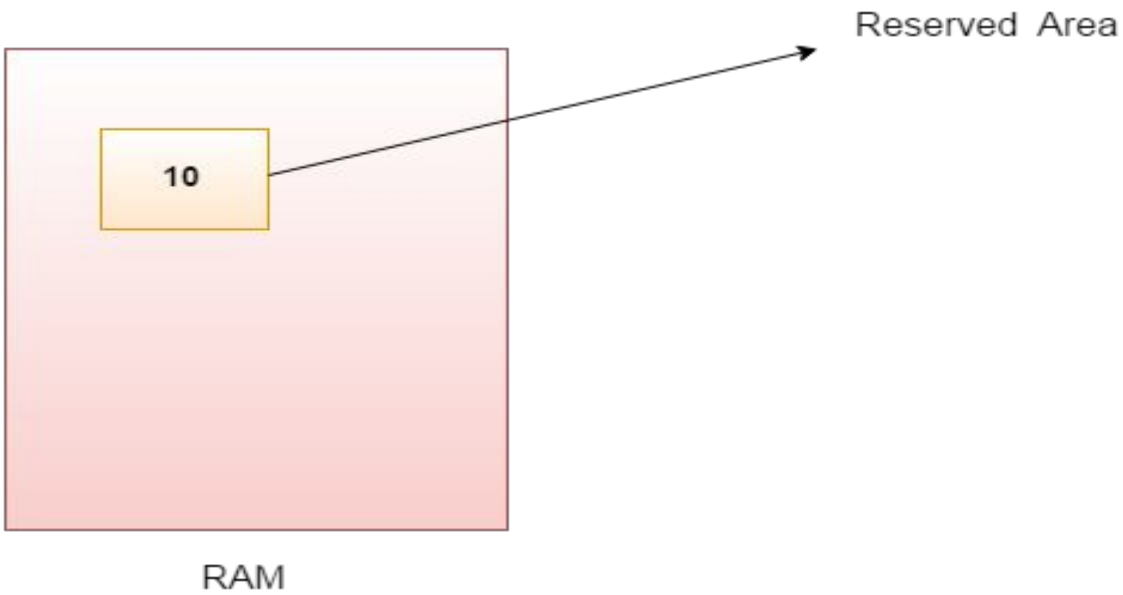
Lecture 4

Variable

Variable is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.



Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable which is declared inside the method is called local variable.

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable . It is not declared as static.

3) Static variable

A variable that is declared as static is called static variable. It cannot be local.
We will have detailed learning of these variables in next chapters.

Array Variables

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type.

