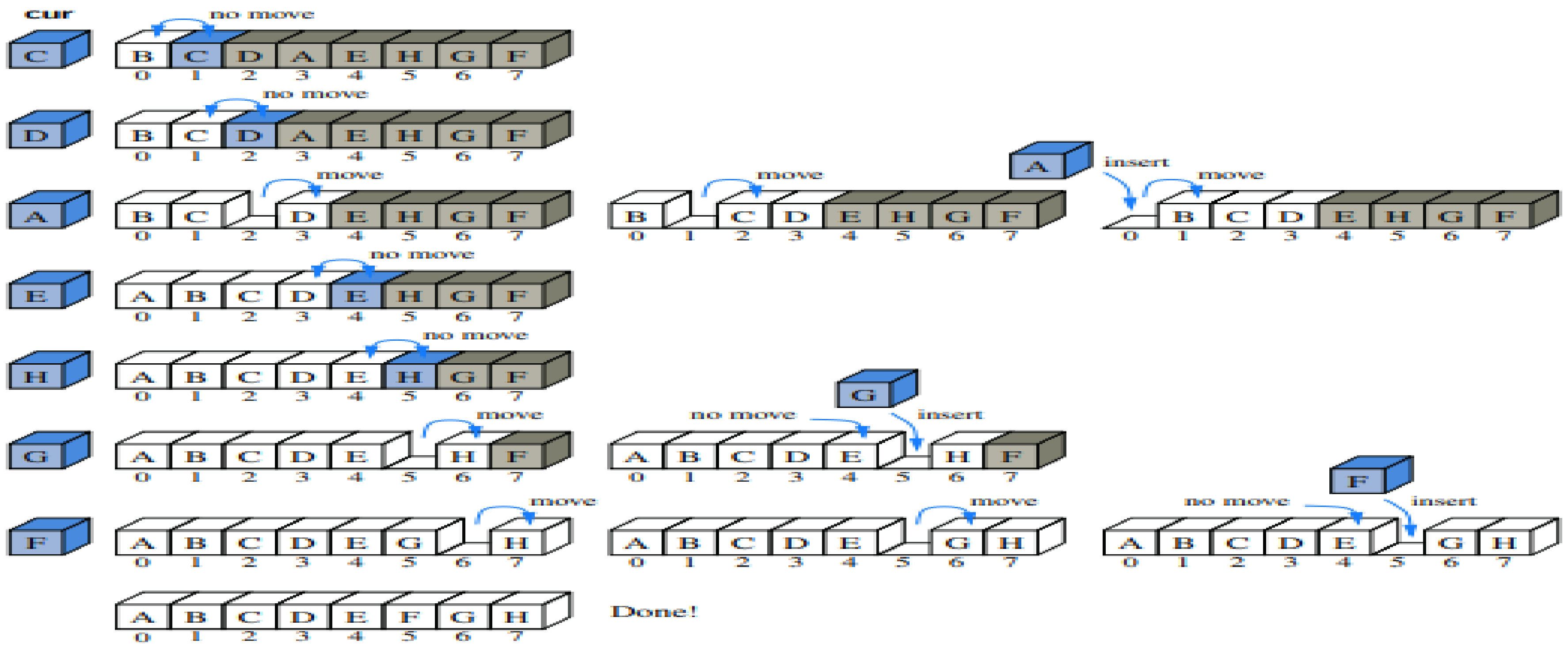


Figure 3.5: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.

