Data Structures and Algorithms (23CSE203)

Introduction

What is Data Structure?

- It is a way of organizing data in a computer in such a way that we can perform operations on these data in an effective way.
- it is a systematic way of organizing and accessing data.

Importance of Data Structures in Daily Life

- Manage large volumes of data in systems such as banking, healthcare, and stock markets.
- Enable faster access and manipulation of data.
- Power applications like search engines, GPS, social media, and e-commerce.
- Essential for memory optimization in devices like smartphones and IoT devices.
- Improve code performance and scalability in everyday programming.

- A **data structure** is a systematic way of organizing and accessing data, and an **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
- Characterizing the running times of algorithms and data structure operations is important, as it provides a natural measure of their efficiency or “goodness”.
- The running time of an algorithm or data structure operation increases with the input size.
- Focusing on running time as a primary measure of goodness requires that we be able to use a few mathematical tools.

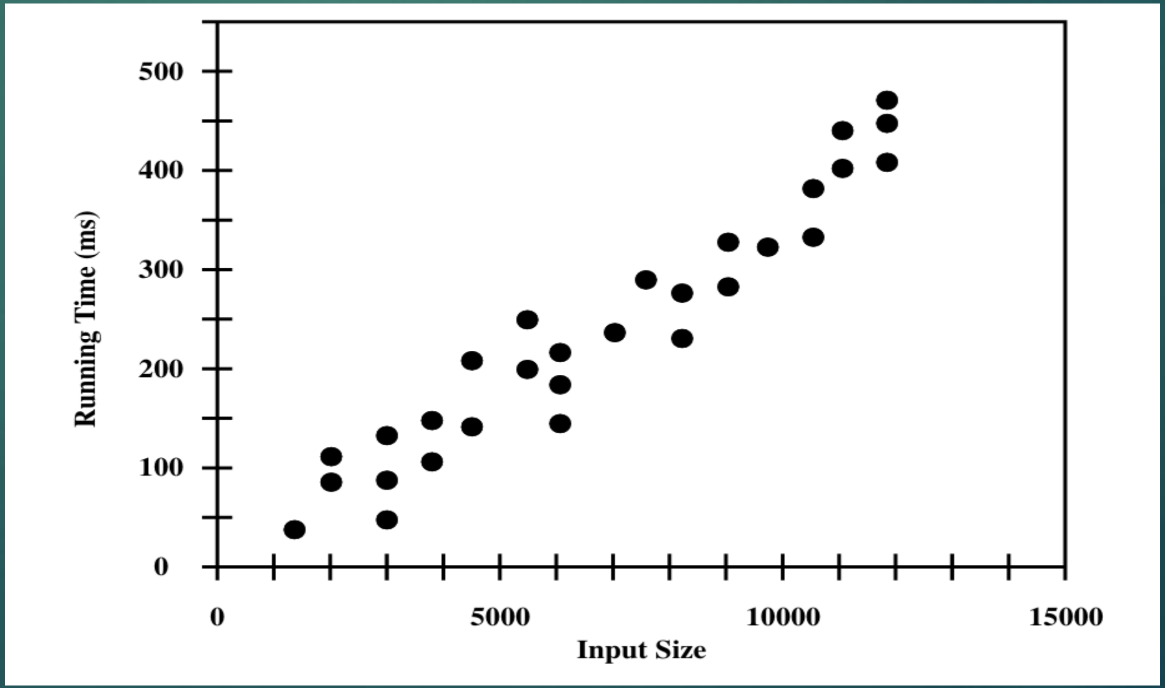
In this Unit

- Commonly used Data Structures (DS)
- Basic Complexity Analysis
- Costs & benefits associated with every DS
- Recursion
- Implementation of Different DS

- **Abstract Data Type** (ADT) is a model or concept for data types, where only the behavior (operations) is defined, but not the implementation details.
- An ADT is a logical description of how data is organized and what operations are performed on it, without specifying how these operations are implemented.
- Examples of ADTs:
  - Array, List, Stack, Queue, Tree, Graph.
- Key Characteristics:
  - ✓ Focuses on what the data structure does, not how it does it.
  - ✓ Implementation is hidden (encapsulation)
  - ✓ Promotes modularity and code reusability
  - ✓ Interfaces define the set of operations, like insert(), delete(), search().

Experimental Studies

```
from time import time
start_time = time( )                    #record the starting time
run algorithm
end_time = time( )                        #record the ending time
elapsed = end_time − start_time   #compute the elapsed time
```



Challenges of Experimental Analysis

- Experimental running times of two algorithms are difficult to directly com pare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Primitive Operations

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function (excluding operations executed within the function)
- Returning from a function.

Best case

The key is the first integer in the array. In this case the running time is short. This is the best case for linear search algorithm, because it is not possible for linear search to look at less than one value.

Worst Case

If the key doesn't present in the array or key is the last in the array, then the running time of linear search is relatively long, because the algorithm must examine 'n' values. This is the worst case for this algorithm, because linear search never looks at more than 'n' values.

When analyzing algorithms, we often consider the worst-case scenario.

Average case

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

Example: If we implement linear search as a program and run it many times on many different arrays of size n.

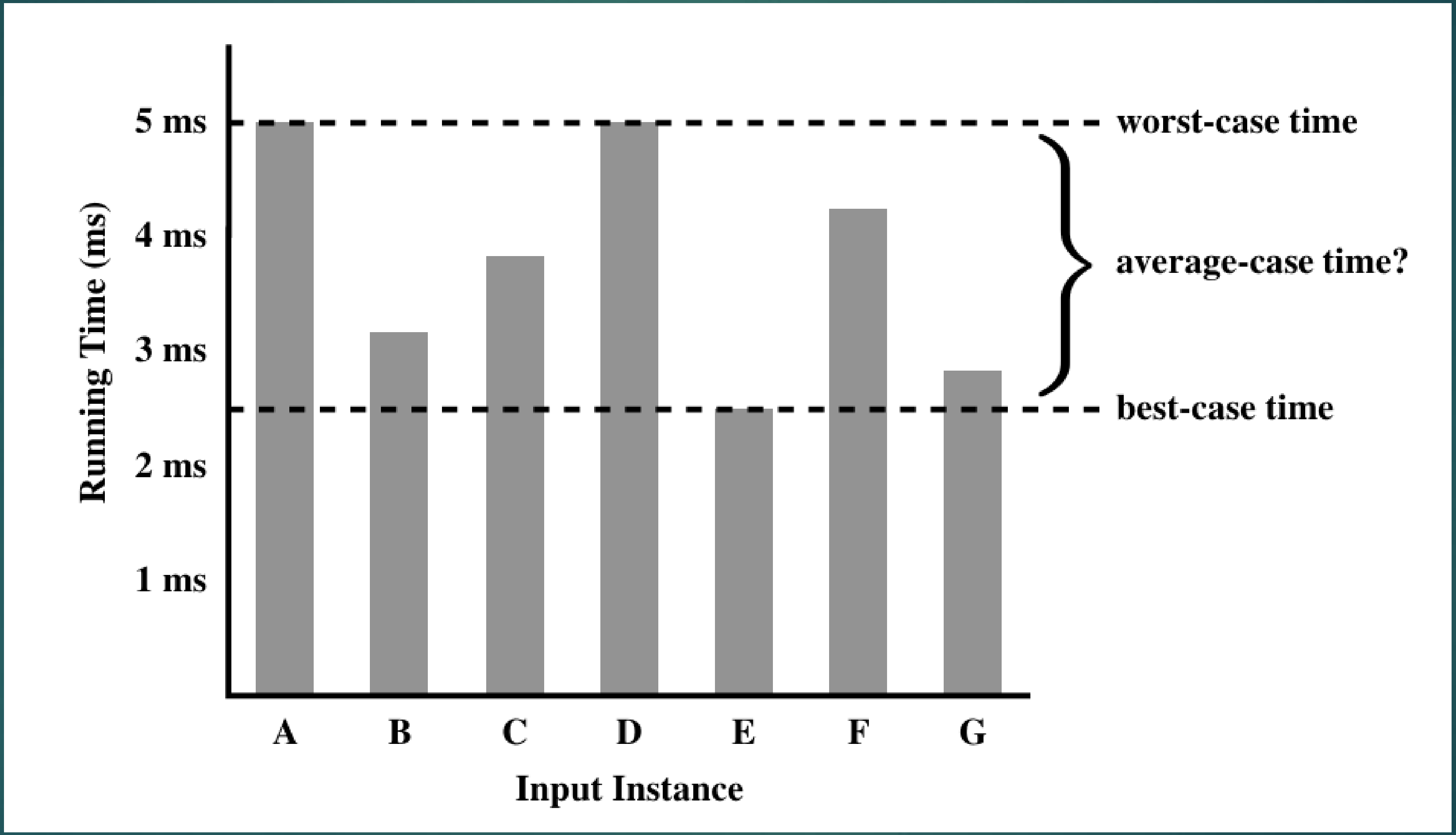
we expect the algorithm on average to go halfway through the array before finding the value we seek.

Worst case analysis - The Winner

- Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple. Also, this approach typically leads to better algorithms.

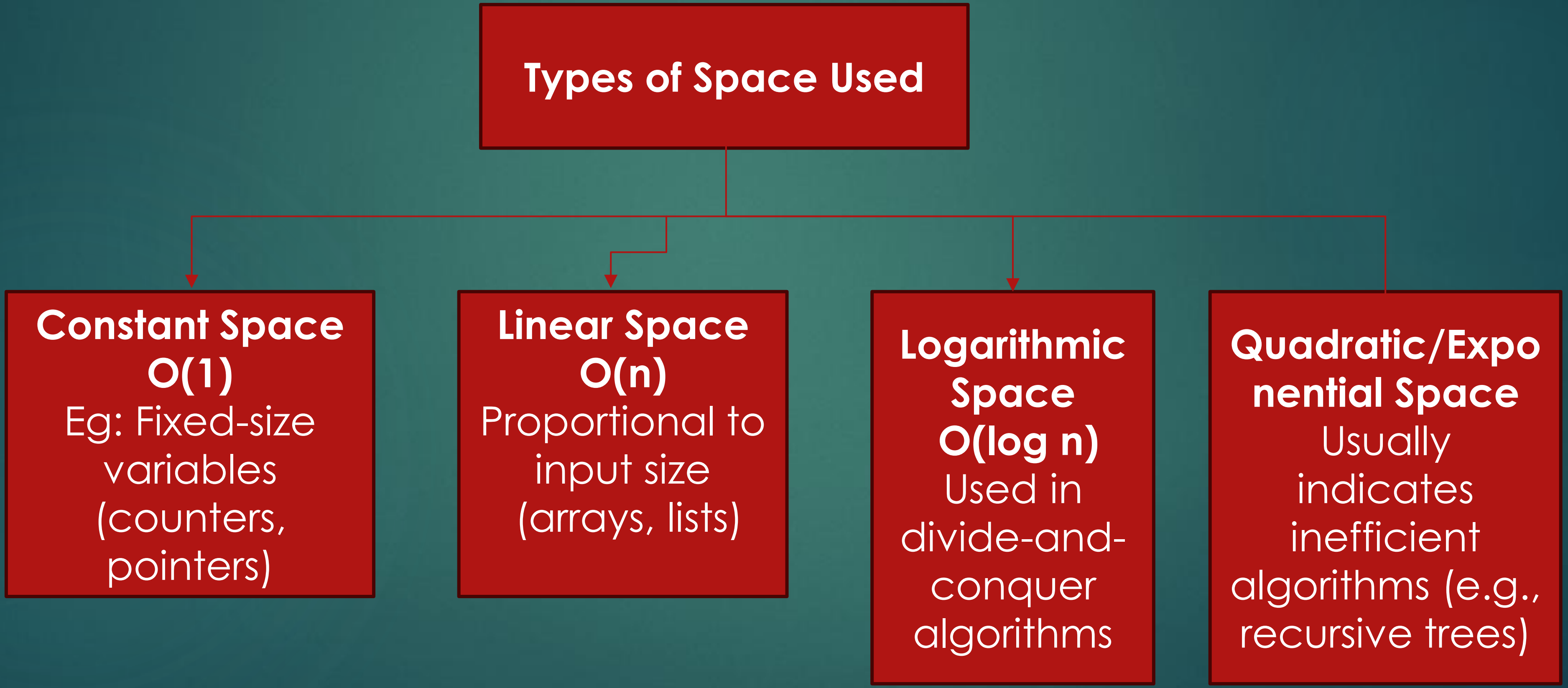
- In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case.

Focusing on the Worst-Case Input



Space Complexity

- Space complexity refers to the total amount of memory (space) used by an algorithm during its execution.
- Includes: Data space (Space required for constants, variables, intermediate variables, dynamic variables), Auxiliary (temporary) space , Recursion stack (To store return address, return values etc.)
- Efficient use of memory is crucial for performance, especially in memory-constrained systems.



Types of Space Used

Constant Space O(1)
Eg: Fixed-size variables (counters, pointers)

Linear Space O(n)
Proportional to input size (arrays, lists)

Logarithmic Space O(log n)
Used in divide-and-conquer algorithms

Quadratic/Exponential Space
Usually indicates inefficient algorithms (e.g., recursive trees)

```
def square(n):
    result = n * n
    return result
```

- Only two variables : ‘n’ and ‘result’. No matter what ‘n’ is, memory use stays the same. So space complexity is O(1).

```
def print_numbers(n):
    numbers = []
    for i in range(1, n + 1):
        numbers.append(i)
    return numbers
```

- We store all numbers from 1 to n ; the list grows as ‘n’ grows. So space complexity is O(n)

```
def count_down(n):
    if n == 0:
        return
    print(n)
    count_down(n // 2)
```

- Function call happen as ‘n’ is divided by 2 each time. Call Stack uses O(logn)

```
def build_table(n):
    table = []
    for i in range(n):
        row = []
        for j in range(n):
            row.append(i * j)
        table.append(row)
    return table
```

- We create ‘n*n’ table like a multiplication table. Space Complexity is O(n*n)

Time Complexity

- There are several methods to find the runtime of a program
  - ✓ Operation Count
  - ✓ Step Count
  - ✓ **Asymptotic notations**
  - ✓ Practical method

Functions used in the Analysis of Algorithms

- **The Constant Function**
- ✓ The simplest function we can think of is the constant function. This is the function, f(n)= c, for some fixed constant c.
- ✓ It does not matter what the value of n is; f(n) will always be equal to the constant value c
- ✓ Any other constant function, f(n)=c, can be written as a constant c times g(n).Thatis, f(n)=c.g(n) in this case.
- ✓ It characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

The Logarithm Function

- ✓ the logarithm function, f (n)= $\log_b n$, for some constant b > 1. This function is defined as follows: x = $\log_b n$ if and only if $b^x = n$.
- ✓ The value b is known as the base of the logarithm.
- ✓ Any other constant function, f(n)=c, can be written as a constant c times g(n). That is, f(n)=c.g(n) in this case.
- ✓ It characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

- **The Linear Function**

- ✓ Given an input value n, the linear function f assigns the value n itself; ; $f(n) = n.$
- ✓ This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements.
- ✓ The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer's memory, because reading in the n objects already requires n operations.

- **The N-Log-N Function**
- ✓ The function that assigns to an input n the value of n times the logarithm base-two of n; $f(n) = nlogn$.
- ✓ This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function .
- ✓ The fastest possible algorithms for sorting n arbitrary values require time proportional to nlogn.