Syntax

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

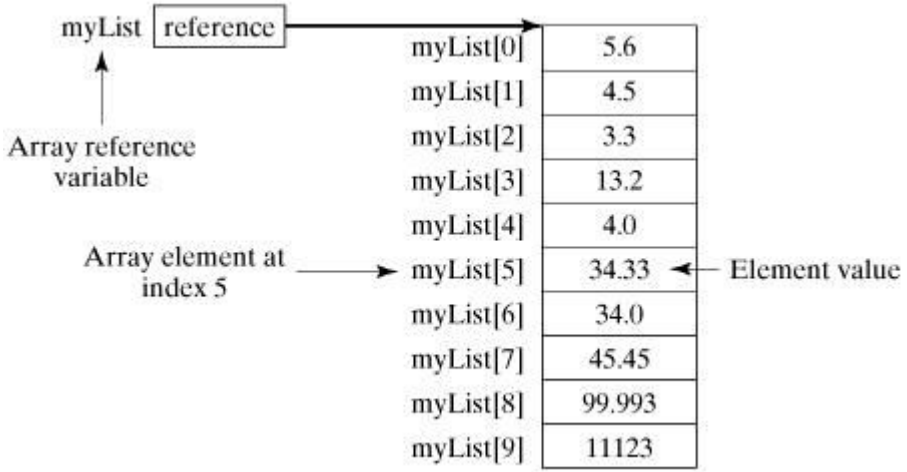
Create Array

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

Example

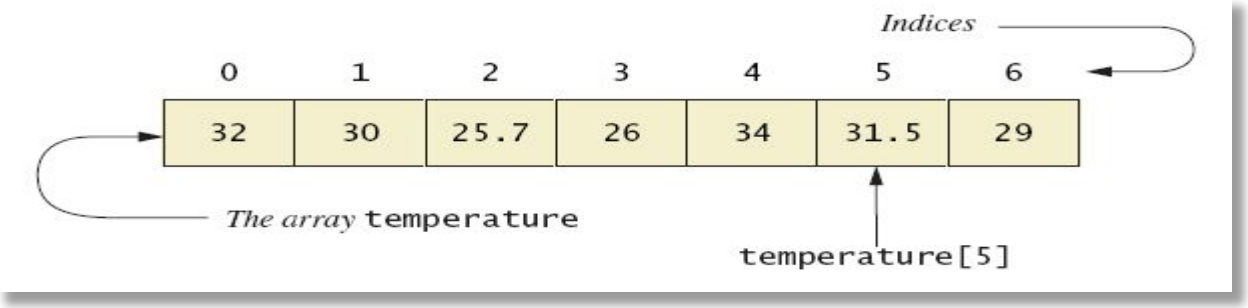
Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

```
double[] myList = new double[10];
```



Creating and Accessing Arrays

- Figure 7.1 A common way to visualize an array



- Note sample program, listing 7.1

```
class ArrayOfTemperatures
```

The Instance Variable `length`

- As an object an array has only one public instance variable
  - Variable `length`
    - Contains number of elements in the array
    - It is final, value cannot be changed
-

Java - Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.**println()** method, for example, the system actually executes several statements in order to display a message on the console.

Creating Method

```
public static int methodName(int a, int b) {
   // body
}
```

- **public static** – modifier
- **int** – return type
- **methodName** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters
- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when −

- the return statement is executed.
- it reaches the method ending closing brace.

The void Keyword

The void keyword allows us to create methods which do not return a value.

Array Details

- Syntax for declaring an array with **new**

```
Base_Type[] Array_Name = new Base_Type[Length];
```

- The number of elements in an array is its length
- The type of the array elements is the array's base type

Square Brackets with Arrays

- With a data type when declaring an array

```
int [ ] pressure;
```

- To enclose an integer expression to declare the length of the array

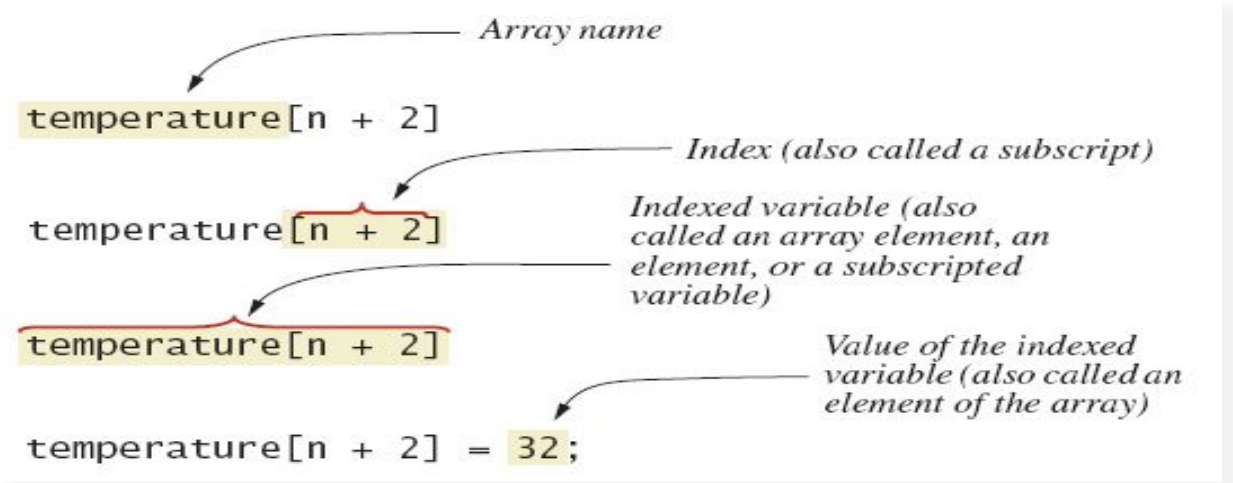
```
pressure = new int [100];
```

- To name an indexed value of the array

```
pressure[3] = keyboard.nextInt();
```

Array Details

- Figure 7.2 Array terminology



The Instance Variable `length`

- As an object an array has only one public instance variable
  - Variable `length`
    - Contains number of elements in the array
    - It is final, value cannot be changed
-

More About Array Indices

- Index of first array element is 0
- Last valid Index is `arrayName.length - 1`
- Array indices must be within bounds to be valid
  - When program tries to access outside bounds, run time error occurs

Initializing Arrays

- Possible to initialize at declaration time

```
double[] reading = {3.3, 15.8, 9.7};
```