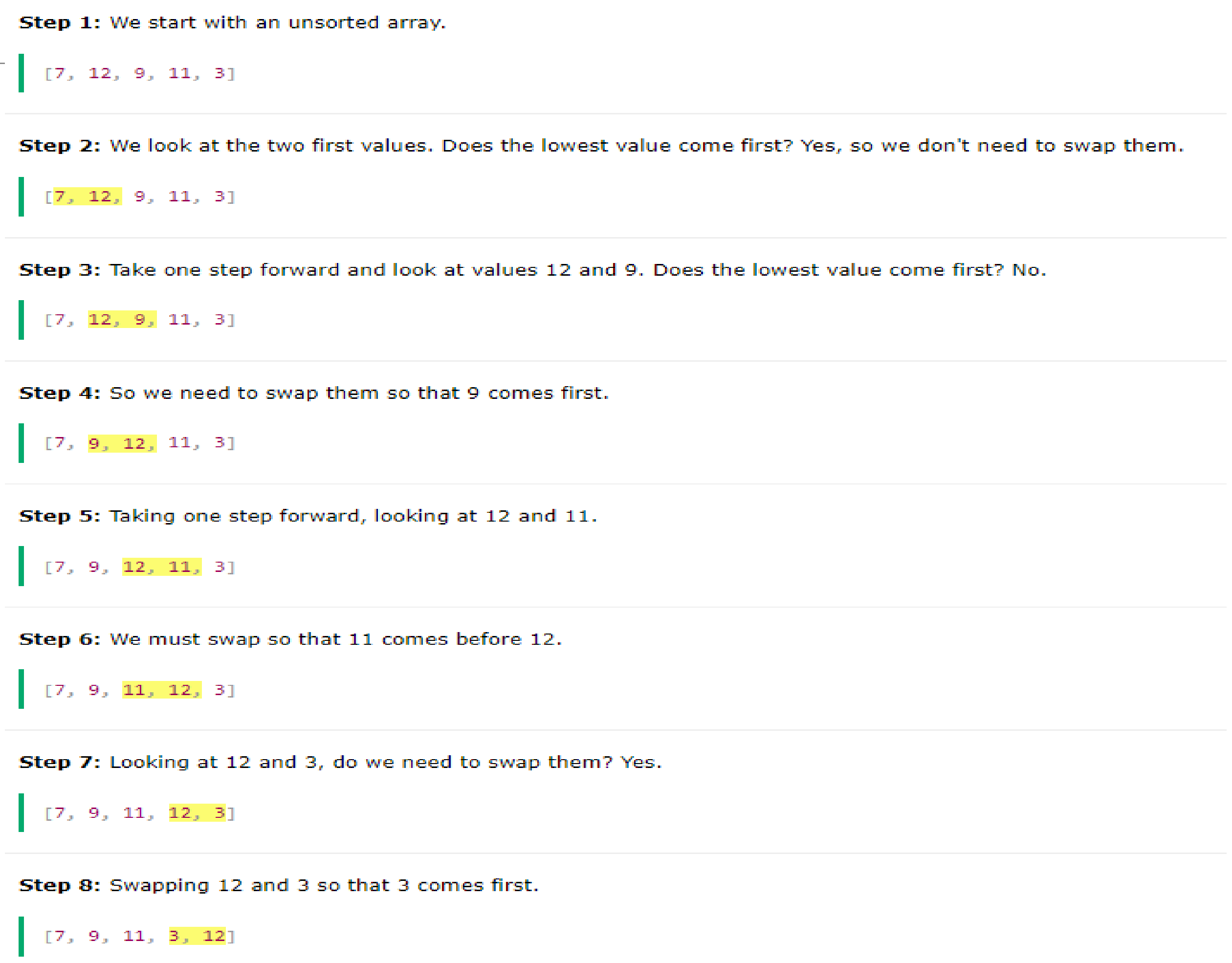
Other Array sorting algorithms (1/3)

This section we will only discuss the theory of sorting algorithms, no implementation code in Java. This will be fully learnt in other courses.

Bubble Sort - The word 'Bubble' comes from how this algorithm works; it makes the highest values 'bubble up'. Bubble sort works:

1. Go through the array, one value at a time.
2. For each value, compare the value with the next value.
3. If the value is higher than the next one, swap the values so that the highest value comes last.
4. Go through the array as many times as there are values in the array.