- **The Quadratic Function**
- ✓ Given an input value ‘n’, the function f assigns the product of ‘n’ with itself; $f(n) = n^2$
- ✓ There are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times.
- **The Cubic Function and Other Polynomials**

$$f(n) = n^3$$

$$f(n) = 2 + 5n + n^2$$

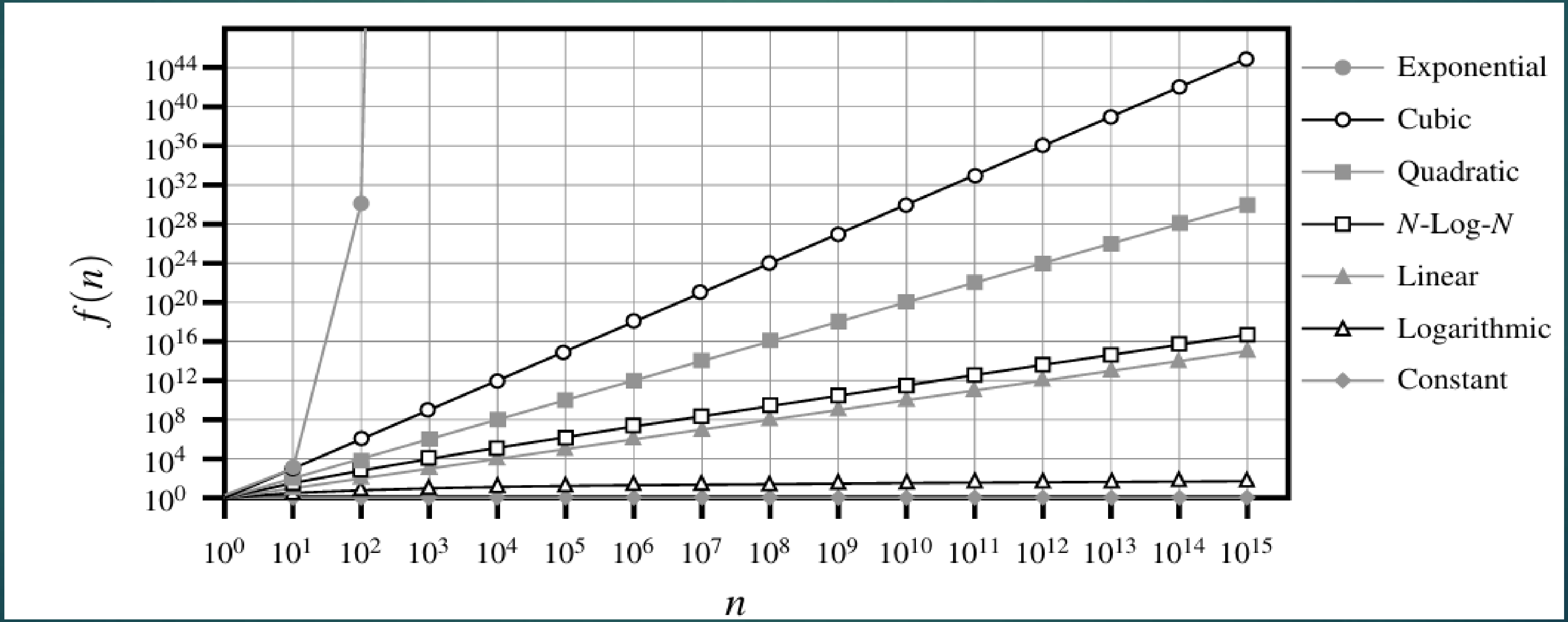
$$f(n) = 1 + n^3$$

$$f(n) = n^2$$

$$f(n) = n$$

The Exponential Function

- ✓ $f(n) = b^n$; where ‘b’ is a positive constant, called the base, and the argument ‘n’ is the exponent.
- ✓ Function $f(n)$ assigns to the input argument ‘n’ the value obtained by multiplying the base ‘b’ by itself ‘n’ times.



Asymptotic analysis

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- The goal is to determine the best case, worst case and average case time required to execute a given task.

Asymptotic notation

- It describes the rate of growth of functions
- It is a way to compare the sizes of functions
- Focus on what is important by abstracting low order terms and constant factors.
- Types of asymptotic notations are:

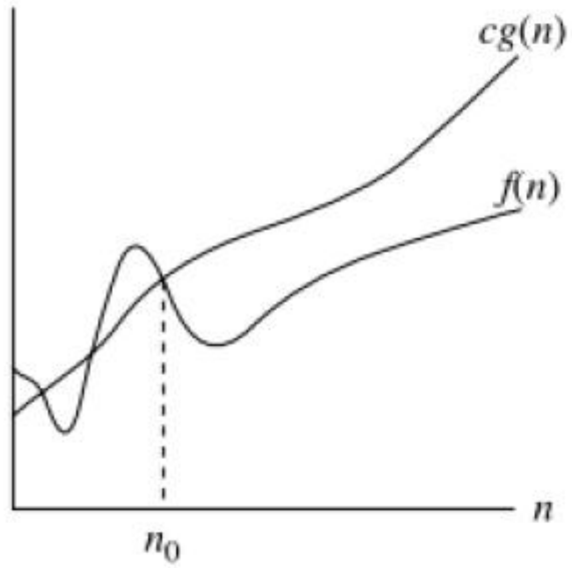
1. **Big-O Notation (O)** – It describes worst case scenario
2. **Omega Notation (Ω)** – It describes best case scenario.
3. **Theta Notation (θ)** – It represents the average complexity of an algorithm

Properties of Asymptotic Notations

- If **$f(n) = O(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq f(n) \leq c.g(n)$**, for all $n \geq n0$
- If **$f(n) = \Omega(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq c.g(n) \leq f(n)$**, for all $n \geq n0$
- If **$f(n) = \Theta(g(n))$**, then there exists positive constants c1, c2, n0 such that **$0 \leq c1.g(n) \leq f(n) \leq c2.g(n)$**, for all $n \geq n0$
- If **$f(n) = o(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq f(n) < c.g(n)$**, for all $n \geq n0$
- If **$f(n) = \omega(g(n))$**, then there exists positive constants c, n0 such that **$0 \leq c.g(n) < f(n)$**, for all $n \geq n0$

O-notation

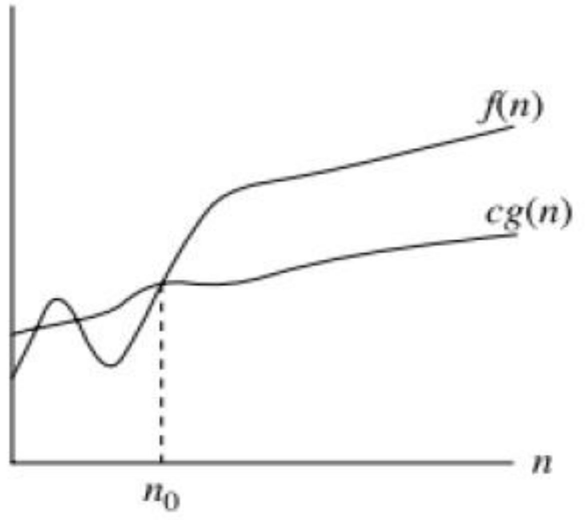
$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}\ .$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Ω-notation

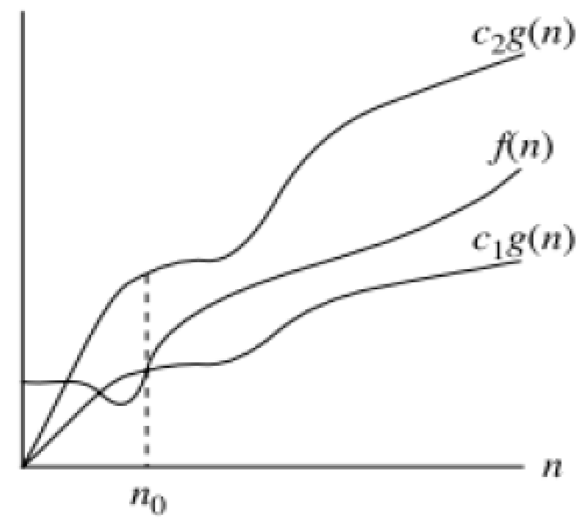
$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Recursive Function

- **Back Substitution Method**
- **Recursive Tree Method**
- Master’s Theorem

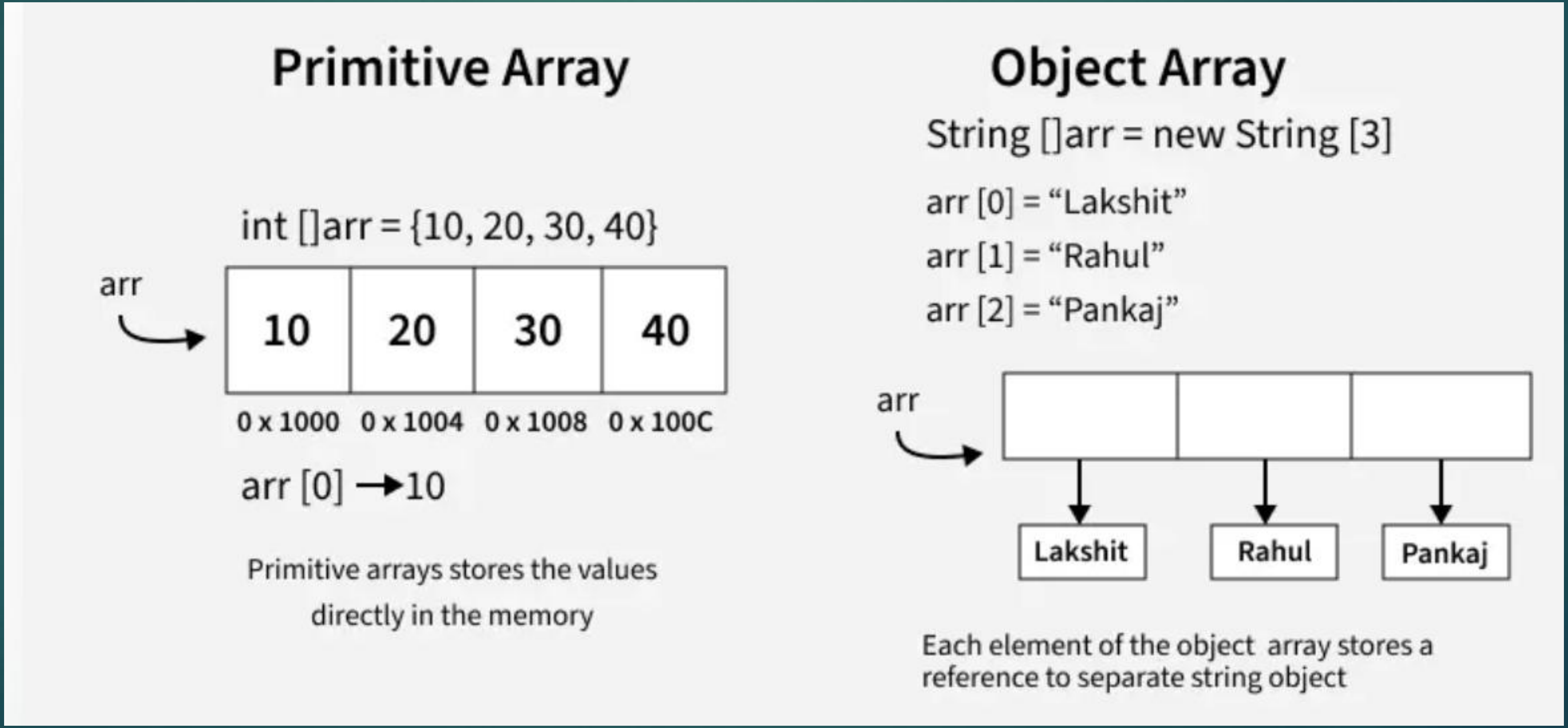
Arrays

- Array is a collection of items of the same variable type that are stored at contiguous memory locations.
- It is one of the most popular and simple data structures used in programming.
- Random Access : i-th item can be accessed in O(1) Time as we have the base address and every item or reference is of same size.
- Cache Friendliness : Since items / references are stored at contiguous locations, we get the advantage of locality of reference.
- It is not useful in places where the operations like insert in the middle, delete from middle and search in a unsorted data.

- **Basic terminologies of Array**
  - **Array Index**: In an array, elements are identified by their indexes. Array index starts from 0.
  - **Array element**: Elements are items stored in an array and can be accessed by their index.
  - **Array Length**: The length of an array is determined by the number of elements it can contain.

➢ Memory representation of Array

- ❑ In an array, all the elements are stored in contiguous memory locations.
- ❑ The elements will be allocated sequentially in memory.
- ❑ This allows for efficient access and manipulation of elements.


Primitive Array
int []arr = {10, 20, 30, 40}
arr
10
20
30
40
0 x 1000
0 x 1004
0 x 1008
0 x 100C
arr [0] →10
Primitive arrays stores the values directly in the memory
Object Array
String []arr = new String [3]
arr [0] = "Lakshit"
arr [1] = "Rahul"
arr [2] = "Pankaj"
arr
Lakshit
Rahul
Pankaj
Each element of the object array stores a reference to separate string object

Memory representation of Array

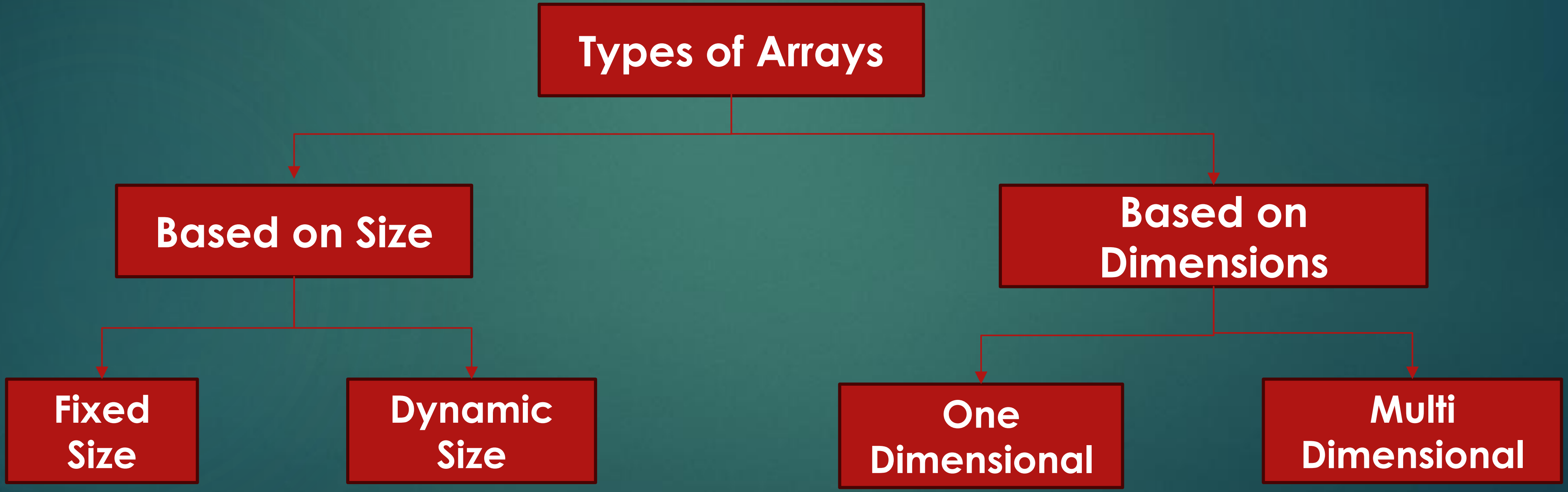
- In an array, all the elements are stored in contiguous memory locations.
- The elements will be allocated sequentially in memory.
- This allows for efficient access and manipulation of elements.