- Also may use normal assignment statements
  - One at a time
  - In a loop

```
int[] count = new int[100];
for (int i = 0; i < 100; i++)
    count[i] = 0;
```

Arrays in Classes and Methods: Outline

- Indexed Variables as Method Arguments
- Entire Arrays as Arguments to a Method
- Arguments for the Method main
- Array Assignment and Equality
- Methods that Return Arrays

Entire Arrays as Arguments

- Declaration of array parameter similar to how an array is declared
- Example:

```
public class SampleClass
{
    public static void incrementArrayBy2(double[] anArray)
    {
        for (int i = 0; i < anArray.length; i++)
            anArray[i] = anArray[i] + 2;
    }
    <The rest of the class definition goes here.>
}
```

Entire Arrays as Arguments

- Note – array parameter in a method heading does not specify the length
  - An array of any length can be passed to the method
  - Inside the method, elements of the array can be changed
- When you pass the entire array, do not use square brackets in the actual parameter

Arguments for Method main

- Recall heading of method **`main`**

```
public static void main (String[] args)
```

- This declares an array
  - Formal parameter named **`args`**
  - Its base type is **`String`**
- Thus possible to pass to the run of a program multiple strings
  - These can then be used by the program

Array Assignment and Equality

- Arrays are objects
  - Assignment and equality operators behave (misbehave) as specified in previous chapter
- Variable for the array object contains memory address of the object
  - Assignment operator = copies this address
  - Equality operator == tests whether two arrays are stored in same place in memory

Array Assignment and Equality

- Note results of **==**
- Note definition and use of method **`equals`**
  - Receives two array parameters
  - Checks length and each individual pair of array elements
- Remember array types are reference types

Partially Filled Arrays

- Array size specified at definition
- Not all elements of the array might receive values
  - This is termed a *partially filled array*
- Programmer must keep track of how much of array is used

Sorting, Searching Arrays: Outline

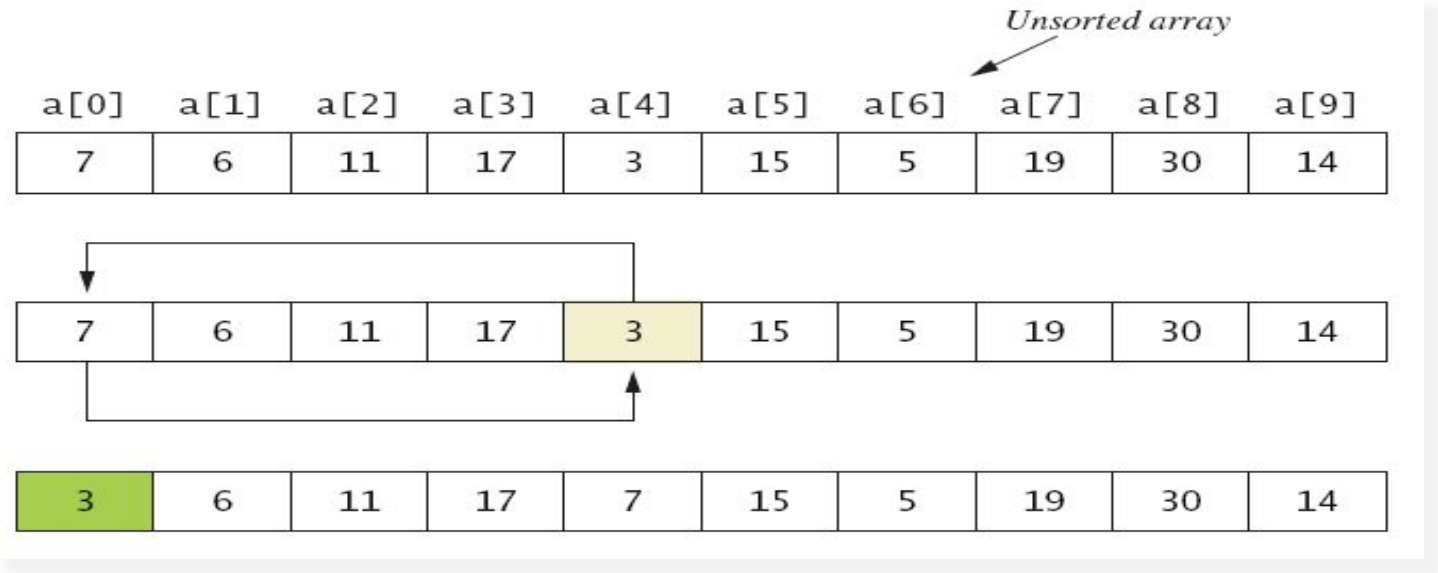
- Selection Sort
- Other Sorting Algorithms
- Searching an Array

