GREATER JAKARTA • BANDUNG • MALANG

COMP6048001 Data Structures

Algorithm Efficiency and ArrayList
Week 3

Maria Seraphina Astriani
seraphina@binus.ac.id

People
Innovation
Excellence

Session Learning Outcomes

Upon successful completion of this course, students are expected to be able to:

- LO 1. Describe the use of various data structures
- LO 2. Apply appropriate operations for maintaining common data structures
- LO 3. Apply appropriate data structures and simple algorithms for solving computing problems
- LO 5. Explain the efficiency of some basic algorithms

Topics

- Algorithm Efficiency
- ArrayList

Algorithm Efficiency

Algorithm Efficiency and Big-O

- Whenever we write a new class, we will discuss the efficiency of its methods so that you know how they compare to similar methods in other classes.

- You can’t easily measure the amount of time it takes to run a program with modern computers.

Algorithm Efficiency and Big-O

- When you issue the command

```
java MyProgram
```

  (or click the Run button of your integrated development environment [IDE]), the operating system first loads the Java Virtual Machine (JVM).
- The JVM then loads the `.class` file for MyProgram, it then loads other `.class` files that MyProgram references, and finally your program executes.

Algorithm Efficiency and Big-O

- Most of the time it takes to run your program is occupied with the first two steps.
- If you run your program a second time immediately after the first, it may seem to **take less time**.
- This is because the operating system may have kept the files in a local memory area called a cache.
- However, if you have a large enough or com-plicated enough problem, then the actual running time of your program will dominate the time required to load the JVM and .class files.

Algorithm Efficiency and Big-O

- Because it is very difficult to get a precise measure of the performance of an algorithm or program, we normally try to approximate the effect of a change in the number of data items, *n*, that an algorithm processes.

- In this way, we can see how an algorithm's execution time increases with respect to *n*, so we can compare two algorithms by examining their growth rates.

Growth Rate

	n = 5 (Computation for 5 elements)	n = 10	n = 100	n = 1000	Growth Rate
Algorithm A	5	10	100	1.000	…….
Algorithm B	25	100	10.000	1.000.000	…….

Big-O

- Understanding how the execution time (and memory requirements) of an algorithm grows as a function of increasing input size gives programmers a tool for comparing various algorithms and how they will perform.
- Computer scientists have developed a useful terminology and notation for investigating and describing the relationship between input size and execution time.
- For example, if the time is approximately doubled when the number of inputs, *n*, is doubled, then the algorithm grows at a linear rate. Thus, we say that the growth rate has an order of *n*.

Formal Definition of Big-O

- Consider a program that is structured as follows:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        Simple Statement
    }
}
for (int k = 0; i < n; k++) {
     Simple Statement 1
     Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

- Let us assume that each Simple Statement takes one unit of time and that the for statements are free.
- The nested loop executes a Simple Statement n^2 times.
- Then five Simple Statements are executed n times in the loop with control variable k.
- Finally, 25 Simple Statements are executed after this loop.

Formal Definition of Big-O

- We would then conclude that the expression

  $$T(n) = n^2 + 5n + 25$$

  shows the relationship between processing time and n (the number of data items processed in the loop), where T(*n*) represents the processing time as a function of n.
- It should be clear that the n^2 term dominates as n becomes large.

Formal Definition of Big-O

- In terms of T(n), formally, the big-O notation

$$T(n) = O(f(n))$$

- The growth rate of f(n) will be determined by the growth rate of the fastest-growing term (the one with the largest exponent), which in this case is the n^2 term.
- This means that the algorithm in this example is an $O(n^2)$ algorithm rather than an $O(n^2 + 5n + 25)$ algorithm.
- In general, it is safe to ignore all constants and drop the lower-order terms when determining the order of magnitude for an algorithm.