Declaration of Array

- int arr[];   // This array will store integer type element
- char arr[];   // This array will store char type element
- float arr[]; // This array will store float type element

Initialization of Array

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[] = { 'a', 'b', 'c', 'd', 'e' };
float arr[] = { 1.4f, 2.0f, 24f, 5.0f, 0.0f }
```



- **Fixed Sized Arrays:**
  - The size of an array cannot be changed once it has been defined.
  - Memory is allocated based on the size specified within the square brackets during declaration.
  - Allocating more memory than needed wastes space if fewer elements are used.
  - Allocating less memory than required won't be enough to hold all the elements.
  - Static memory allocation is not ideal in situations where the required size is uncertain.
  - Examples:

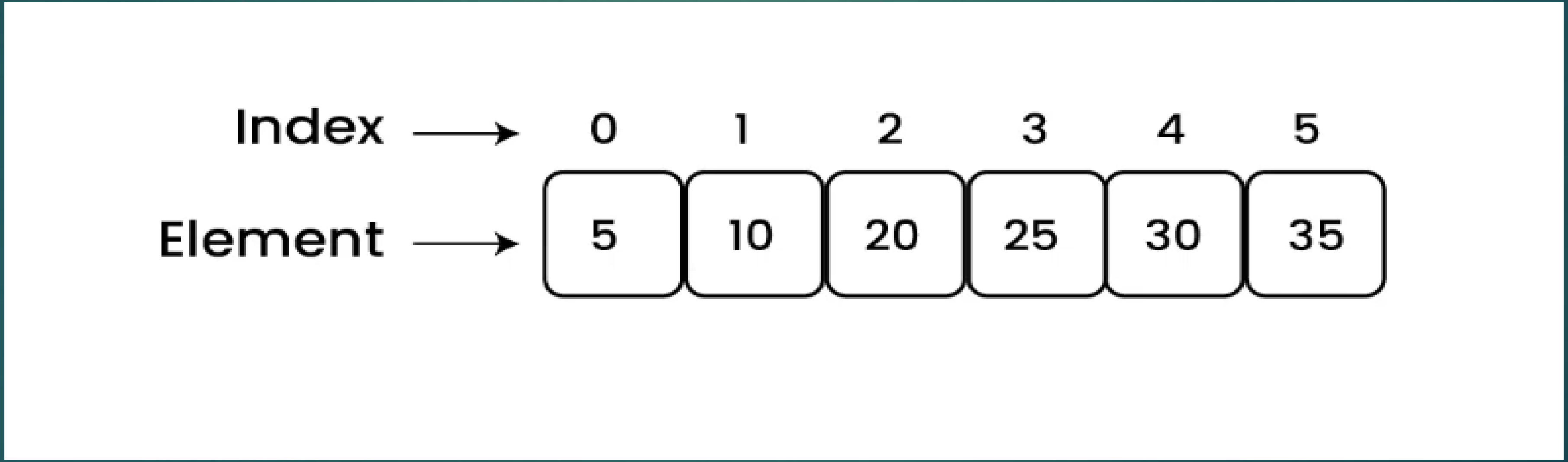
```
int[] arr1 = new int [5];
int[] arr2 = {1, 2, 3, 4, 5};
```

- **Dynamic Sized Arrays:**
  - The array size can change based on user requirements while the program is running.
  - Programmers don't need to be concerned about specifying the exact size in advance.
  - Elements can be added or removed as needed during execution.
  - Examples:

```
ArrayList<Integer> arr = new ArrayList<>();
```

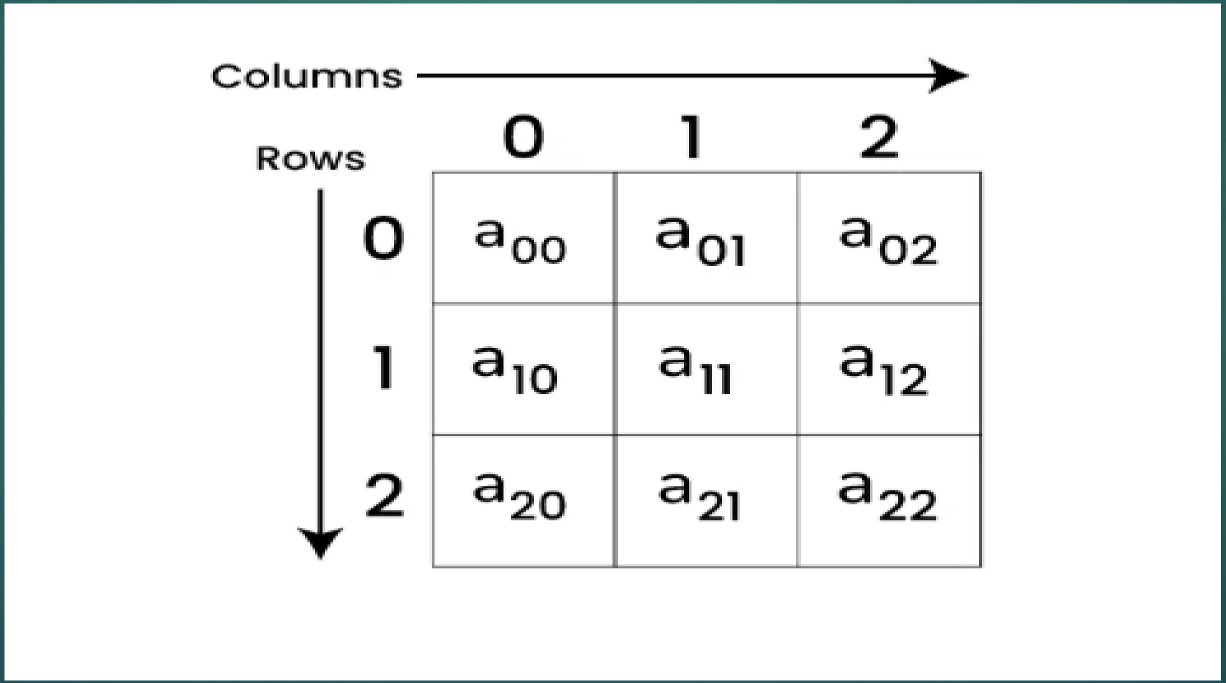
One Dimensional Arrays (1-D Array):

- A one-dimensional array can be visualized as a row where elements are stored consecutively in a linear sequence.



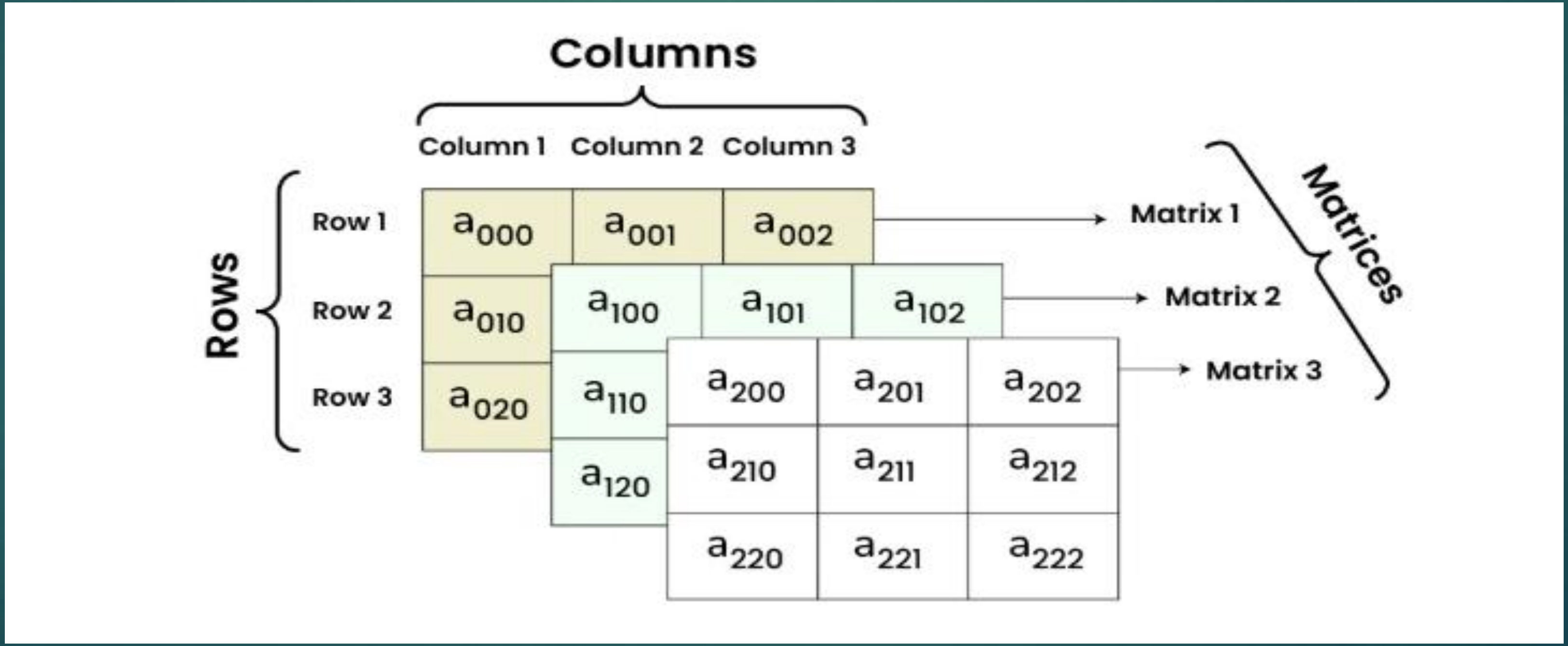
➢ Two Dimensional Arrays (2-D Array or Matrix):

- 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.


Columns
Rows
$$\begin{array}{c|c|c|c} & 0 & 1 & 2 \\ \hline 0 & a_{00} & a_{01} & a_{02} \\ \hline 1 & a_{10} & a_{11} & a_{12} \\ \hline 2 & a_{20} & a_{21} & a_{22} \end{array}$$

Three Dimensional Arrays (3-D Array or Matrix):

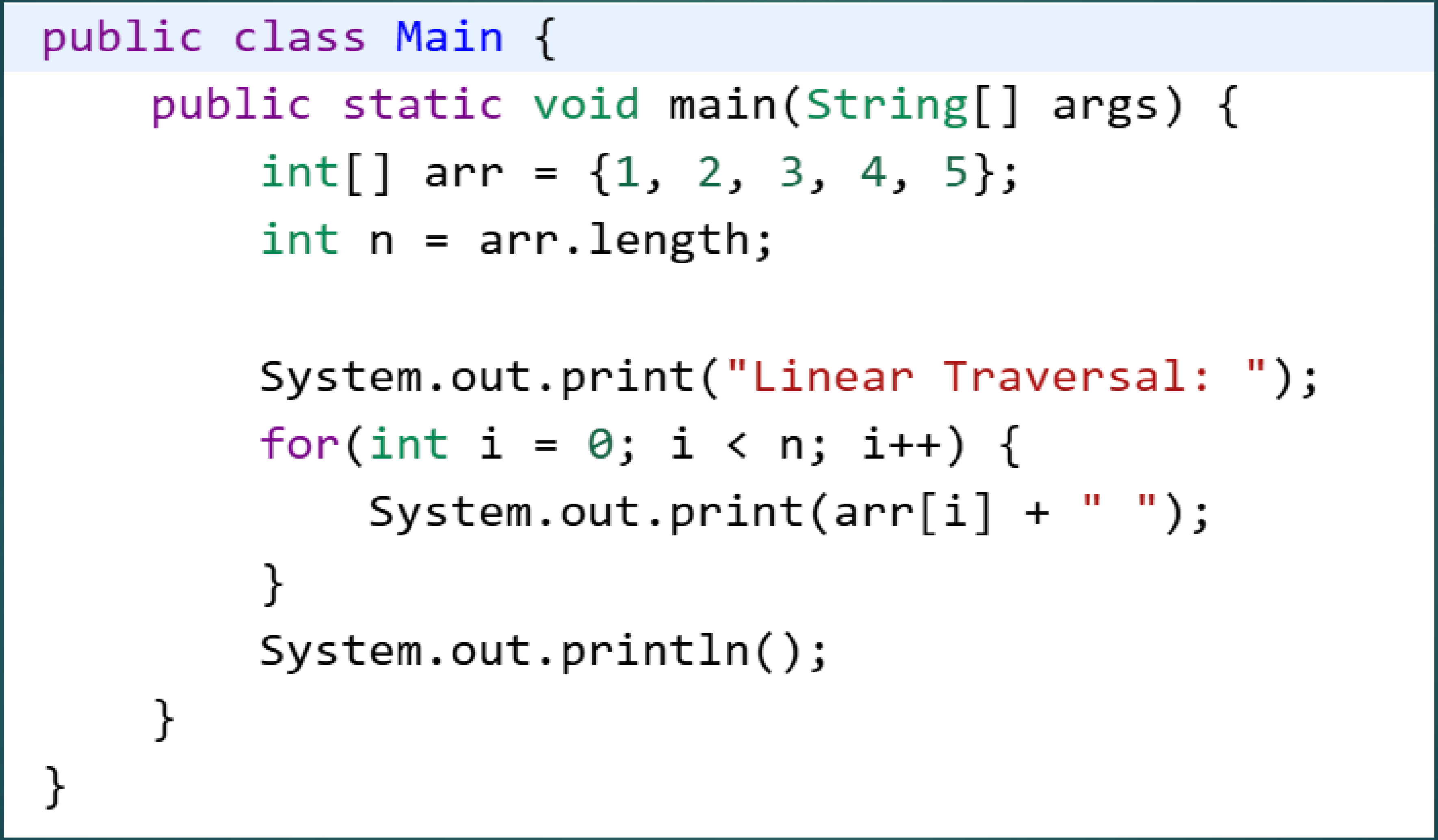
- A 3-D array can be visualized as a collection of 2D arrays stacked together, forming a cube-like structure for storing data in three dimensions.



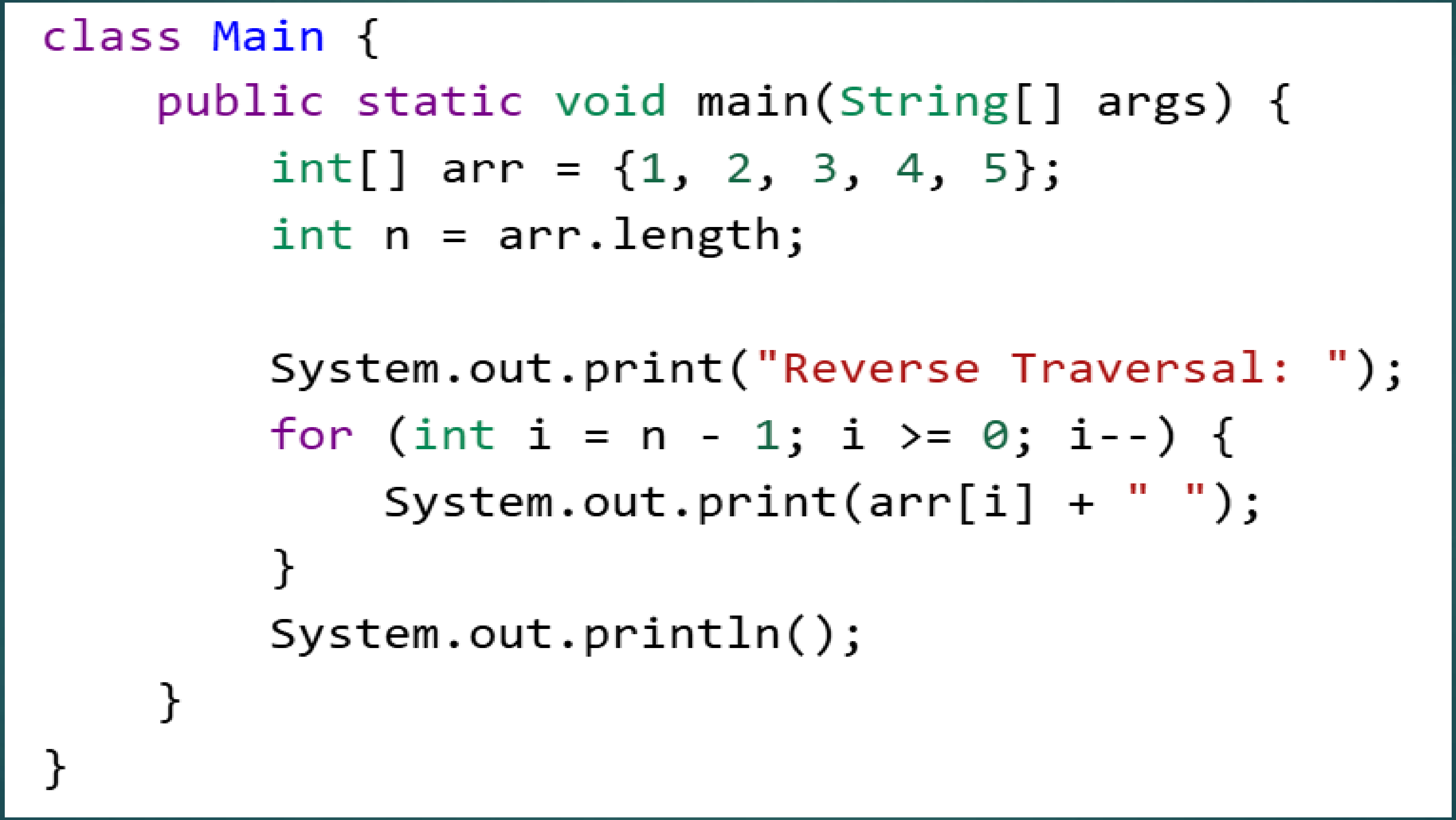
Columns
Column 1
Column 2
Column 3
Rows
Row 1
Row 2
Row 3
a_{000} a_{001} a_{002}
a_{010} a_{020}
a_{100} a_{101} a_{102}
a_{110} a_{120}
a_{200} a_{201} a_{202}
a_{210} a_{211} a_{212}
a_{220} a_{221} a_{222}
Matrix 1
Matrix 2
Matrix 3
Matrices

Operations on Array

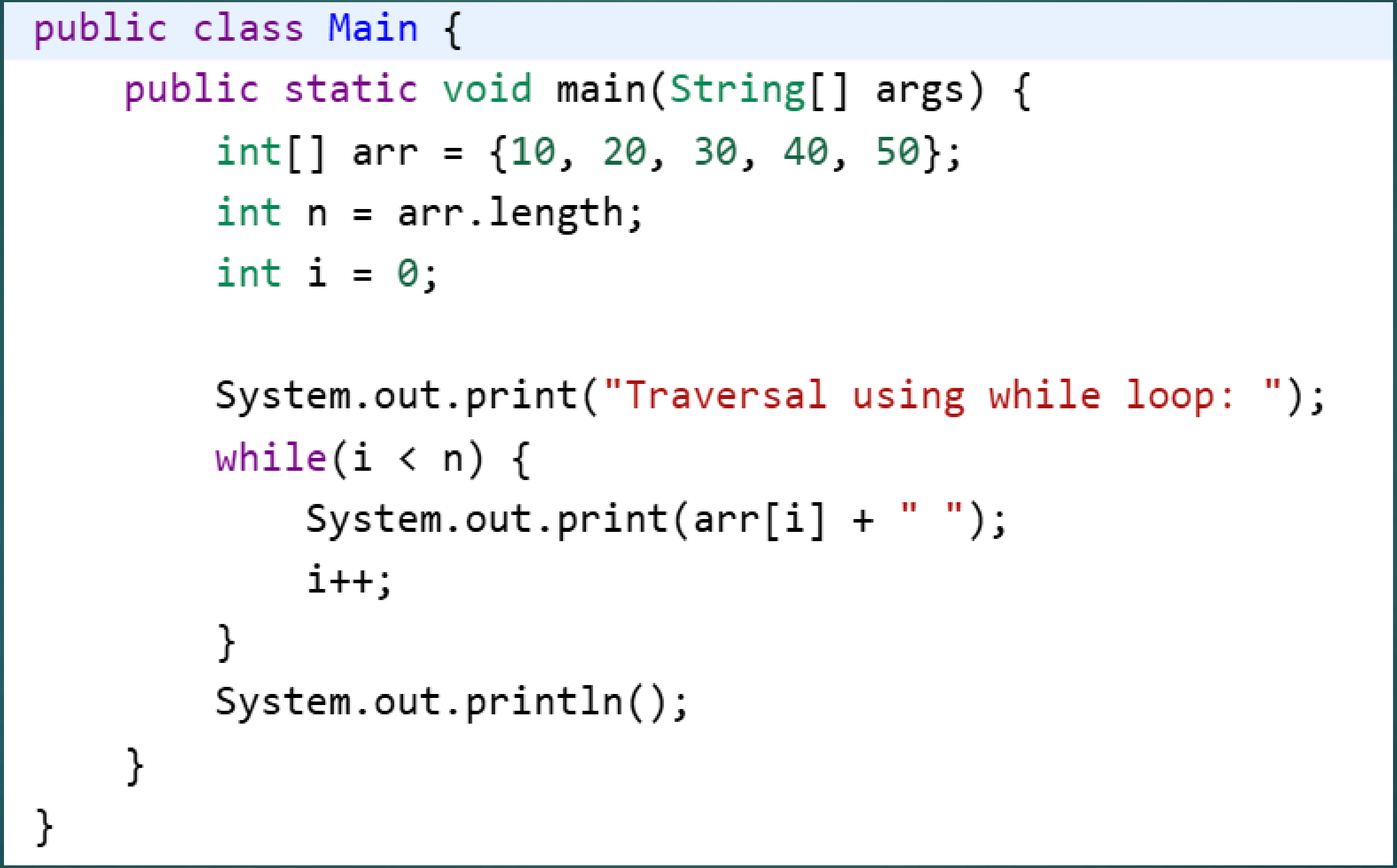
- Array Traversal : The process of accessing and processing each element of an array sequentially.
- Array occupies a contiguous block of memory where elements are stored in an indexed manner.
- Each element can be accessed using its index.
- Types of array Traversal : Sequential (linear), Reverse



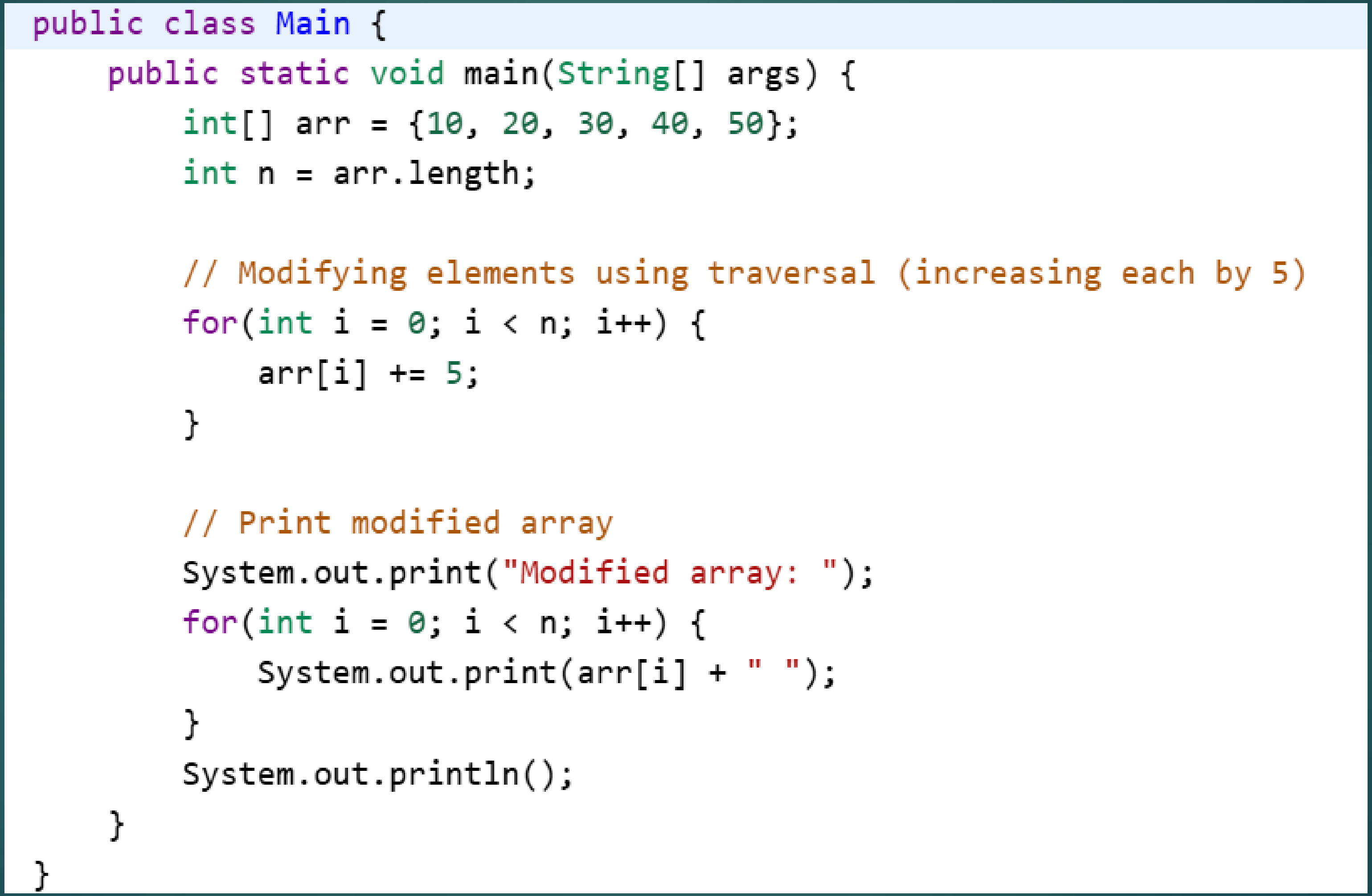
```java
public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;