Selection Sort

- Consider arranging all elements of an array so they are ascending order
- Algorithm is to step through the array
  - Place smallest element in index 0
  - Swap elements as needed to accomplish this
- Called an interchange sorting algorithm

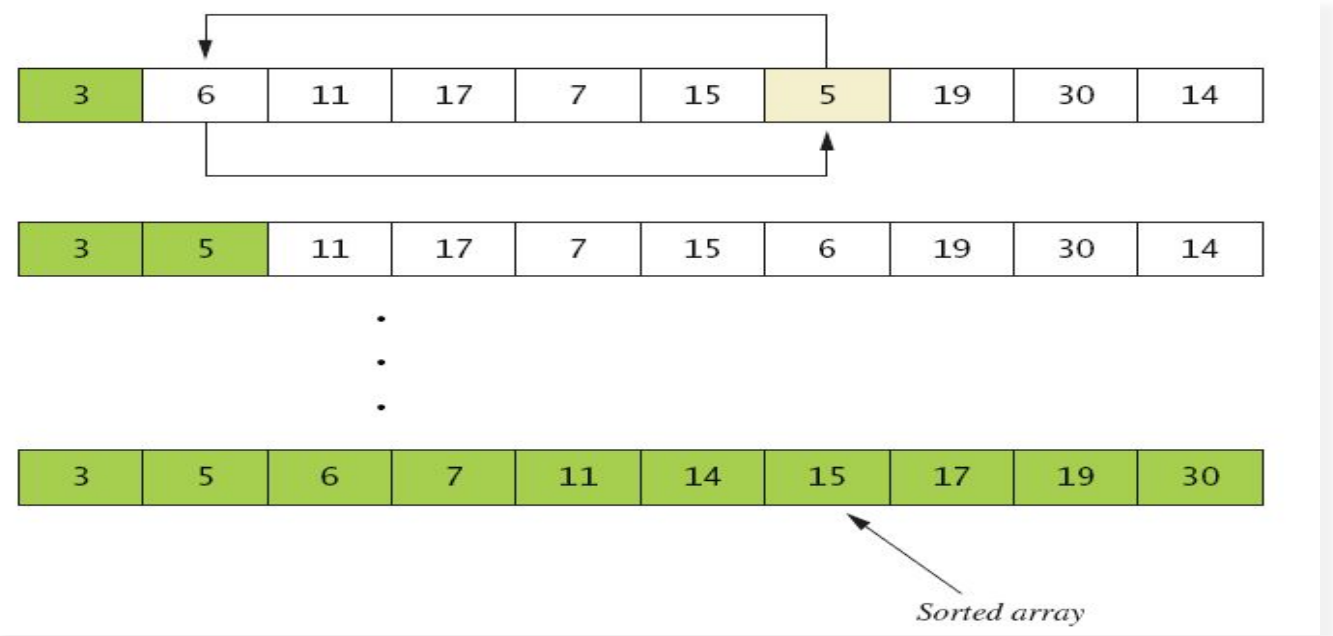
Selection Sort

- Figure 7.5a



Selection Sort

- Figure 7.5b



Selection Sort

- Algorithm for selection sort of an array

```
for (index = 0; index < a.length – 1; index++)
{// Place the correct value in a[index]:
    indexOfNextSmallest = the index of the smallest value among
                          a[index], a[index+1], ..., a[a.length – 1]
    Interchange the values of a[index] and a[indexOfNextSmallest].
    // Assertion: a[0] <= a[1] <=...<= a[index] and these
    // are the smallest of the original array elements.
    // The remaining positions contain the rest of the
    // original array elements.
}
```

Selection Sort

- View implementation of selection sort, listing 7.10
  `class ArraySorter`
- View demo program, listing 7.11
  `class SelectionSortDemo`

```
Array values before sorting:
7 5 11 2 16 4 18 14 12 30
Array values after sorting:
2 4 5 7 11 12 14 16 18 30
```

Sample screen output

Other Sorting Algorithms

- Selection sort is simplest
  - But it is very inefficient for large arrays
- Java Class Library provides for efficient sorting
  - Has a class called Arrays
  - Class has multiple versions of a sort method