Ignore All Constants and Drop The Lower-Order Terms

- Example:

  Function
  $n^2 + 3$

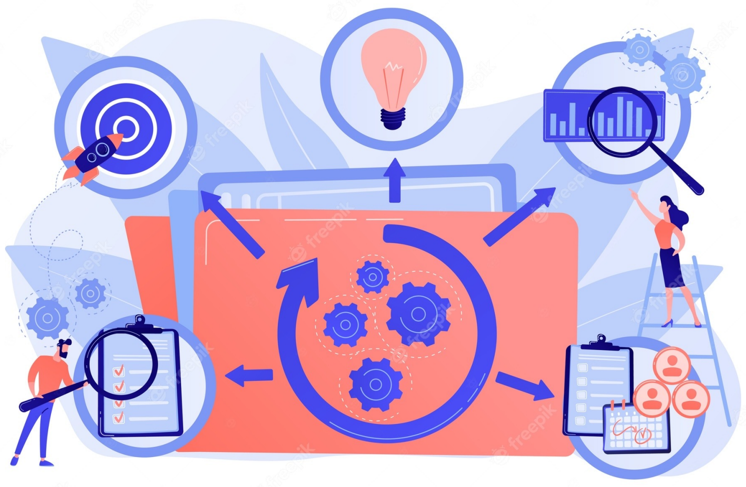
n	$n^2 + 3$	n^2
10	103	100
100	10003	10000
1000	1000003	1000000
10000	100000003	100000000
100000	10000000003	10000000000

- Example:

  Function
  $n \log n + n/2 + 5$

n	$n \log n + n/2 + 5$	$n \log n$
10	40	30
100	655	600
1000	9505	9000
10000	135005	130000
100000	1650005	1600000

Comparing Performance



Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Figure 2.3 shows the growth rate of a logarithmic, a linear, a log-linear, a quadratic, a cubic, and an exponential function by plotting $f(n)$ for a function of each type. Note that for small values of n, the exponential function is smaller than all of the others. As shown, it is not until n reaches 20 that the linear function is smaller than the quadratic. This illustrates two points. For small values of n, the less efficient algorithm may be actually more efficient. If you know that you are going to process only a limited amount of data, the $O(n^2)$ algorithm may be much more appropriate than the $O(n \log n)$ algorithm that has a large constant factor. However, algorithms with exponential growth rates can start out small but very quickly grow to be quite large.