        System.out.print("Linear Traversal: ");
        for(int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```



```
class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int n = arr.length;

        System.out.print("Reverse Traversal: ");
        for (int i = n - 1; i >= 0; i--) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

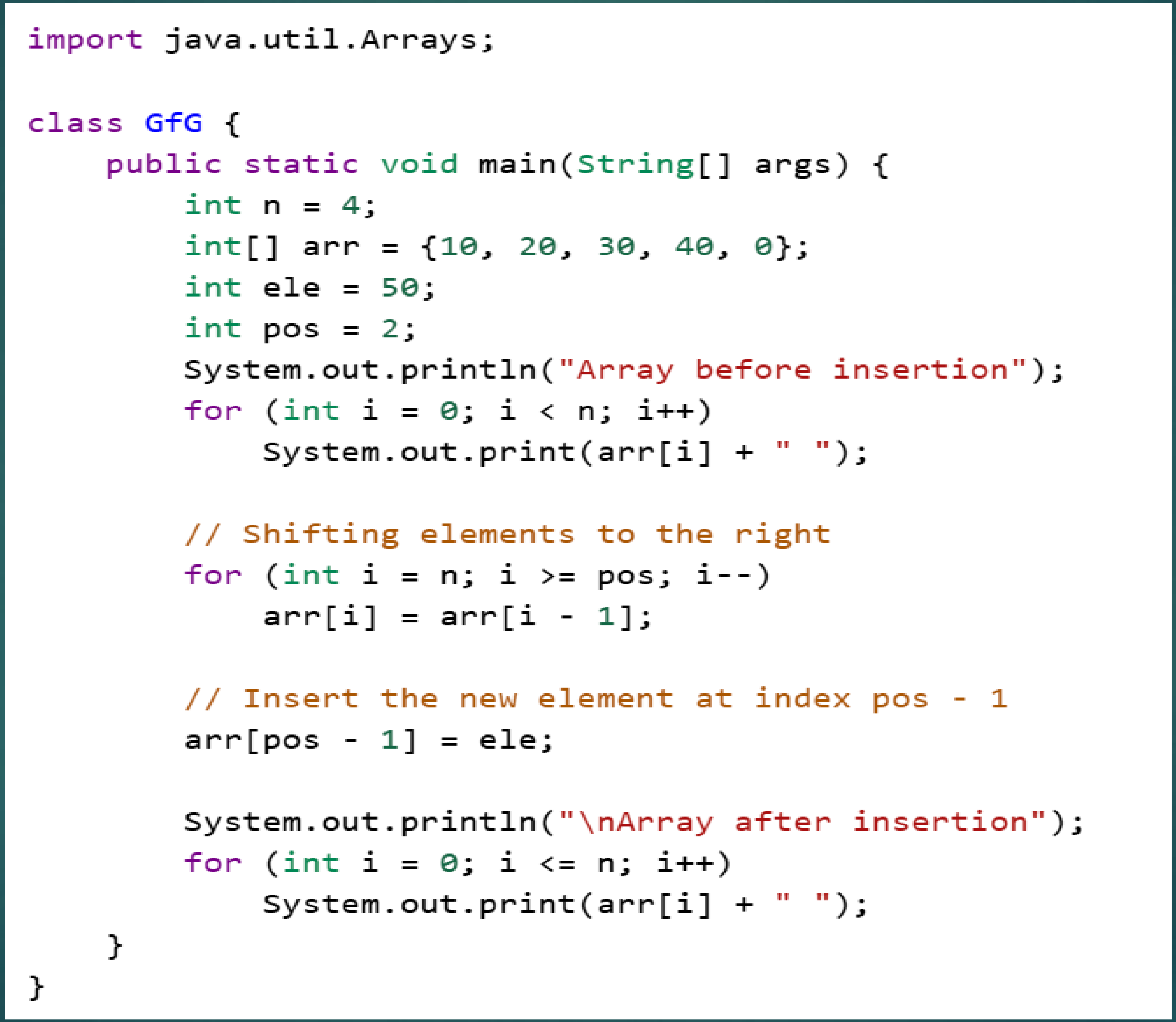


```java
public class Main {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int n = arr.length;
        int i = 0;