(W3Schools, 2025)



Step 1: We start with an unsorted array.

```
[7, 12, 9, 11, 3]
```

Step 2: We look at the two first values. Does the lowest value come first? Yes, so we don't need to swap them.

```
[7, 12, 9, 11, 3]
```

Step 3: Take one step forward and look at values 12 and 9. Does the lowest value come first? No.

```
[7, 12, 9, 11, 3]
```

Step 4: So we need to swap them so that 9 comes first.

```
[7, 9, 12, 11, 3]
```

Step 5: Taking one step forward, looking at 12 and 11.

```
[7, 9, 12, 11, 3]
```

Step 6: We must swap so that 11 comes before 12.

```
[7, 9, 11, 12, 3]
```

Step 7: Looking at 12 and 3, do we need to swap them? Yes.

```
[7, 9, 11, 12, 3]
```

Step 8: Swapping 12 and 3 so that 3 comes first.

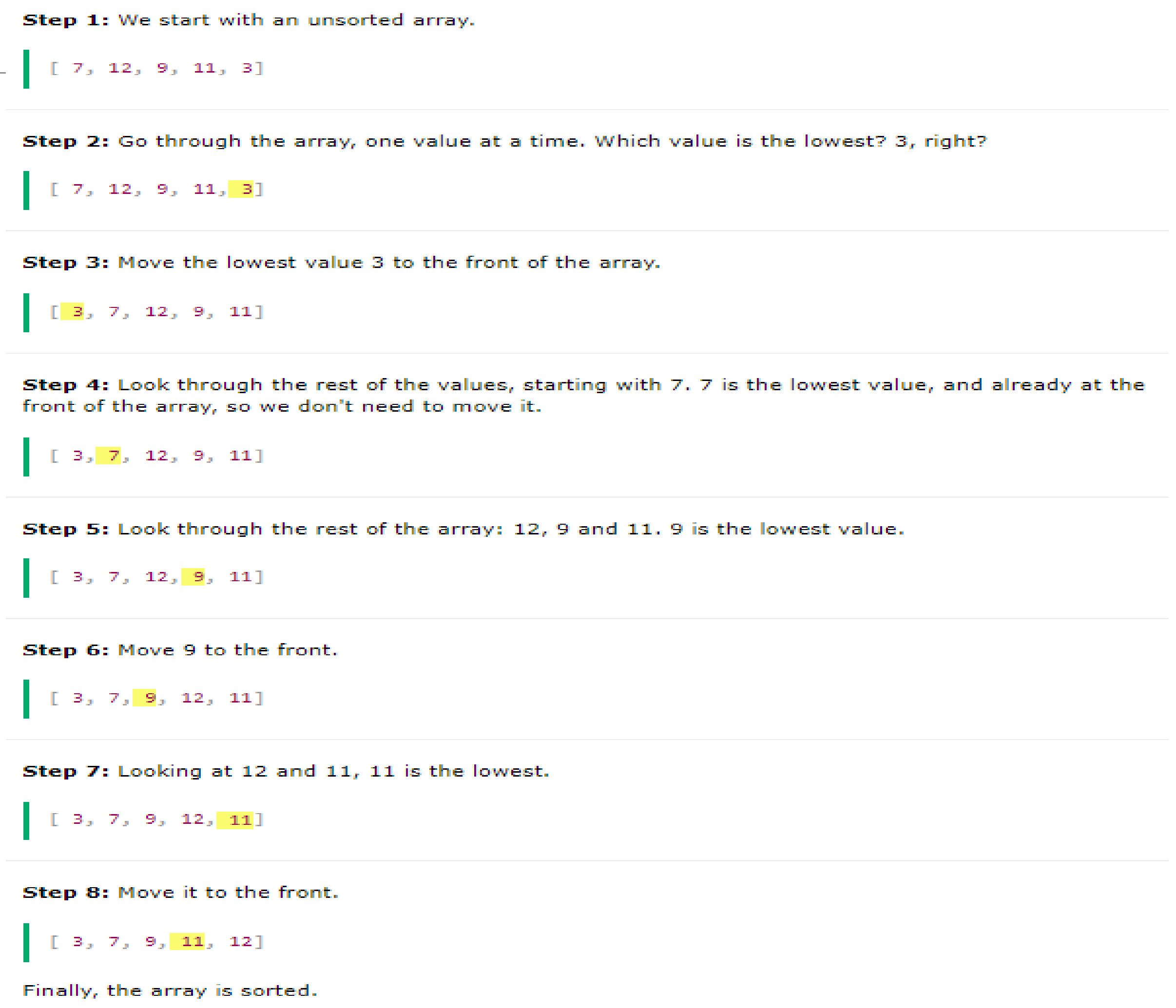
```
[7, 9, 11, 3, 12]
```

Other Array sorting algorithms (2/3)