FYI

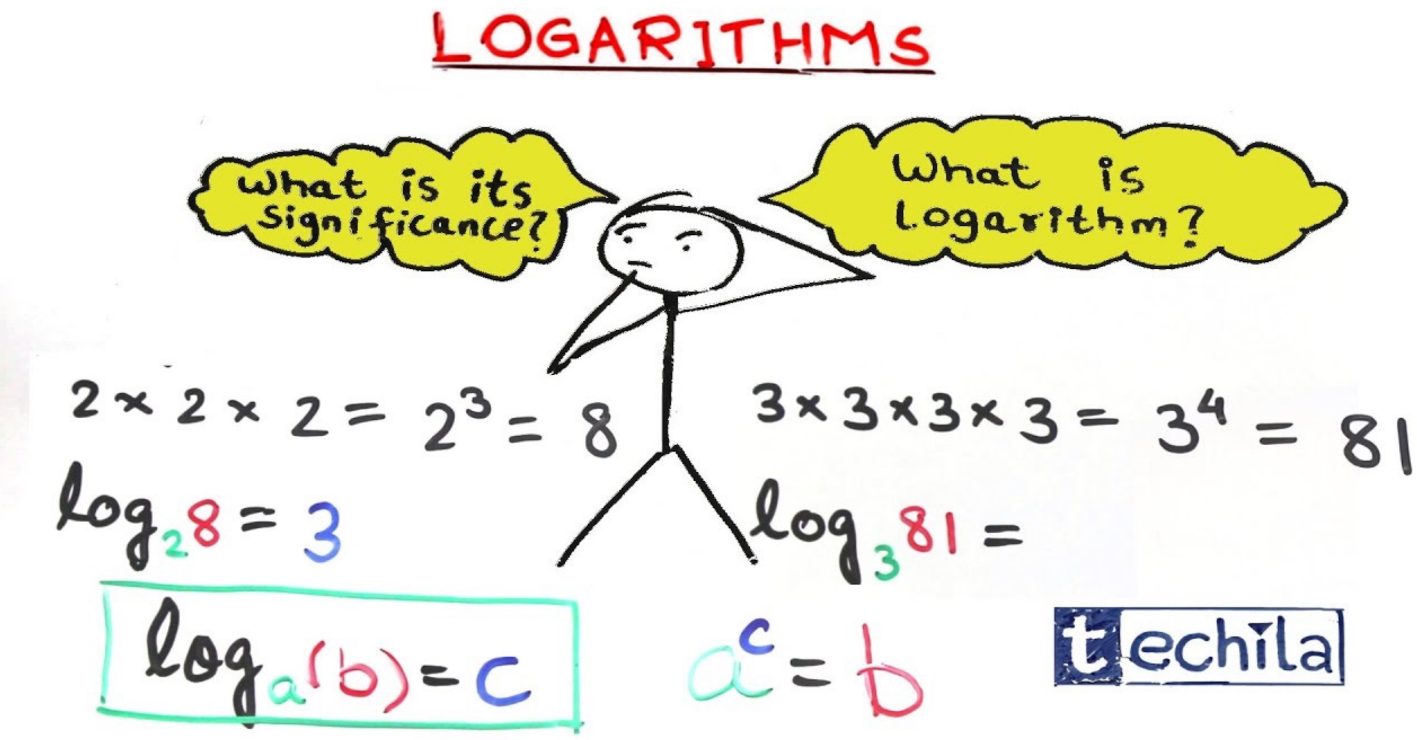
- Logarithms

  $\log_{2}(8) \Rightarrow 2^{3} = 8$

- In CS