        System.out.print("Traversal using while loop: ");
        while(i < n) {
            System.out.print(arr[i] + " ");
            i++;
        }
        System.out.println();
    }
}
```



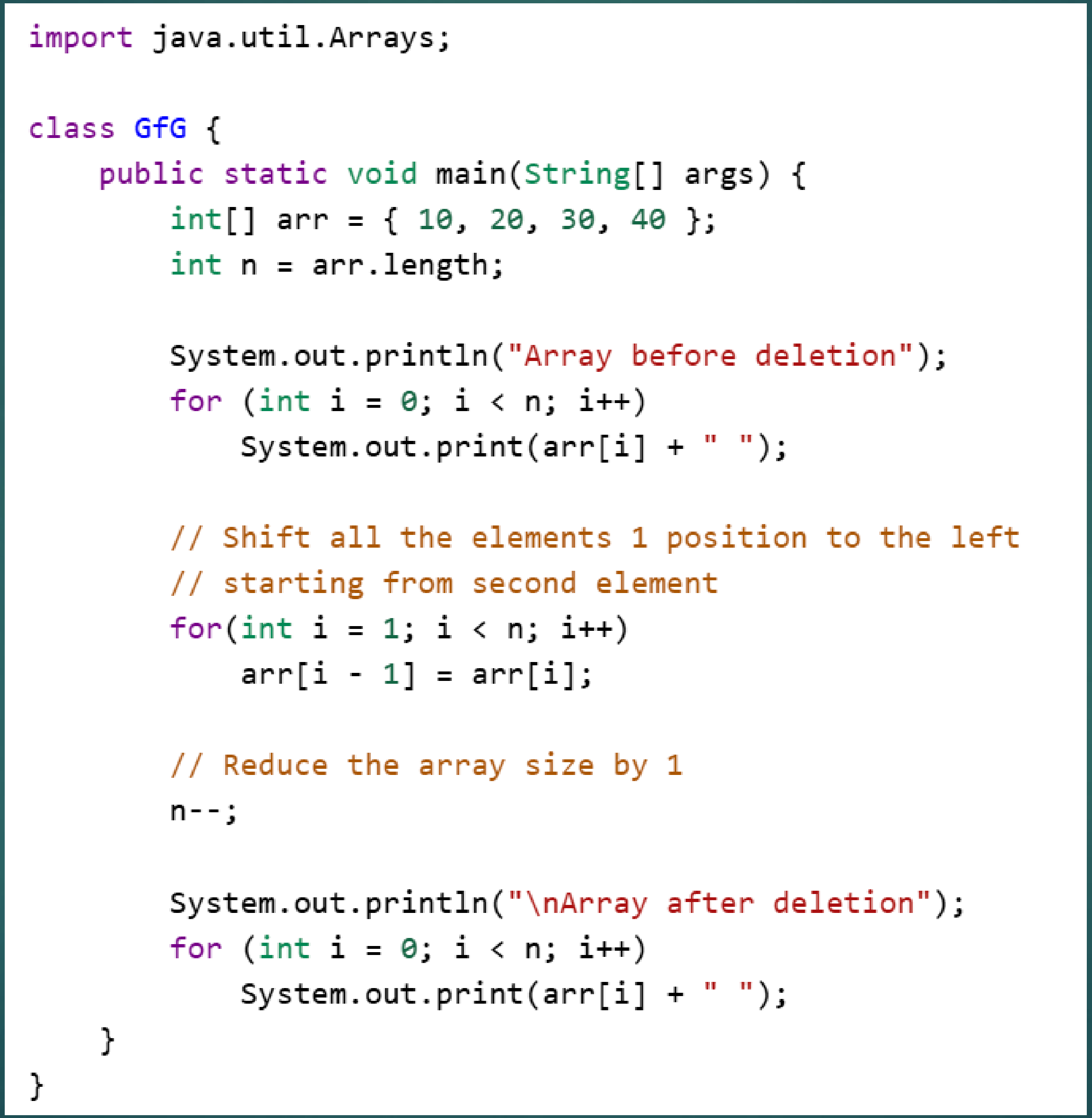
```
public class Main {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int n = arr.length;

        // Modifying elements using traversal (increasing each by 5)
        for(int i = 0; i < n; i++) {
            arr[i] += 5;
        }

        // Print modified array
        System.out.print("Modified array: ");
        for(int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

- Insertion: The process of adding a new element at a specific position while maintaining the order of the existing elements.
  - **Identify the Position**: Determine where the new element should be inserted.
  - **Shift Elements**: Move the existing elements one position forward to create space for the new element.
  - **Insert the New Element**: Place the new value in the correct position.
  - **Update the Size (if applicable)**: If the array is dynamic, its size is increased.
  - Eg: *arr = [10, 20, 30, 40, 50] converted to arr = [10, 20, 25, 30, 40, 50]*
  - *Types of Insertion (beginning, specific position, end)*

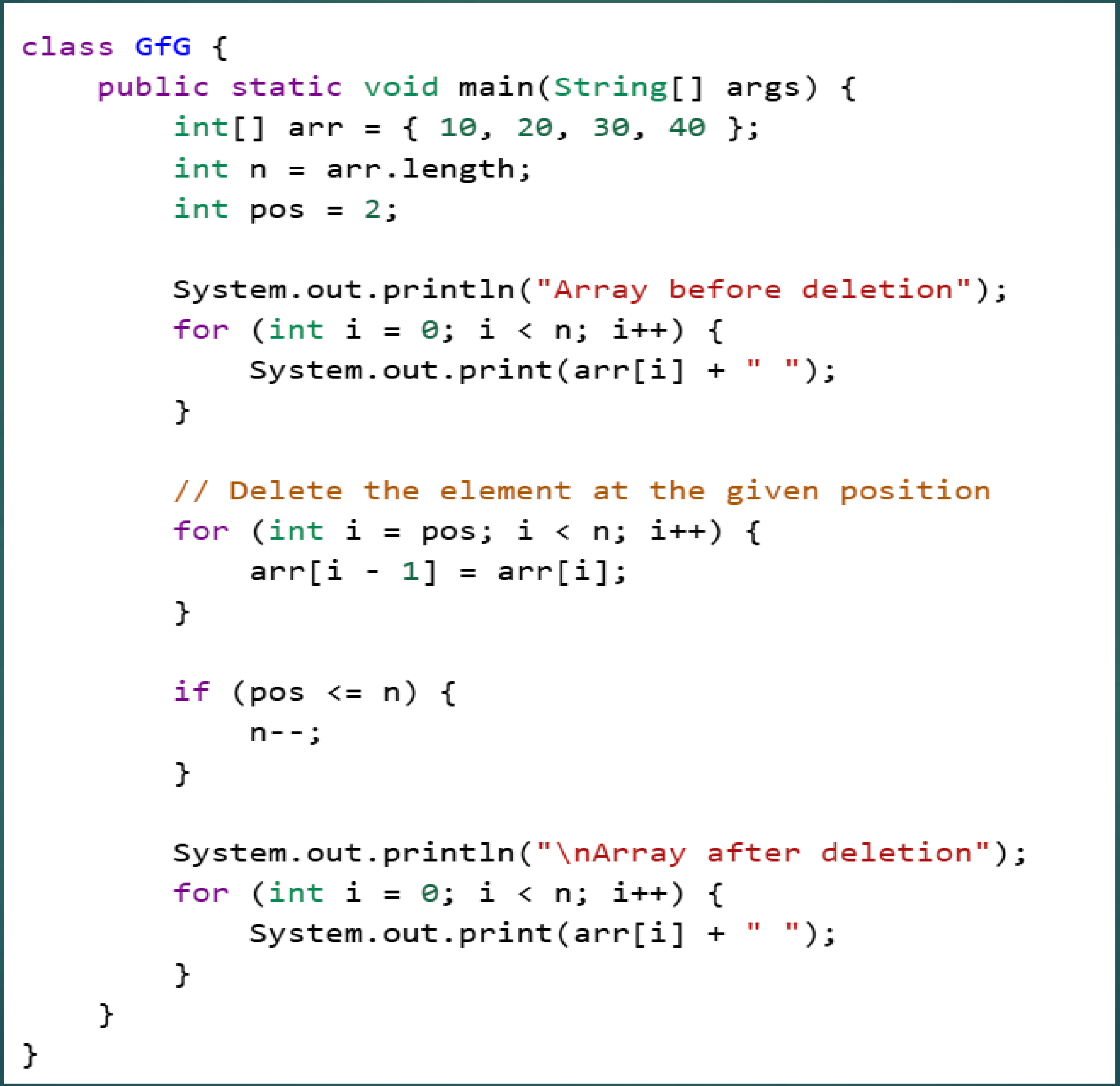


```java
import java.util.Arrays;

class GfG {
    public static void main(String[] args) {
        int n = 4;
        int[] arr = {10, 20, 30, 40, 0};
        int ele = 50;
        int pos = 2;
        System.out.println("Array before insertion");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");

        // Shifting elements to the right
        for (int i = n; i >= pos; i--)
            arr[i] = arr[i - 1];

        // Insert the new element at index pos - 1
        arr[pos - 1] = ele;

        System.out.println("\nArray after insertion");
        for (int i = 0; i <= n; i++)
            System.out.print(arr[i] + " ");
    }
}
```

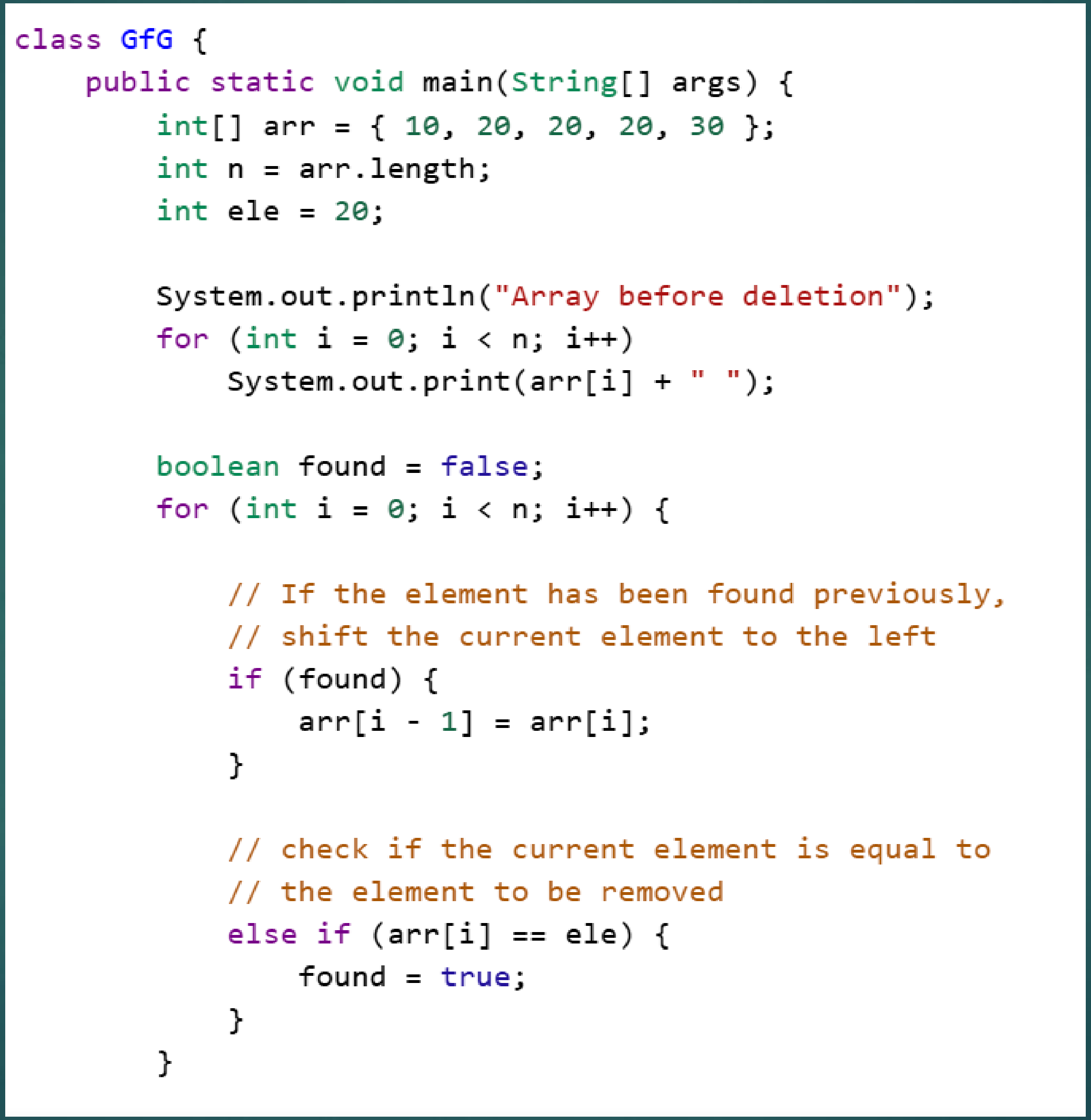
- Deletion: the process of removing an element from a specific position while maintaining the order of the remaining elements.
  - **Identify the Position**: Find the index of the element to be deleted.
  - **Shift Elements**: Move the elements after the deleted element one position to the left.
  - **Update the Size (if applicable)**: If using a dynamic array, the size might be reduced.
  - Types of deletion (Beginning, specific index, end)



```java
import java.util.Arrays;

class GfG {
    public static void main(String[] args) {
        int[] arr = { 10, 20, 30, 40 };
        int n = arr.length;

        System.out.println("Array before deletion");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");

        // Shift all the elements 1 position to the left
        // starting from second element
        for(int i = 1; i < n; i++)
            arr[i - 1] = arr[i];

        // Reduce the array size by 1
        n--;

        System.out.println("\nArray after deletion");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }
}
```