Searching an Array

- Method used in **`OneWayNoRepeatsList`** is sequential search
  - Looks in order from first to last
  - Good for unsorted arrays
- Search ends when
  - Item is found … or …
  - End of list is reached
- If list is sorted, use more efficient searches

Multidimensional Arrays: Outline

- Multidimensional-Array Basics
- Multidimensional-Array Parameters and Returned Values
- Java's Representation of Multidimensional
- Ragged Arrays
- Programming Example: Employee Time Records

Multidimensional-Array Basics

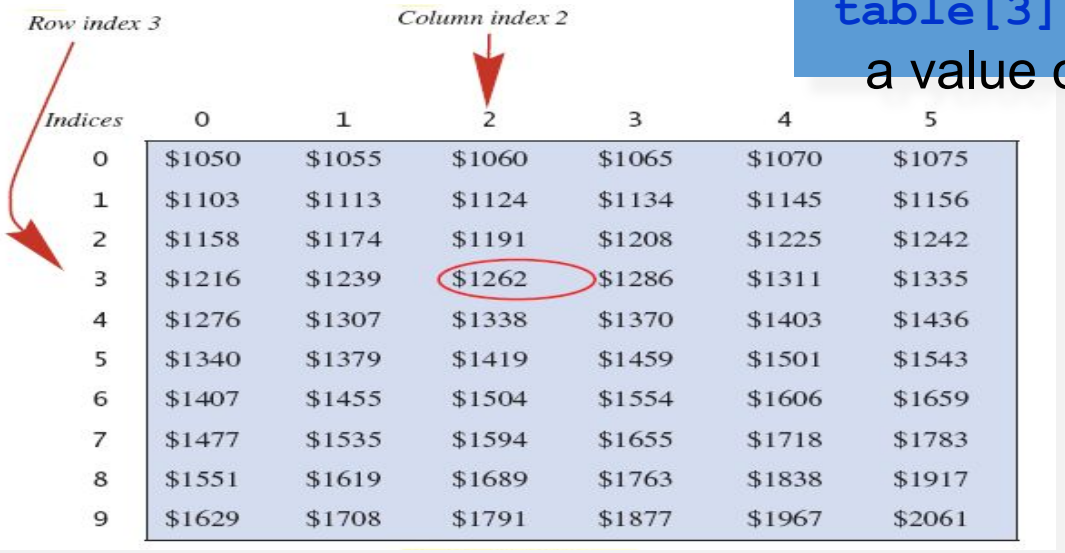
- Consider Figure 7.6, a table of values

Savings Account Balances for Various Interest Rates Compounded Annually (Rounded to Whole Dollar Amounts)

Year	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
6	$1340	$1379	$1419	$1459	$1501	$1543
7	$1407	$1455	$1504	$1554	$1606	$1659
8	$1477	$1535	$1594	$1655	$1718	$1783
9	$1551	$1619	$1689	$1763	$1838	$1917
10	$1629	$1708	$1791	$1877	$1967	$2061

Multidimensional-Array Basics

- Figure 7.7 Row and column indices for an array named **`table`**



Indices	0	1	2	3	4	5
0	$1050	$1055	$1060	$1065	$1070	$1075
1	$1103	$1113	$1124	$1134	$1145	$1156
2	$1158	$1174	$1191	$1208	$1225	$1242
3	$1216	$1239	$1262	$1286	$1311	$1335
4	$1276	$1307	$1338	$1370	$1403	$1436
5	$1340	$1379	$1419	$1459	$1501	$1543
6	$1407	$1455	$1504	$1554	$1606	$1659
7	$1477	$1535	$1594	$1655	$1718	$1783
8	$1551	$1619	$1689	$1763	$1838	$1917
9	$1629	$1708	$1791	$1877	$1967	$2061

`table[3][2]` has a value of 1262

Multidimensional-Array Basics

- We can access elements of the table with a nested for loop
- Example:

```
for (int row = 0; row < 10; row++)
   for (int column = 0; column < 6; column++)
      table[row][column] =
          balance(1000.00, row + 1, (5 + 0.5 * column));
```

- View sample program, listing 7.12

```
class InterestTable
```

Multidimensional-Array Basics

Balances for Various Interest Rates Compounded Annually
(Rounded to Whole Dollar Amounts)

Years	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
6	$1340	$1379	$1419	$1459	$1501	$1543
7	$1407	$1455	$1504	$1554	$1606	$1659
8	$1477	$1535	$1594	$1655	$1718	$1783
9	$1551	$1619	$1689	$1763	$1838	$1917
10	$1629	$1708	$1791	$1877	$1967	$2061