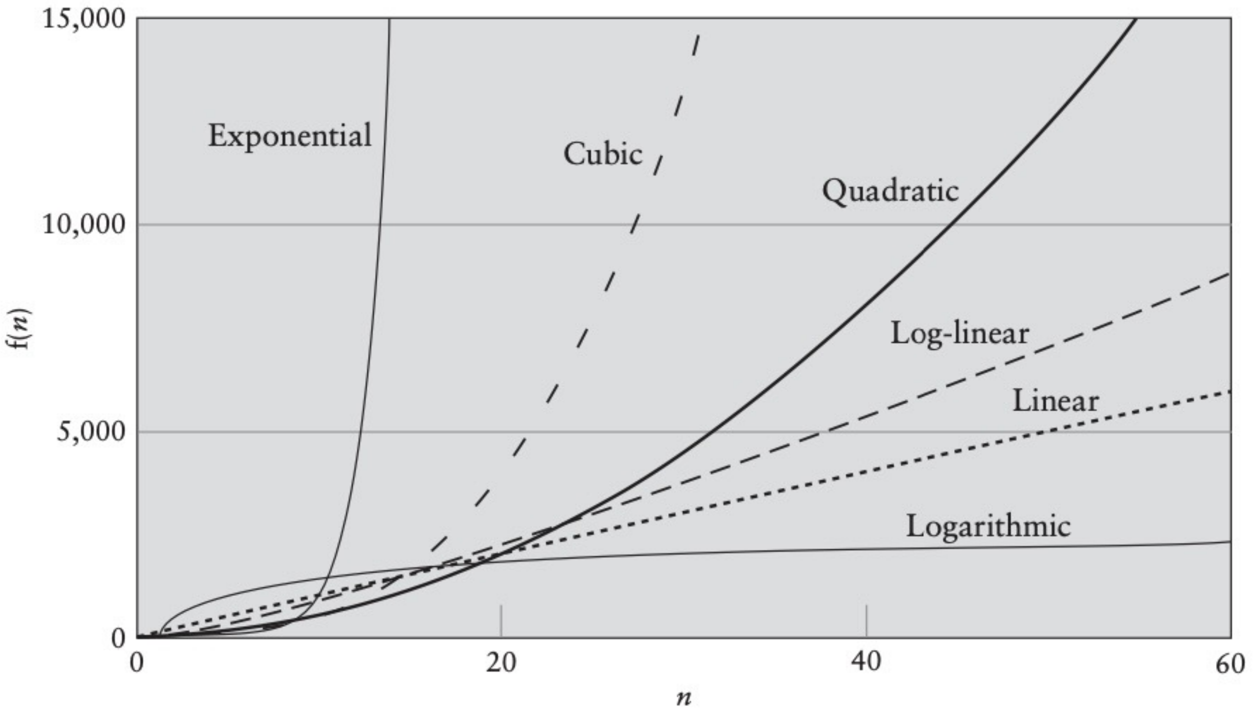
  $\log(8) \Rightarrow 2^{3} = 8$
  $\log 8 \Rightarrow 2^{3} = 8$