Selection Sort - The algorithm looks through the array again and again, moving the next lowest values to the front, until the array is sorted. How selection sort works:

1. Go through the array to find the lowest value.
2. Move the lowest value to the front of the unsorted part of the array.
3. Go through the array again as many times as there are values in the array.

(W3Schools, 2025)



Step 1: We start with an unsorted array.

```
[ 7, 12, 9, 11, 3]
```

Step 2: Go through the array, one value at a time. Which value is the lowest? 3, right?

```
[ 7, 12, 9, 11, 3]
```

Step 3: Move the lowest value 3 to the front of the array.

```
[ 3, 7, 12, 9, 11]
```

Step 4: Look through the rest of the values, starting with 7. 7 is the lowest value, and already at the front of the array, so we don't need to move it.

```
[ 3, 7, 12, 9, 11]
```

Step 5: Look through the rest of the array: 12, 9 and 11. 9 is the lowest value.

```
[ 3, 7, 12, 9, 11]
```

Step 6: Move 9 to the front.

```
[ 3, 7, 9, 12, 11]
```

Step 7: Looking at 12 and 11, 11 is the lowest.

```
[ 3, 7, 9, 12, 11]
```

Step 8: Move it to the front.

```
[ 3, 7, 9, 11, 12]
```

Finally, the array is sorted.

Other Array sorting algorithms (3/3)

Quick sort - is one of the fastest sorting algorithms.

The Quicksort algorithm takes an array of values, chooses one of the values as the 'pivot' element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it. Quick sort algorithm can be described like this:

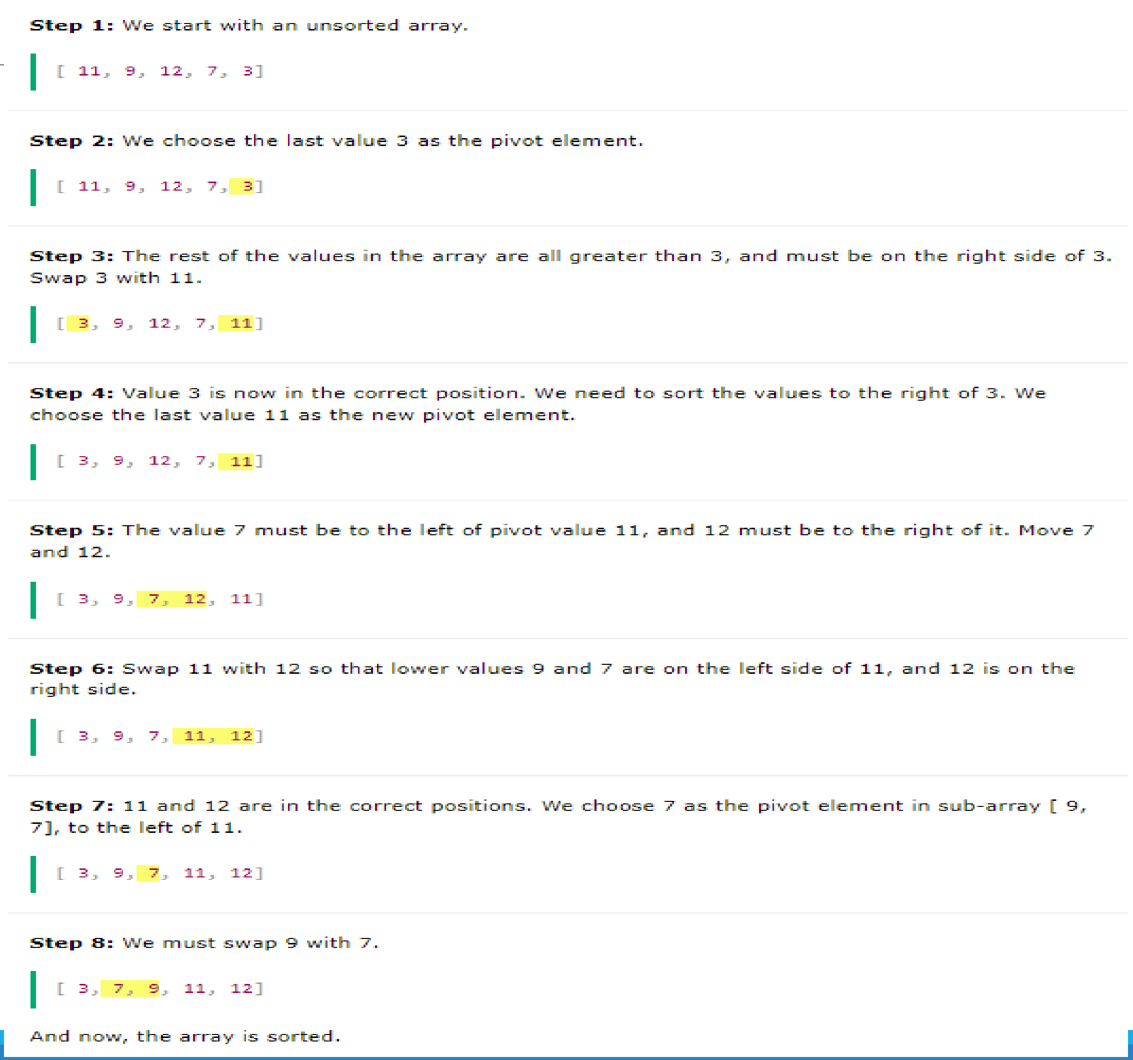
Choose a value in the array to be the pivot element.

Order the rest of the array so that lower values than the pivot element are on the left, and higher values are on the right.

Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and higher values.

Do the same operations (recursively) for the sub-arrays on the left and right side of the pivot element.

(W3Schools, 2025)



Step 1: We start with an unsorted array.

```
[ 11, 9, 12, 7, 3]
```

Step 2: We choose the last value 3 as the pivot element.

```
[ 11, 9, 12, 7, 3]
```

Step 3: The rest of the values in the array are all greater than 3, and must be on the right side of 3. Swap 3 with 11.

```
[ 3, 9, 12, 7, 11]
```

Step 4: Value 3 is now in the correct position. We need to sort the values to the right of 3. We choose the last value 11 as the new pivot element.

```
[ 3, 9, 12, 7, 11]
```

Step 5: The value 7 must be to the left of pivot value 11, and 12 must be to the right of it. Move 7 and 12.

```
[ 3, 9, 7, 12, 11]
```

Step 6: Swap 11 with 12 so that lower values 9 and 7 are on the left side of 11, and 12 is on the right side.

```
[ 3, 9, 7, 11, 12]
```

Step 7: 11 and 12 are in the correct positions. We choose 7 as the pivot element in sub-array [9, 7], to the left of 11.

```
[ 3, 9, 7, 11, 12]
```

Step 8: We must swap 9 with 7.