Sample screen output

Sorting Arrays

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

Multidimensional-Array Parameters and Returned Values

- Methods can have
  - Parameters that are multidimensional-arrays
  - Return values that are multidimensional-arrays
- View sample code, listing 7.13

```
class InterestTable2
```

Ragged Arrays

- Not necessary for all rows to be of the same length
- Example:

```
int[][] b;
b = new int[3][];
b[0] = new int[5]; //First row,  5 elements
b[1] = new int[7]; //Second row, 7 elements
b[2] = new int[4]; //Third row,  4 elements
```

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119

Bubble Sort Algorithm

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

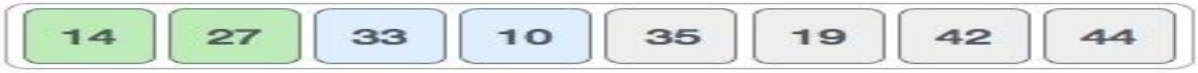
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Steps

Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

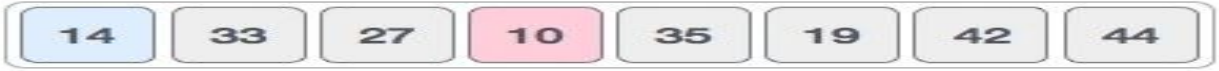
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

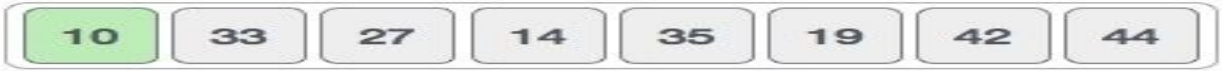
How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