- Please answer the following questions:
  - log 2 = ?
  - log 32 = ?



https://www.youtube.com/watch?v=O821iMglrog

Different Growth Rate



The raw numbers in Figure 2.3 can be deceiving. Part of the reason is that big-O notation ignores all constants. An algorithm with a logarithmic growth rate $\mathbf{O}(\log n)$ may be more complicated to program, so it may actually take more time per data item than an algorithm with a linear growth rate $\mathbf{O}(n)$. For example, at $n = 25$, Figure 2.3 shows that the processing time is approximately 1800 units for an algorithm with a logarithmic growth rate and 2500 units for an algorithm with a linear growth rate. Comparisons of this sort are pretty meaningless. The logarithmic algorithm may actually take more time to execute than the linear algorithm for this relatively small data set. Again, what is important is the growth rate of these two kinds of algorithms, which tells you how the performance of each kind of algorithm changes with n.

Effects of Different Growth Rates

O(f(n))	f(50)	f(100)	f(100)/f(50)
$O(1)$	1	1	1
$O(\log n)$	5.64	6.64	1.18
$O(n)$	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	3.0×10^{64}	9.3×10^{57}	3.1×10^{93}

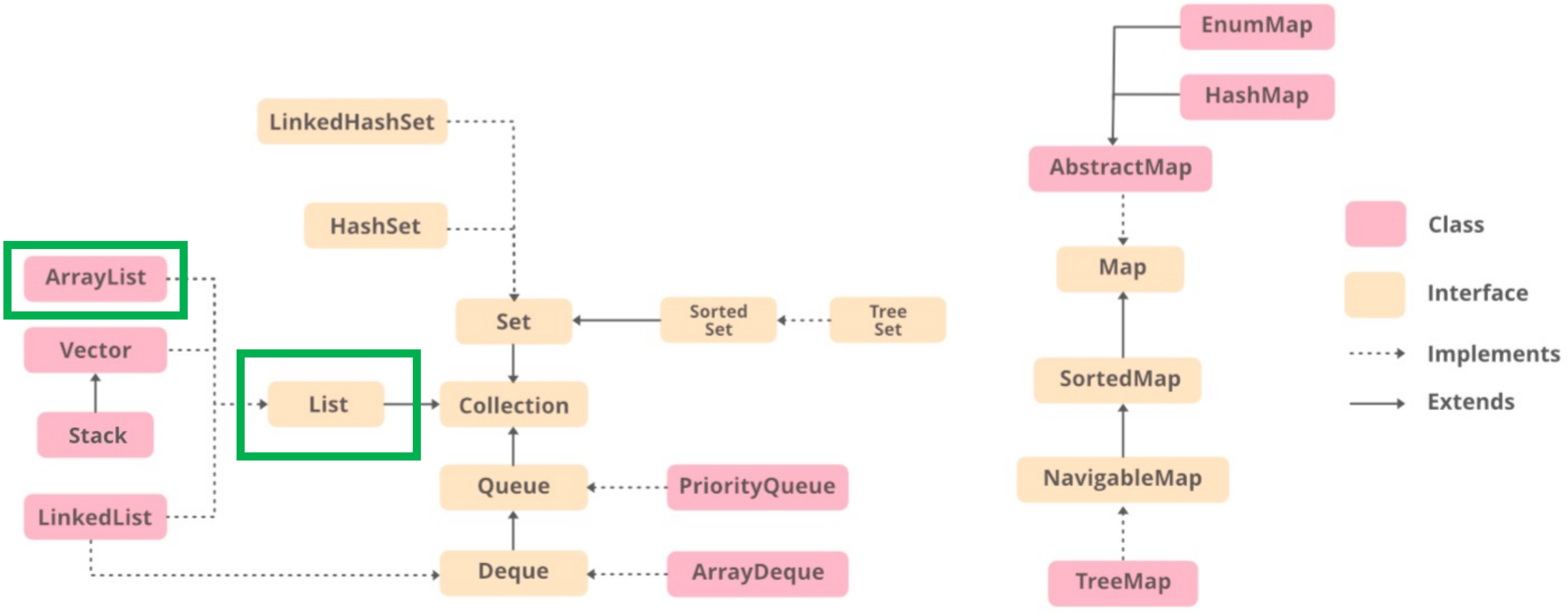
Hands-on Tutorial

- Big-O and the code

- Determine the Big-O based on the code

ArrayList

Big Picture – List and ArrayList in Java



https://www.geeksforgeeks.org/difference-between-list-and-arraylist-in-java/

Array

- An *array* is an indexed data structure.

- You can’t do the following with an array object:
  - Increase or decrease its length, which is fixed.
  - Add an element at a specified position without shifting the other elements to make room.
  - Remove an element at a specified position without shifting the other elements to fill in the resulting gap.

List

- The classes that implement the Java `List` interface (part of Java API `java.util`) all provide methods to do these operations and more.
- The following table shows some of the methods in the Java List interface.

Methods of Interface `java.util.List<E>`

Method	Behavior
`public E get(int index)`	Returns the data in the element at position `index`
`public E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`public int size()`	Gets the current size of the `List`
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`
`E remove(int index)`	Removes the entry formerly at position `index` and returns it

The symbol E in table is a type parameter. Type parameters are analogous to method parameters. In the declaration of an interface or class, the type parameter represents the data type of all objects stored in a collection.

List

- Methods of Interface `java.util.List<E>` perform the following operations:
  - Return a reference to an element at a specified location (method get)
  - Find a specified target value (method get)
  - Add an element at the end of the list (method add)
  - Insert an element anywhere in the list (method add)
  - Remove an element (method remove)
  - Replace an element in the list with another (method set)
  - Return the size of the list (method size)
  - Sequentially access all the list elements without having to manipulate a subscript

The `ArrayList` Class

- The simplest class that implements the List interface is the `ArrayList` class.
- An `ArrayList` object is an improvement over an array object in that it supports all of the operations just listed.
- `ArrayList` objects are used most often when a programmer wants to be able to grow a list by adding new elements to the end but still needs the capability to access the elements stored in the list in arbitrary order.