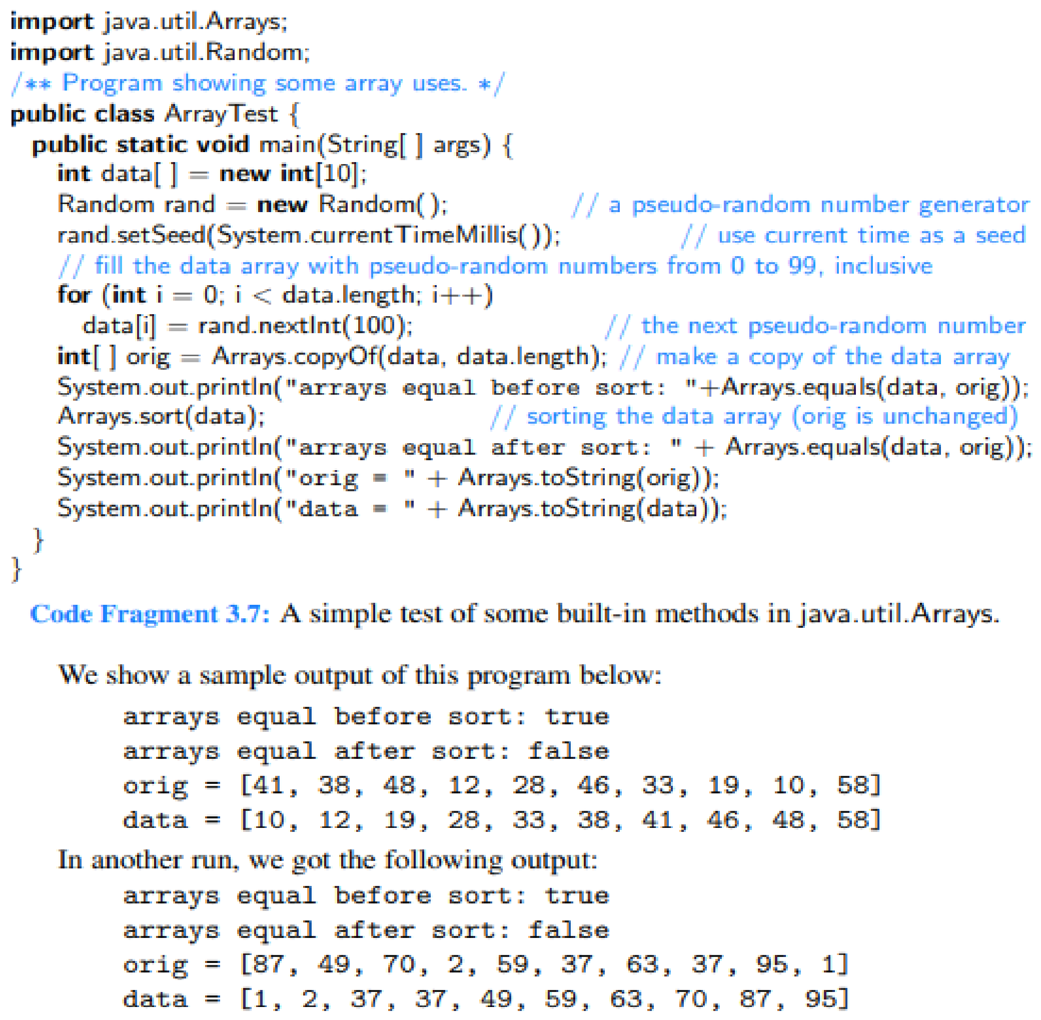
```
[ 3, 7, 9, 11, 12]
```

And now, the array is sorted.

References

W3Schools. (2025, February 27). *DSA Arrays*. Retrieved from W3Schools: https://www.w3schools.com/dsa/dsa_data_arrays.php

An example of using Pseudo-Random Numbers



```
import java.util.Arrays;
import java.util.Random;
/** Program showing some array uses. */
public class ArrayTest {
  public static void main(String[ ] args) {
    int data[ ] = new int[10];
    Random rand = new Random( );              // a pseudo-random number generator
    rand.setSeed(System.currentTimeMillis( ));            // use current time as a seed
    // fill the data array with pseudo-random numbers from 0 to 99, inclusive
    for (int i = 0; i < data.length; i++)
      data[i] = rand.nextInt(100);                 // the next pseudo-random number
    int[ ] orig = Arrays.copyOf(data, data.length); // make a copy of the data array
    System.out.println("arrays equal before sort: "+Arrays.equals(data, orig));
    Arrays.sort(data);                    // sorting the data array (orig is unchanged)
    System.out.println("arrays equal after sort: " + Arrays.equals(data, orig));
    System.out.println("orig = " + Arrays.toString(orig));
    System.out.println("data = " + Arrays.toString(data));
  }
}
```

Code Fragment 3.7: A simple test of some built-in methods in java.util.Arrays.