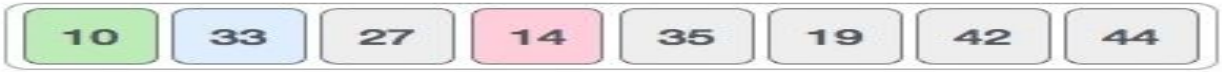


So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

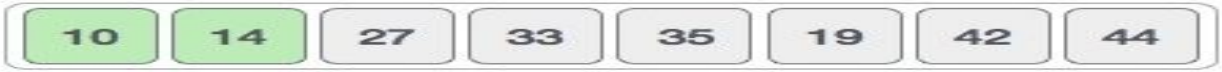


For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

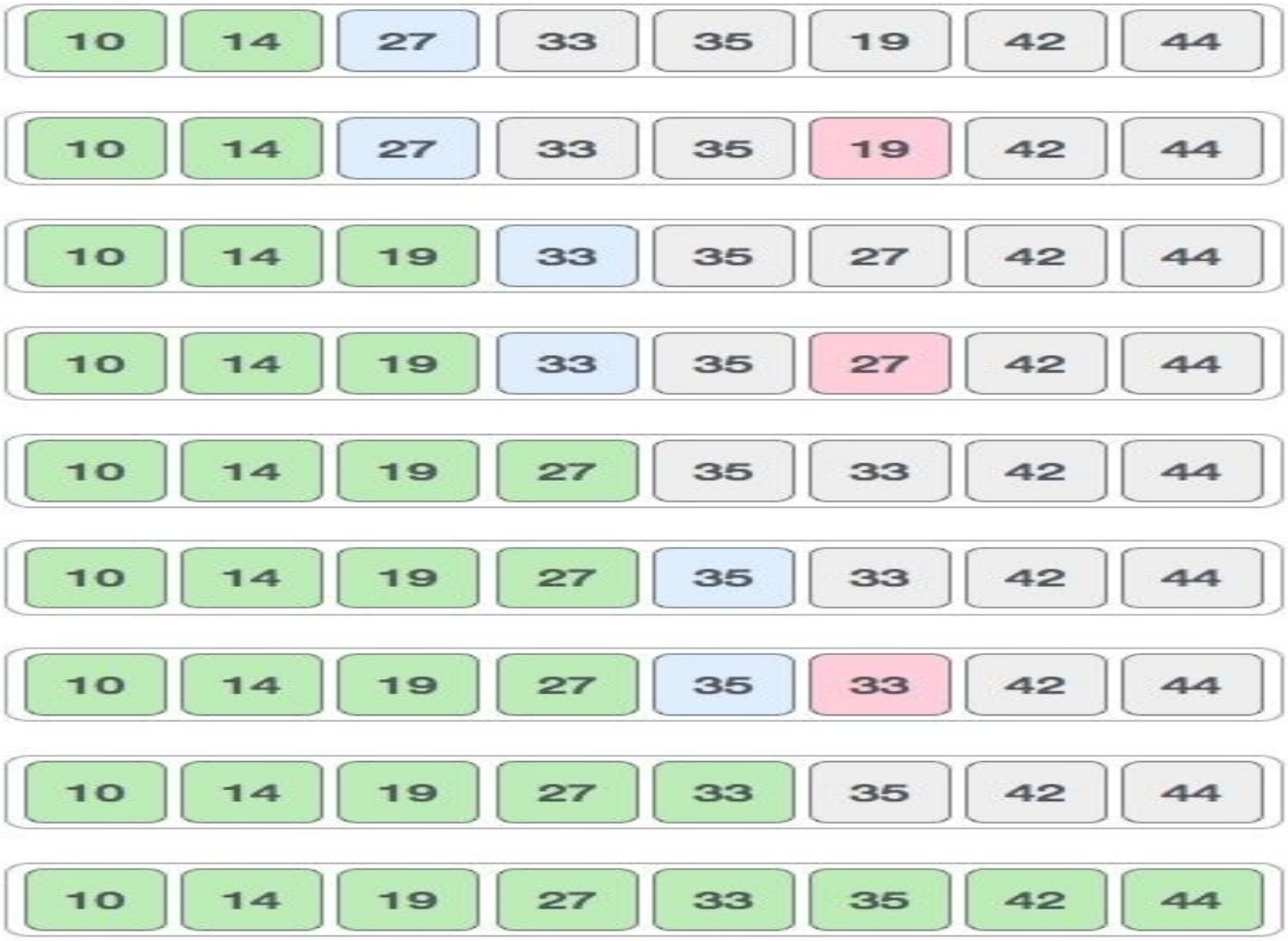


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Algorithm

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Sort Example Code

Searching Array

The simplest type of search is the sequential search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the desired element is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key.

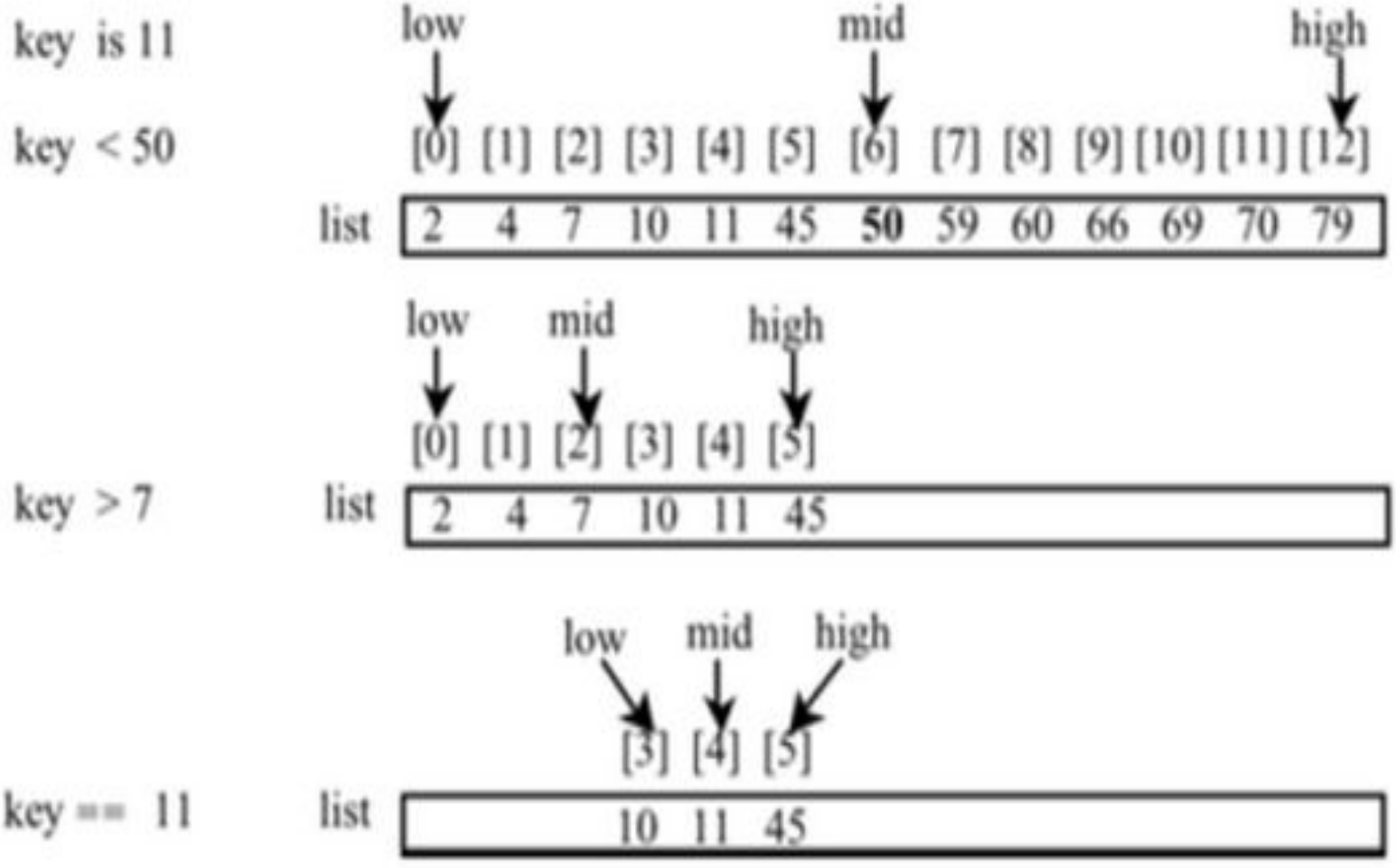
```
public static int linearSearch(int[] list, int key) {
   for (int i = 0; i < list.length; i++)
      if (key == list[i])
         return i;  //found it! so we immediately exit the method
   return -1;     //didn't find it, but we have to return something
 }
```