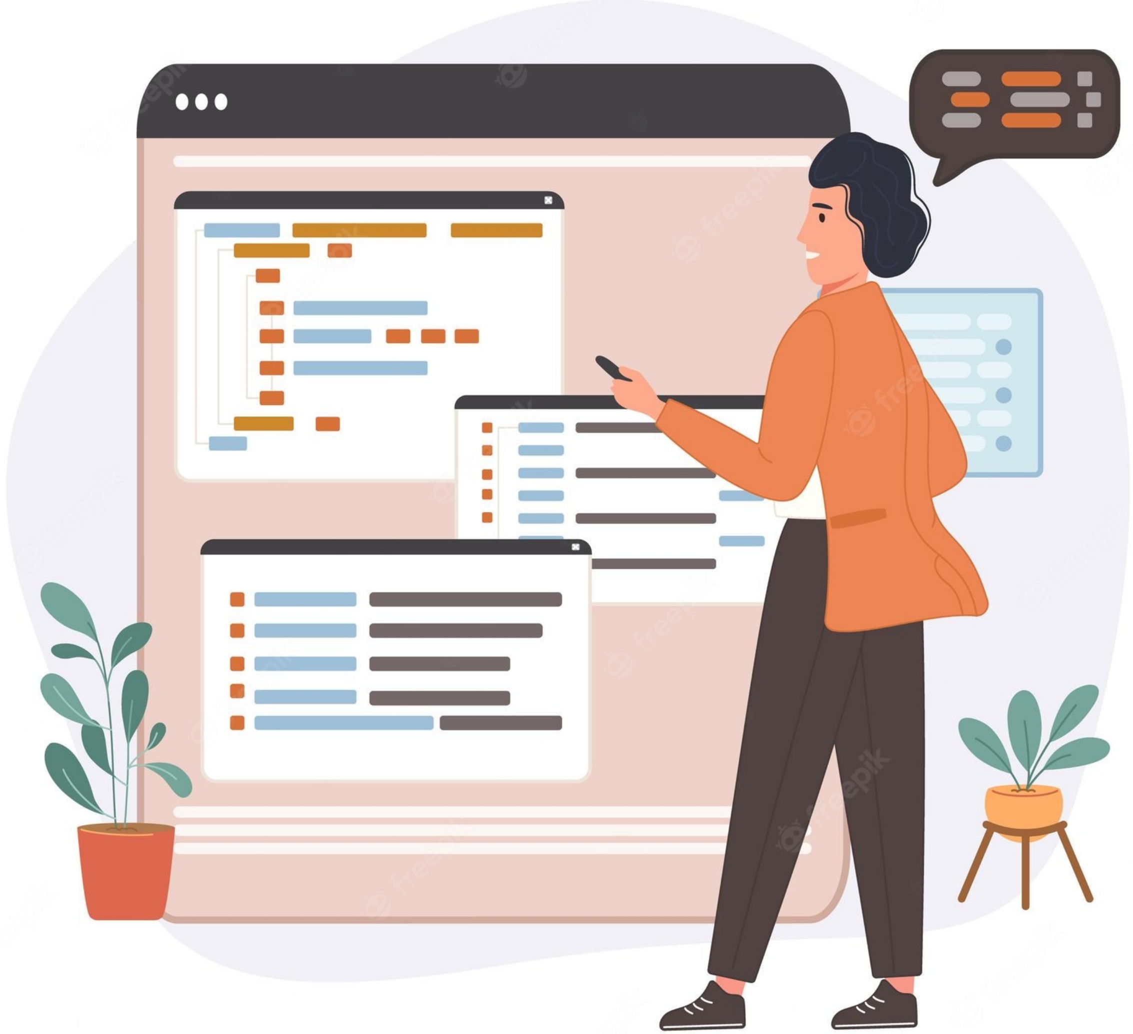
The `ArrayList` Class

- The size of an `ArrayList` automatically increases as new elements are added to it, and the size decreases as elements are removed.
- An `ArrayList` object has an instance method size that returns its current size.
- Each `ArrayList` object has a *capacity*, which is the number of elements it can store.
- If you add a new element to an `ArrayList` whose current size is equal to its capacity, the capacity is automatically increased.

The `ArrayList` Class

The statements:

- declare List variables
- add()
- size()
- remove()
- get()
- set()
- indexOf()



Applications of `ArrayList`

- The following statements create an `ArrayList<Integer>` object and load it with the values stored in a type `int[]` array.

```
List<Integer> some = new ArrayList<>();
int[] nums = {5, 7, 2, 15};
for (int numsNext : nums) {
    some.add(numsNext);
    System.out.println(some);
}
```

- Loop exit occurs after the last Integer object is stored in some.
- The output displayed by this fragment follows:

```
[5]
[5, 7]
[5, 7, 2]
[5, 7, 2, 15]
```

Array and ArrayList Tutorial

```
Printing Array arr : [1, 2, 3, 4, 5]
arr index 2 : 3
Printing ArrayList aL : [a, b, c, d, e]
aL size : 5
After removed index 1 : [a, c, d, e]
get aL index 3 : e
Printing List from arr: [1, 2, 3, 4, 5]
```

- Array
- ArrayList
- Converting Array to ArrayList

FYI (https://www.geeksforgeeks.org/array-vs-arraylist-in-java/):

- An array can contain both primitive data types as well as objects of a class depending on the definition of the array.
- However, ArrayList only supports object entries, not the primitive data types.

Additional Materials

Sorting Algorithms: Bubble Sort and Selection Sort

Bubble Sort (a.k.a Sinking Sort)

- Repeatedly traverse through the list and swap neighboring elements if not in order till no swap is required.