We show a sample output of this program below:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
data = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]
```

In another run, we got the following output:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [87, 49, 70, 2, 59, 37, 63, 37, 95, 1]
data = [1, 2, 37, 37, 49, 59, 63, 70, 87, 95]
```

By using a pseudorandom number generator to determine program values, we get a different input to our program each time we run it. This feature is, in fact, what makes pseudorandom number generators useful for testing code, particularly when dealing with arrays.

Even so, we should not use random test runs as a replacement for reasoning about our code, as we might miss important special cases in test runs. Note, for example, that there is a slight chance that the orig and data arrays will be equal even after data is sorted, namely, if orig is already ordered. The odds of this occurring are less than 1 in 3 million, so it’s unlikely to happen during even a few thousand test runs; however, we need to reason that this is possible

Multi-dimensional arrays (two-dimensional arrays and Positional Games)

- Many computer games, be they strategy games, simulation games, or first-person conflict games, involve objects that reside in a two-dimensional space.
- Software for such positional games needs a way of representing objects in a two-dimensional space. A natural way to do this is with a two-dimensional array, where we use two indices, say i and j, to refer to the cells in the array.
- The first index usually refers to a row number and the second to a column number. Given such an array, we can maintain two-dimensional game boards and perform other kinds of computations involving data stored in rows and columns.
- Arrays in Java are one-dimensional; we use a single index to access each cell of an array. Nevertheless, there is a way we can define two-dimensional arrays in Java—we can create a two-dimensional array as an array of arrays. That is, we can define a two-dimensional array to be an array with each of its cells being another array. Such a two-dimensional array is sometimes also called a matrix.

Multi-dimensional arrays (two-dimensional arrays and Positional Games)

In Java, we may declare a two-dimensional array as follows:

int[ ][ ] data = new int[8][10];

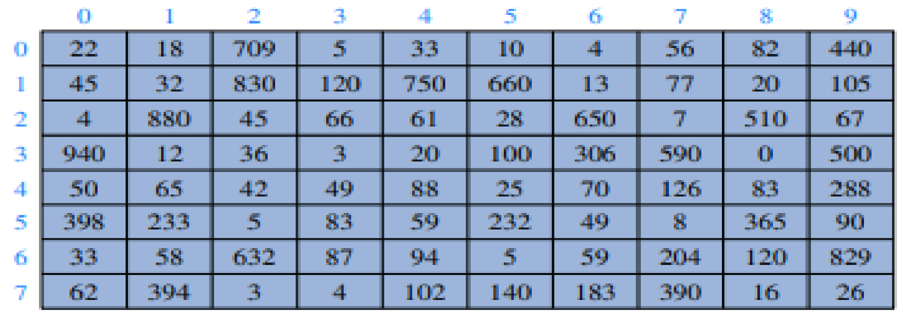
This statement creates a two-dimensional “array of arrays,” data, which is 8×10, having 8 rows and 10 columns. That is, data is an array of length 8 such that each element of data is an array of length 10 of integers. (See Figure 3.7.)

The following would then be valid uses of array data and int variables i, j, and k:

data[i][i+1] = data[i][i] + 3;

j = data.length; // j is 8

k = data[4].length; // k is 10



	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 3.7: Illustration of a two-dimensional integer array, data, which has 8 rows and 10 columns. The value of data[3][5] is 100 and the value of data[6][2] is 632.

Two-dimensional arrays have many applications to numerical analysis. Rather than going into the details of such applications, however, we explore an application of two-dimensional arrays for implementing a simple positional game.

Topic next week

Week 6 Linked List

DSA Week 5 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
  - Refer to the print out, answer all week 5 questions.
  - Refer to the print out, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 1, test 2, Major Assignment, Mid Semester Exam & Final Exam.