Binary Search

Linear Searches work well, but when used on arrays with a large number of elements, they potentially have to check every element to find a match. This can take a while, especially if we are doing multiple such searches.

A binary search works on an ordered list, and first compares the key with the element in the middle of the array. (In the case of an even number of elements in our list, we will use the element that ends the first half of the list as our "middle element").

If the key is less than the middle element, we only need to search the first half of the array, so we continue searching on this smaller list.
If the key is greater than the middle element, we only need to search the second half of the array, so we continue searching on this smaller list.
If the key equals the middle element, we have a match -- end the search


key is 11
low
mid
high
key < 50
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
list
2 4 7 10 11 45 50 59 60 66 69 70 79
low
mid
high
[0] [1] [2] [3] [4] [5]
key > 7
list
2 4 7 10 11 45
low
mid
high
[3] [4] [5]
key == 11
list
10 11 45

```
public static int binarySearch(int[] list, int key) {
   int low = 0;
   int high = list.length - 1;

   while (high >= low) {          //the loop only stops when
                                  //high gets updated to something
                                  //it shouldn't

     int mid = (low + high) / 2;  //note what this does
                                  //if (low + high) is odd

     if (key < list[mid])         //update index of the
       high = mid - 1;            //right-most element considered

     else if (key > list[mid])    //update index of
       low = mid + 1;             //left-most element considered

     else
       return mid;                //found it! now return the
                                  //index and exit the method

   }

   return -1 - low;  //key was not found, so

 }
```

Home work

1. Write a static method isStrictlyIncreasing(double[] in) that returns true if each value in the given array is greater than the value before it, or false otherwise.
2. Write a static method removeDuplicates(Character[] in) that returns a new array of the characters in the given array, but without any duplicate characters. Always keep the first copy of the character and remove subsequent ones. For example, if in contains b, d, a, b, f, a, g, a, a, and f, the method will return an array containing b, d a, f, and g. (Hint: One way to solve this problem is to create a boolean array of the same size as the given array in and use it to keep track of which characters to keep. The values in the new boolean array will determine the size of the array to return.)
3. Write a static method remove(intv,int[] in) that will return a new array of the integers in the given array, but with the value v removed. For example, if v is 3 and in contains 0, 1, 3, 2, 3, 0, 3, and 1, the method will return an array containing 0, 1, 2, 0, and 1.
4. Suppose that we are selling boxes of candy for a fund-raiser. We have five kinds of candy to sell: Mints, Chocolates with Nuts, Chewy Chocolates, Dark Chocolate Creams, and Sugar-Free Suckers. We will record a customer's order as an array of five integers, representing the number of boxes of each kind of candy. Write a static method combineOrder that takes two orders as its arguments and returns an array that represents the combined orders. For example, if order1 contains 0, 0, 3, 4, and 7, and order2 contains 0, 4, 0, 1, and 2, the method should return an array containing 0, 4, 3, 5, and 9.