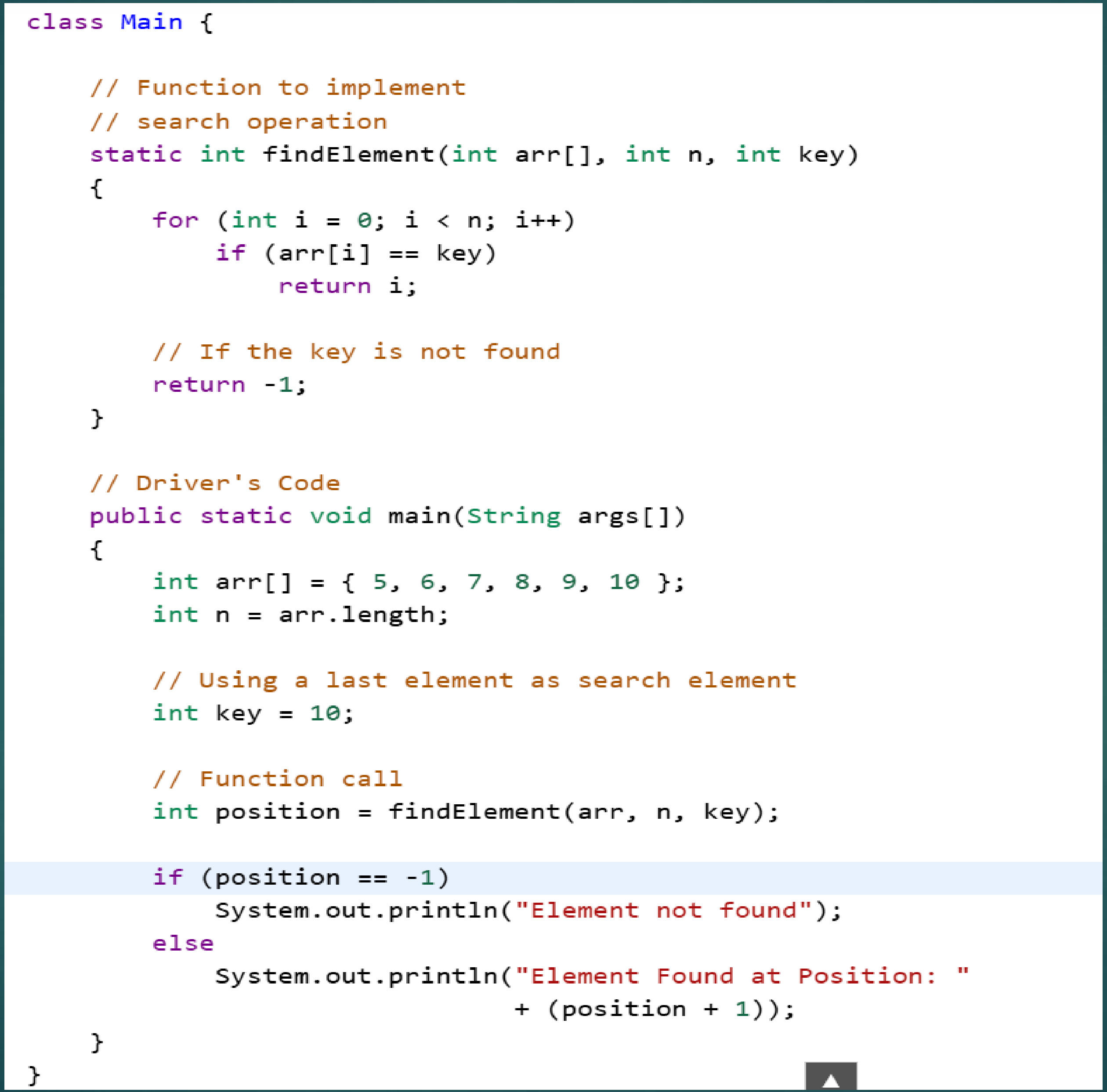
```java
class GfG {
    public static void main(String[] args) {
        int[] arr = { 10, 20, 30, 40 };
        int n = arr.length;
        int pos = 2;

        System.out.println("Array before deletion");
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }

        // Delete the element at the given position
        for (int i = pos; i < n; i++) {
            arr[i - 1] = arr[i];
        }

        if (pos <= n) {
            n--;
        }

        System.out.println("\nArray after deletion");
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```



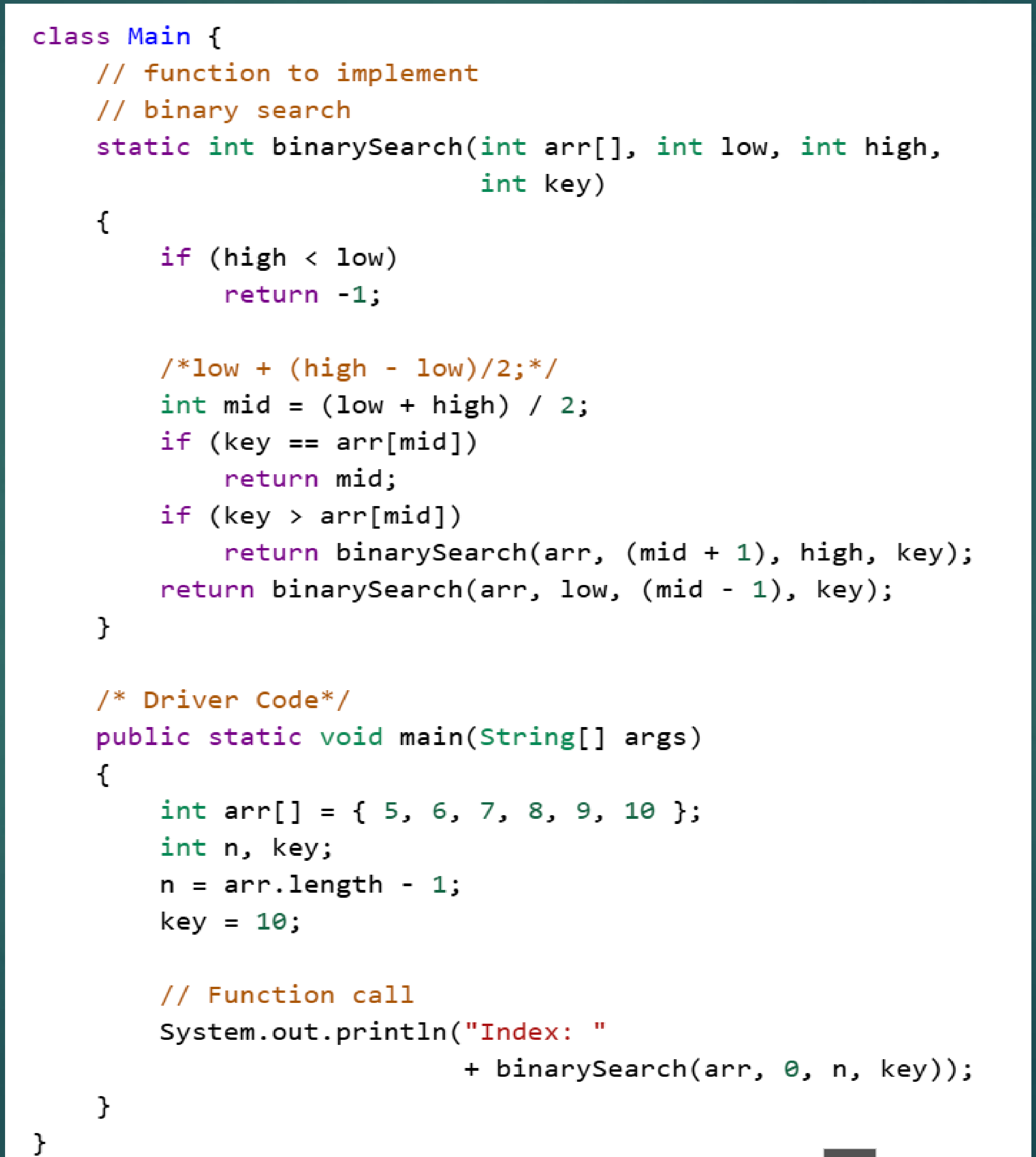
```java
class GfG {
    public static void main(String[] args) {
        int[] arr = { 10, 20, 20, 20, 30 };
        int n = arr.length;
        int ele = 20;

        System.out.println("Array before deletion");
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");

        boolean found = false;
        for (int i = 0; i < n; i++) {

            // If the element has been found previously,
            // shift the current element to the left
            if (found) {
                arr[i - 1] = arr[i];
            }

            // check if the current element is equal to
            // the element to be removed
            else if (arr[i] == ele) {
                found = true;
            }
        }
```

- Searching: the process of finding a specific element in a given list of elements.
- Linear Search
  - This is the simplest search algorithm.
  - It traverses the array one element at a time and compares each element with the target value.
  - If a match is found, it returns the index of the element.
  - If the element is not found, the search continues until the end of the array.

- Binary Search
  - Works only on sorted arrays (in increasing or decreasing order).
  - Uses a divide and conquer approach.
  - It repeatedly divides the search space in half until the target element is found.
  - How it works???
    - Find the middle element of the array.
    - If the target is equal to the middle element, return its index.
    - If the target is less than the middle element, search the left half.
    - If the target is greater than the middle element, search the right half.
    - Repeat until the element is found or the search space is empty.



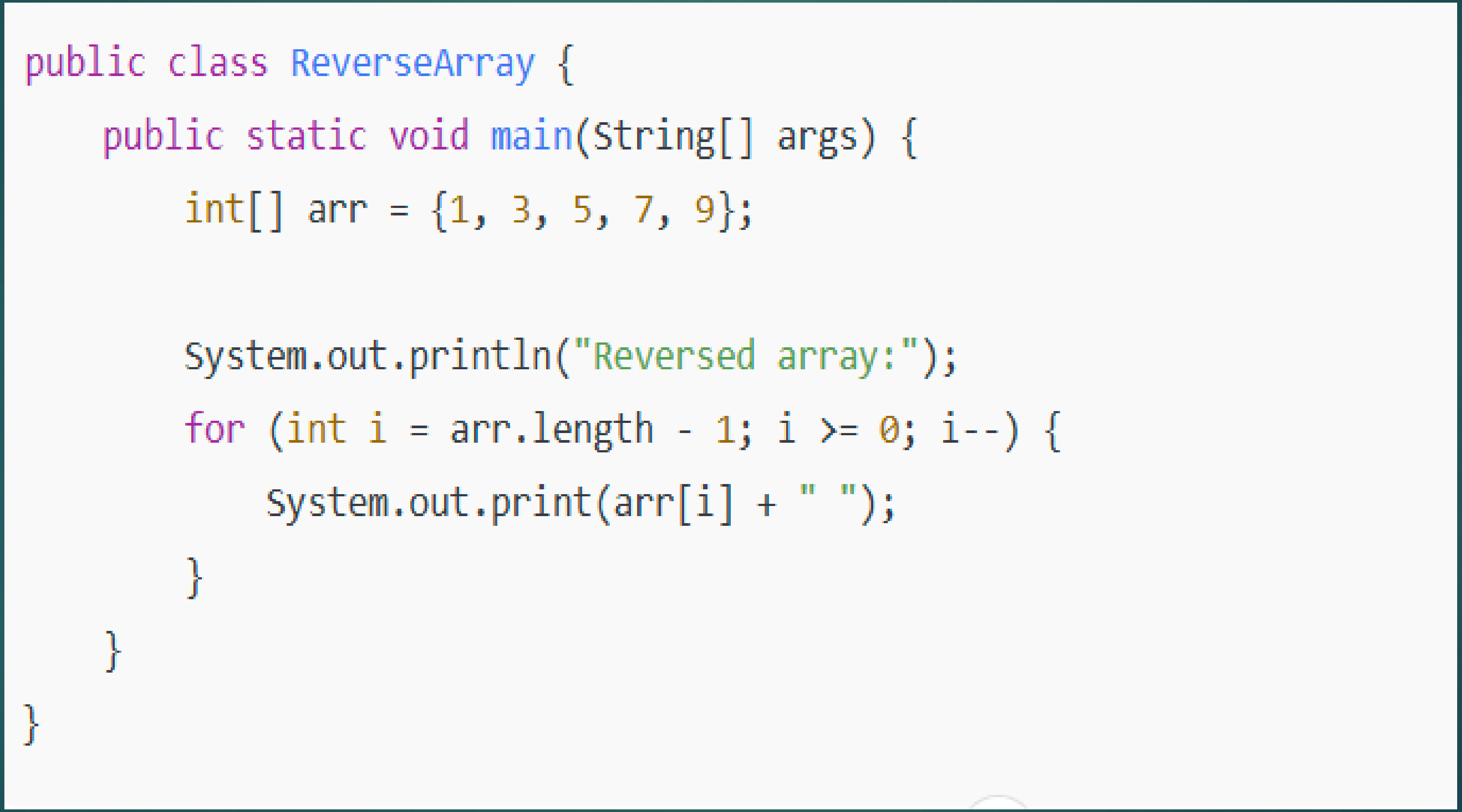
```java
class Main {

    // Function to implement
    // search operation
    static int findElement(int arr[], int n, int key)
    {
        for (int i = 0; i < n; i++)
            if (arr[i] == key)
                return i;

        // If the key is not found
        return -1;
    }

    // Driver's Code
    public static void main(String args[])
    {
        int arr[] = { 5, 6, 7, 8, 9, 10 };
        int n = arr.length;

        // Using a last element as search element
        int key = 10;

        // Function call
        int position = findElement(arr, n, key);

        if (position == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element Found at Position: "
                               + (position + 1));
    }
}
```



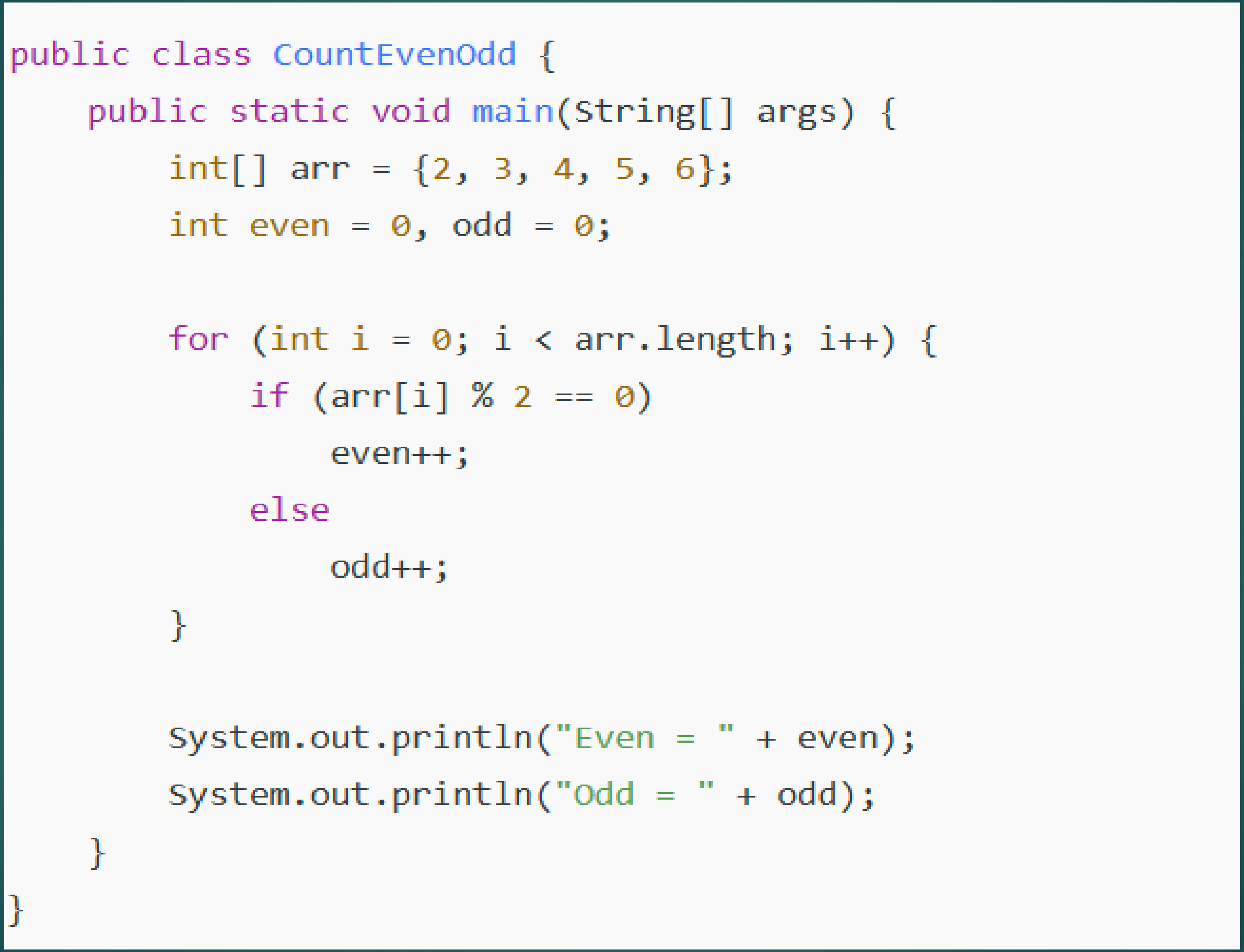
```
class Main {
    // function to implement
    // binary search
    static int binarySearch(int arr[], int low, int high,
                            int key)
    {
        if (high < low)
            return -1;