- Idea
  - Scan the array from left to right;
  - Compare each pair of adjacent elements; and
  - Swap them if they are out of order (i.e., the previous one is larger than the latter one in the default case).

Bubble sort in 2 minutes.mp4

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[0] and A[1]. Swap the two elements if A[0] > A[1].

0	1	2	3	4
6	4	10	2	8

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[0] and A[1]. Swap the two elements if A[0] > A[1].

0	1	2	3	4
4	6	10	2	8

Swap

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[1] and A[2]. Swap the two elements if A[1] > A[2].

0	1	2	3	4
4	6	10	2	8

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[2] and A[3]. Swap the two elements if A[2] > A[3].

0	1	2	3	4
4	6	10	2	8

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[2] and A[3]. Swap the two elements if A[2] > A[3].

0	1	2	3	4
4	6	2	10	8

Swap

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[3] and A[4]. Swap the two elements if A[3] > A[4].

0	1	2	3	4
4	6	2	10	8

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Compare A[3] and A[4]. Swap the two elements if A[3] > A[4].

0	1	2	3	4
4	6	2	8	10

Swap

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1 ends when reaching the last element
  - The largest element is now at the last position

0	1	2	3	4
4	6	2	8	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	4	6	2	8	10

- Pass 2
  - Compare A[0] and A[1]. Swap the two elements if A[0] > A[1].

0	1	2	3	4
4	6	2	8	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	4	6	2	8	10

- Pass 2
  - Compare A[1] and A[2]. Swap the two elements if A[1] > A[2].

0	1	2	3	4
4	6	2	8	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	4	6	2	8	10

- Pass 2
  - Compare A[1] and A[2]. Swap the two elements if A[1] > A[2].

0	1	2	3	4
4	2	6	8	10

Swap

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	4	6	2	8	10

- Pass 2
  - Compare A[2] and A[3]. Swap the two elements if A[2] > A[3].

0	1	2	3	4
4	2	6	8	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	4	6	2	8	10

- Pass 2 ends when reaching the 2nd last element
  - The 2nd largest element is now at the 2nd last position

0	1	2	3	4
4	2	6	8	10

Example

- Pass 3 ends when reaching the 3rd last element
  - The 3rd largest element is now at the 3rd last position

0	1	2	3	4
2	4	6	8	10

- Pass 4 ends when reaching the 4th last element
  - The 4th largest element is now at the 4th last position

0	1	2	3	4
2	4	6	8	10

- A is sorted.

Bubble Sort

- More explanation & code: https://www.geeksforgeeks.org/bubble-sort/
- Conclusion:
  - Pairwise swap
  - The complexity of bubble sort is $O(n^2)$

Selection Sort

- Repeatedly traverse through the unsorted list, find the maximum element and swap it with the last element

- Idea
  - Scan the array from left to right until the i-th last element;
  - Find the i-th largest element in the array; and
  - Swap the i-th largest element and the i-th last element.

i-th = i-th pass

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Scan the array and find the largest element.

0	1	2	3	4
6	4	10	2	8

Example

- To sort input array **A** in ascending order

Index	0	1	2	3	4
Element	6	4	10	2	8

- Pass 1
  - Swap the largest element and the last element.

0	1	2	3	4
6	4	8	2	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	6	4	8	2	10

- Pass 2
  - Scan the array and find the 2nd largest element.

0	1	2	3	4
6	4	8	2	10

Example

- Array **A** after Pass 1

Index	0	1	2	3	4
Element	6	4	8	2	10

- Pass 2
  - Swap the 2nd largest element and the 2nd last element.

0	1	2	3	4
6	4	2	8	10

Example

- Array **A** after Pass 2

Index	0	1	2	3	4
Element	6	4	2	8	10

- Pass 3
  - Scan the array and find the 3rd largest element.

0	1	2	3	4
6	4	2	8	10

Example

- Array **A** after Pass 2

Index	0	1	2	3	4
Element	6	4	2	8	10

- Pass 3
  - Swap the 3rd largest element and the 3rd last element.