        /*low + (high - low)/2;*/
        int mid = (low + high) / 2;
        if (key == arr[mid])
            return mid;
        if (key > arr[mid])
            return binarySearch(arr, (mid + 1), high, key);
        return binarySearch(arr, low, (mid - 1), key);
    }

    /* Driver Code*/
    public static void main(String[] args)
    {
        int arr[] = { 5, 6, 7, 8, 9, 10 };
        int n, key;
        n = arr.length - 1;
        key = 10;

        // Function call
        System.out.println("Index: "
                           + binarySearch(arr, 0, n, key));
    }
}
```

- Applications of Array
  - Storing and accessing data
  - Searching
  - Matrices
  - Implementing other data structures
  - Dynamic programming
  - Data Buffers

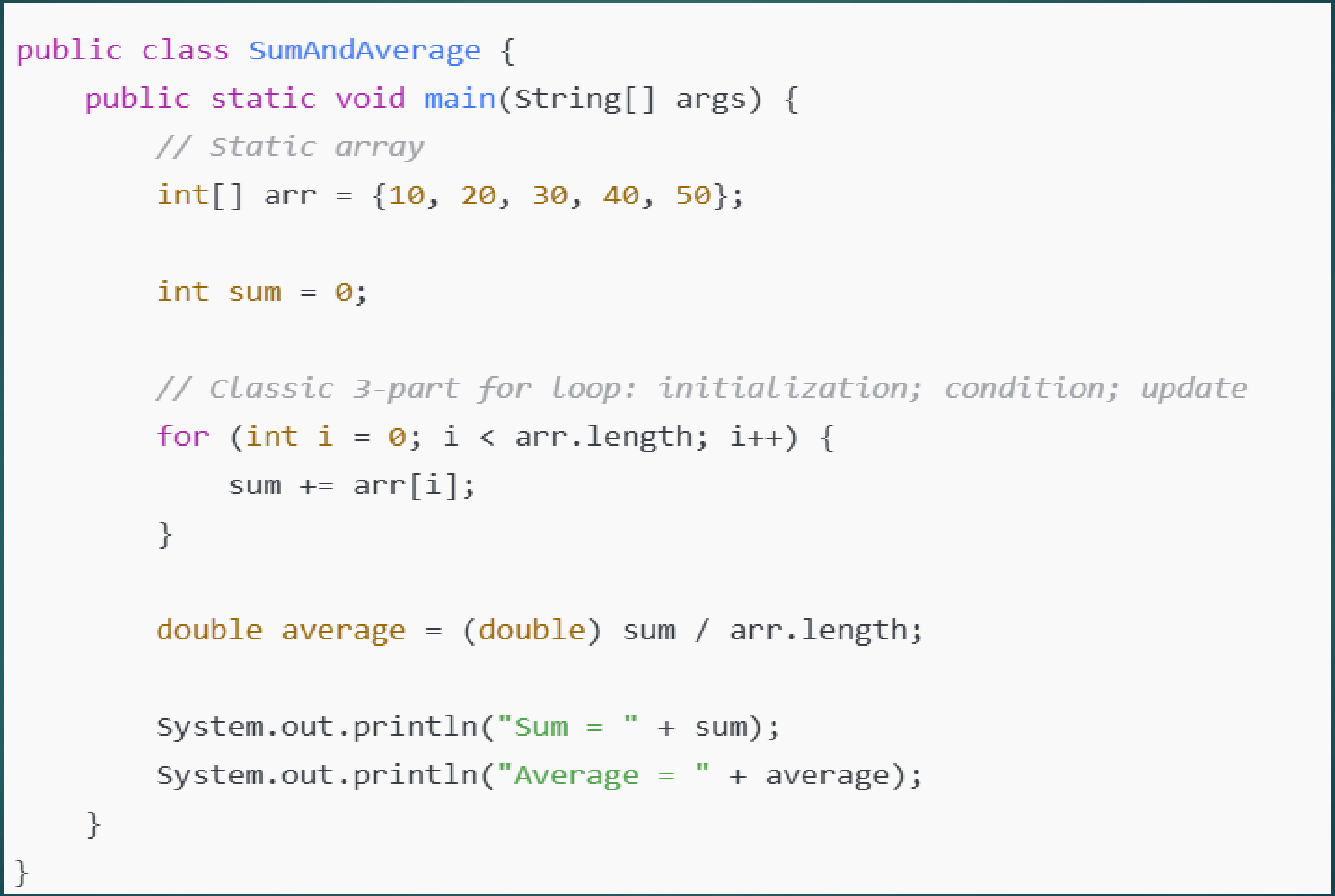
- Advantages of Array
  - Efficient and fast access
  - Memory efficiency
  - Versatility
  - Compatibility with hardware
- Disadvantages of Array
  - Fixed size
  - Memory allocation issues
  - Insertion and deletion challenges
  - Limited data type
  - Lack of flexibility



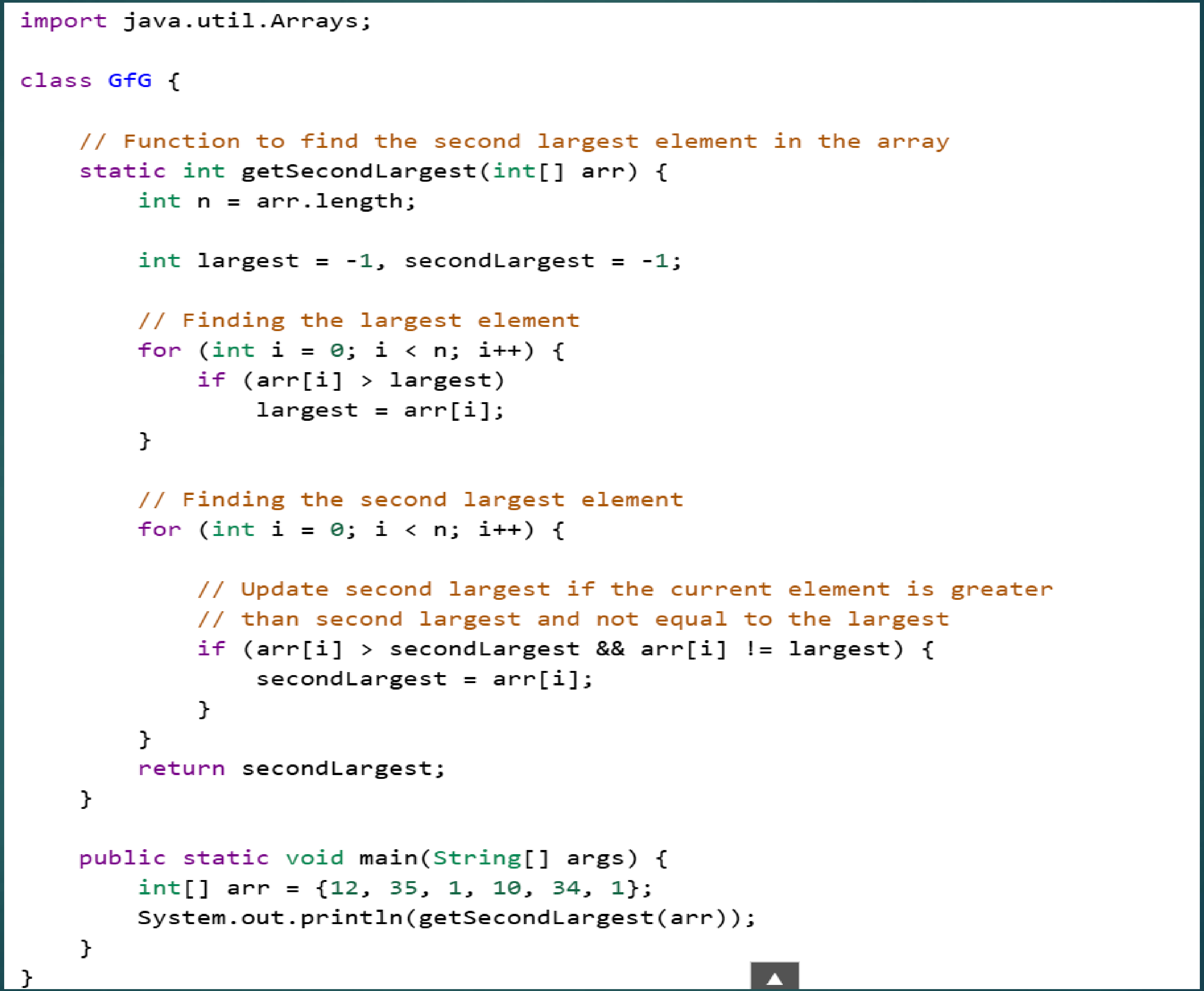
```
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9};

        System.out.println("Reversed array:");
        for (int i = arr.length - 1; i >= 0; i--) {
            System.out.print(arr[i] + " ");
        }
    }
}
```



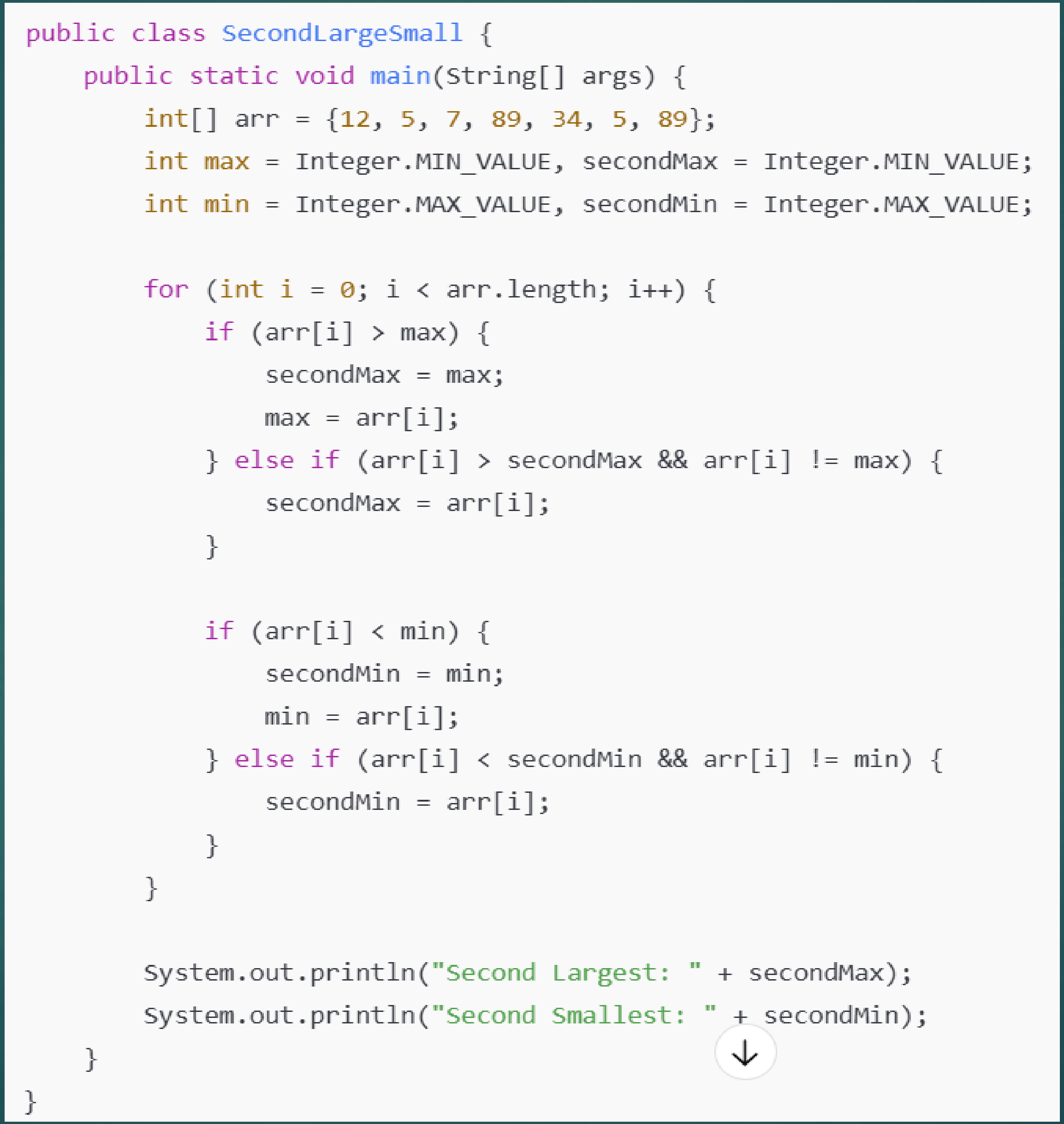
```java
public class CountEvenOdd {
    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 5, 6};
        int even = 0, odd = 0;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] % 2 == 0)
                even++;
            else
                odd++;
        }

        System.out.println("Even = " + even);
        System.out.println("Odd = " + odd);
    }
}
```



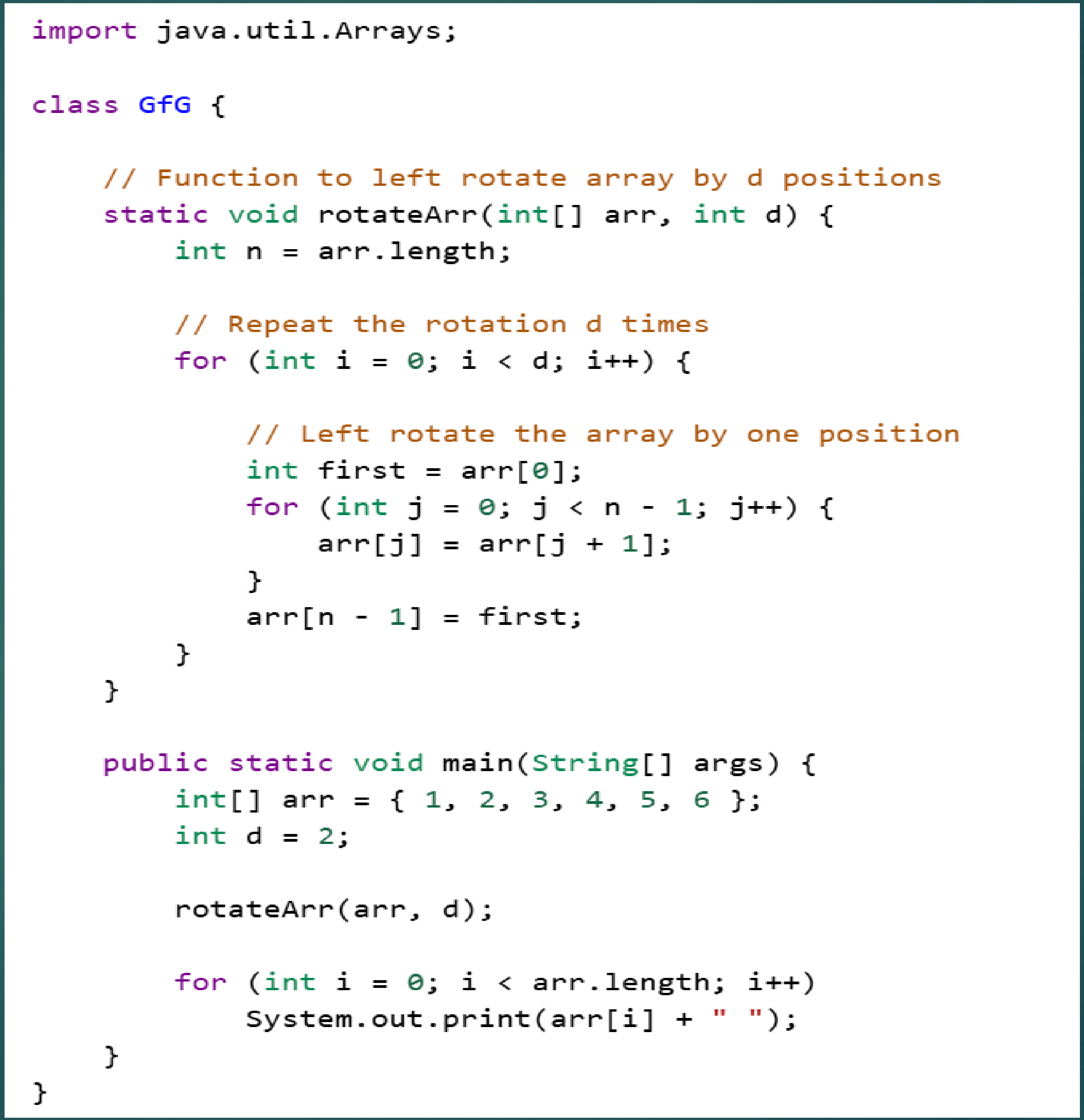
```java
public class SumAndAverage {
    public static void main(String[] args) {
        // Static array
        int[] arr = {10, 20, 30, 40, 50};

        int sum = 0;