0	1	2	3	4
2	4	6	8	10

Example

- Array **A** after Pass 3

Index	0	1	2	3	4
Element	2	4	6	8	10

- Pass 4
  - Find the 4th largest element, and swap it and the 4th last element.

0	1	2	3	4
2	4	6	8	10

  - A is sorted.

Selection Sort

- More explanation & code: https://www.geeksforgeeks.org/selection-sort/

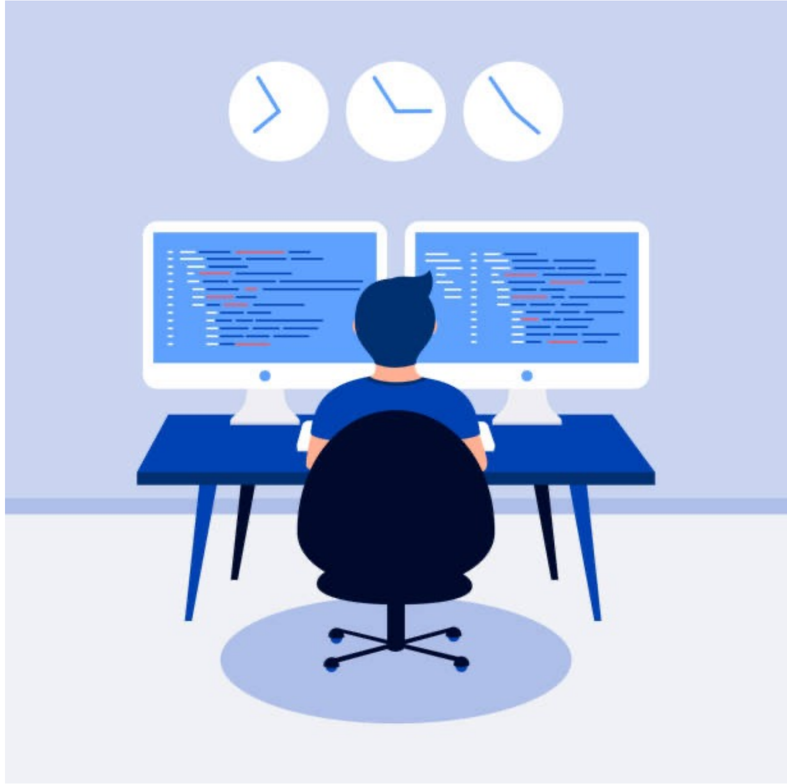
- Conclusion:
  - Find and swap
  - The complexity of Selection Sort is $O(n^2)$

Activities

Activities

1. Determine how many times the output statement is executed in each of the following fragments.
   Indicate whether the algorithm is O(n) or O(n^2).

a.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        System.out.println(i + "  " + j);
```
b.
```
for (int i = 0; i < n; i++)
    for (int j = 0; j < 2; j++)
        System.out.println(i + "  " + j);
```
c.
```
for (int i = 0; i < n; i++)
    for (int j = n - 1; j >= i; j--)
        System.out.println(i + "  " + j);
```
d.
```
for (int i = 1; i < n; i++)
    for (int j = 0; j < i; j++)
        if (j % i == 0)
            System.out.println(i + "  " + j);
```



Activities

2. Trace the execution of the following:

a.
```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = 3; i < anArray.length - 1; i++)
    anArray[i + 1] = anArray[i];
```

and the following:

b.
```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = anArray.length - 1; i > 3; i--)
    anArray[i] = anArray[i - 1];
```

What are the contents of `anArray` (question a and b) after the execution of each loop?

Activities

- Submission
- https://forms.office.com/r/kM3pffBjv0

References

- Koffman, E. B., & Wolfgang, P. A. (2021). Data structures: abstraction and design using Java (4th Edition). John Wiley & Sons. [DSA]
- Weiss, M. A. (2010). Data Structures and Problem Solving Using Java (Fourth Edition). Addison-Wesley. [DSPS]
- Weiss, M. A. (2014). Data structures and algorithm analysis in Java (3rd Ed). Pearson.
- Karumanchi, N. (2017). Data Structures and Algorithms Made Easy In JAVA. CareerMonk.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in Java (6th Ed.). John Wiley & Sons.

Thank you