        // Classic 3-part for loop: initialization; condition; update
        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }

        double average = (double) sum / arr.length;

        System.out.println("Sum = " + sum);
        System.out.println("Average = " + average);
    }
}
```



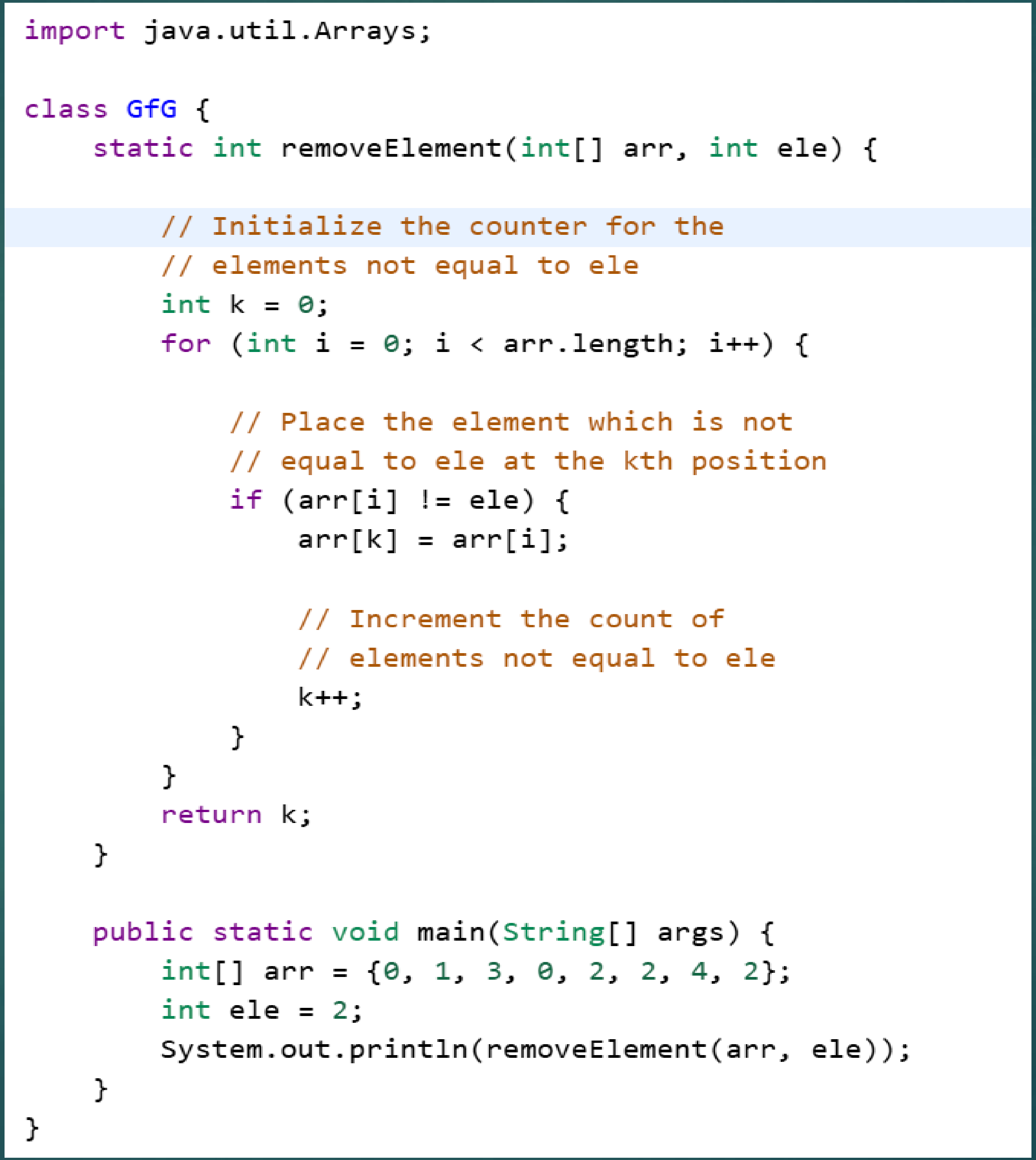
```java
import java.util.Arrays;

class GfG {

    // Function to find the second largest element in the array
    static int getSecondLargest(int[] arr) {
        int n = arr.length;

        int largest = -1, secondLargest = -1;

        // Finding the largest element
        for (int i = 0; i < n; i++) {
            if (arr[i] > largest)
                largest = arr[i];
        }

        // Finding the second largest element
        for (int i = 0; i < n; i++) {

            // Update second largest if the current element is greater
            // than second largest and not equal to the largest
            if (arr[i] > secondLargest && arr[i] != largest) {
                secondLargest = arr[i];
            }
        }
        return secondLargest;
    }

    public static void main(String[] args) {
        int[] arr = {12, 35, 1, 10, 34, 1};
        System.out.println(getSecondLargest(arr));
    }
}
```



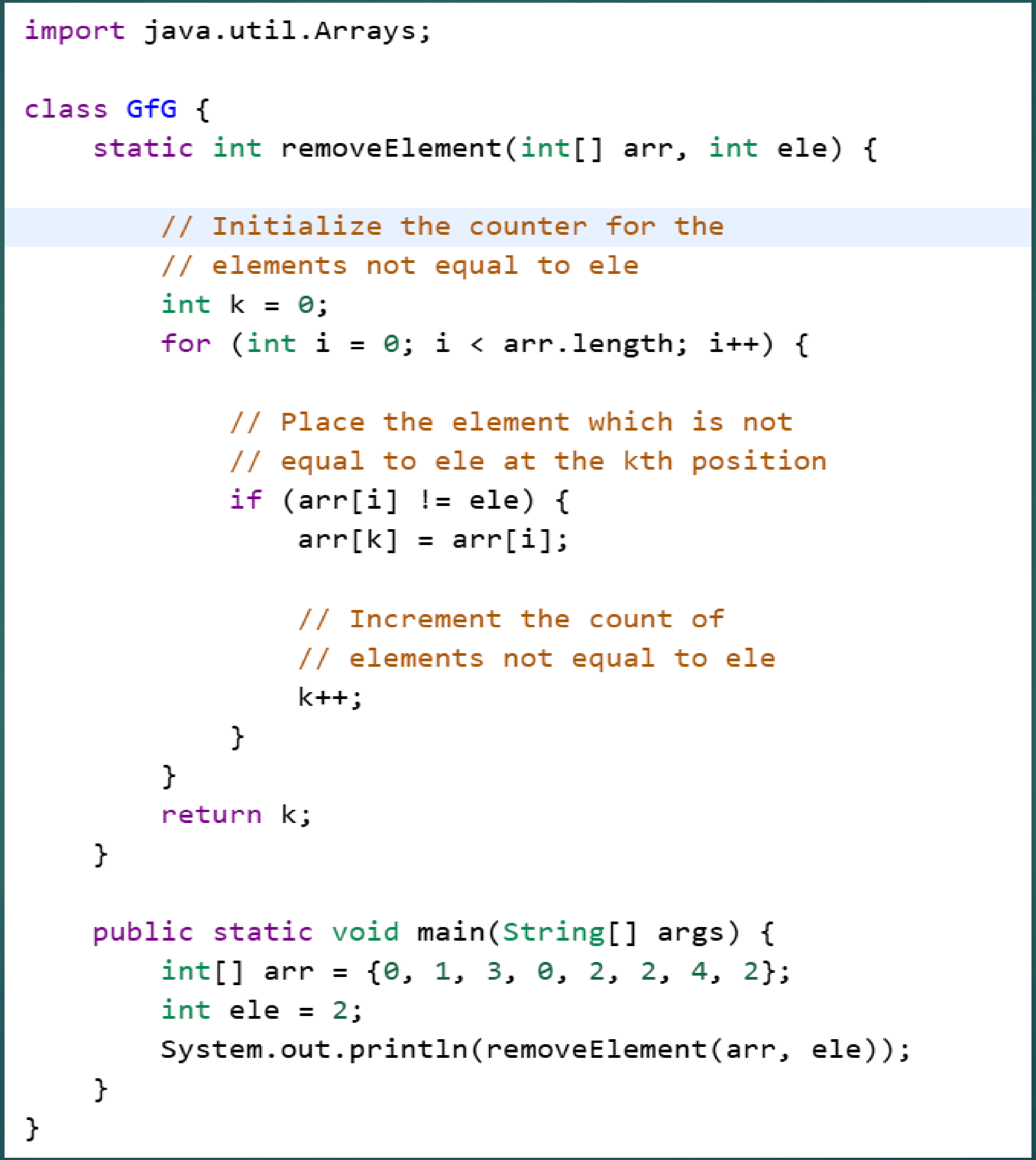
```java
public class SecondLargeSmall {
    public static void main(String[] args) {
        int[] arr = {12, 5, 7, 89, 34, 5, 89};
        int max = Integer.MIN_VALUE, secondMax = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE, secondMin = Integer.MAX_VALUE;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] > max) {
                secondMax = max;
                max = arr[i];
            } else if (arr[i] > secondMax && arr[i] != max) {
                secondMax = arr[i];
            }

            if (arr[i] < min) {
                secondMin = min;
                min = arr[i];
            } else if (arr[i] < secondMin && arr[i] != min) {
                secondMin = arr[i];
            }
        }

        System.out.println("Second Largest: " + secondMax);
        System.out.println("Second Smallest: " + secondMin);
    }
}
```



```java
import java.util.Arrays;

class GfG {

    // Function to left rotate array by d positions
    static void rotateArr(int[] arr, int d) {
        int n = arr.length;

        // Repeat the rotation d times
        for (int i = 0; i < d; i++) {

            // Left rotate the array by one position
            int first = arr[0];
            for (int j = 0; j < n - 1; j++) {
                arr[j] = arr[j + 1];
            }
            arr[n - 1] = first;
        }
    }

    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5, 6 };
        int d = 2;

        rotateArr(arr, d);

        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
```

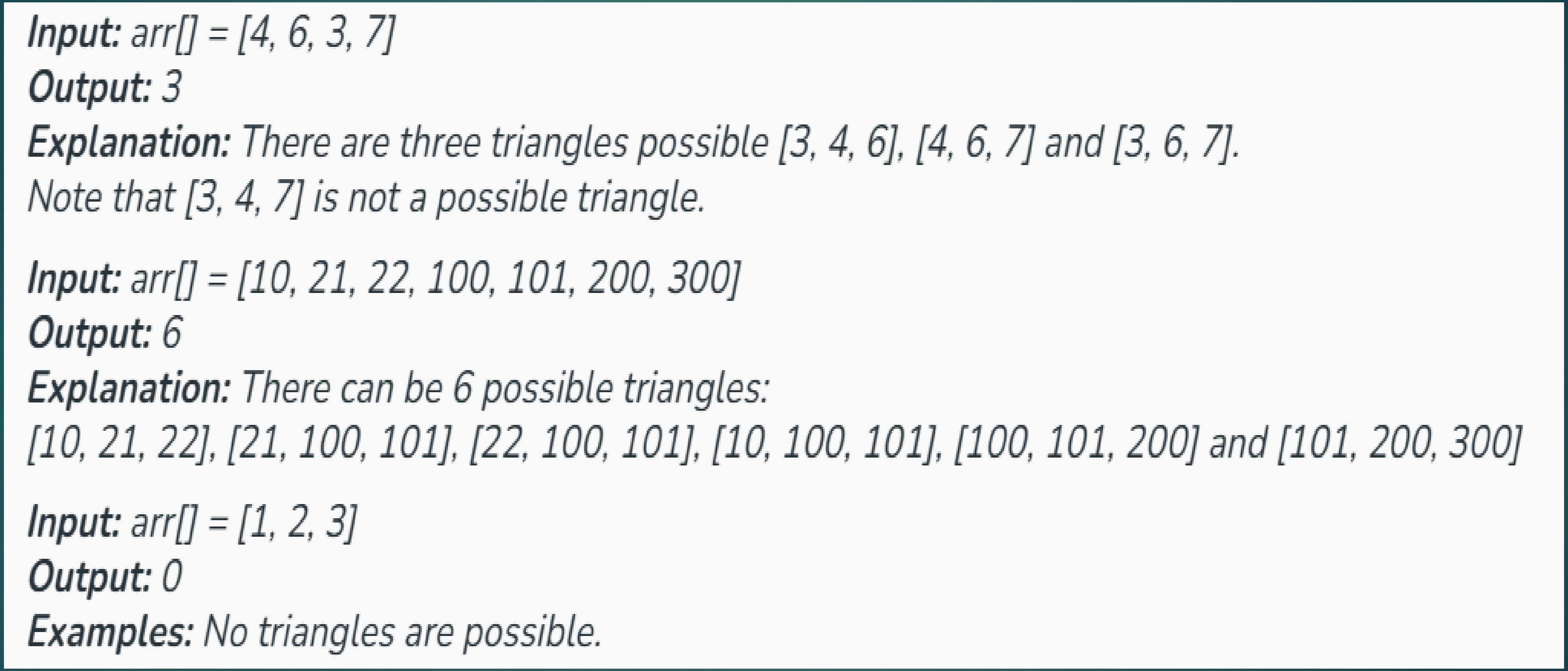


```java
import java.util.Arrays;

class GfG {
    static int removeElement(int[] arr, int ele) {

        // Initialize the counter for the
        // elements not equal to ele
        int k = 0;
        for (int i = 0; i < arr.length; i++) {

            // Place the element which is not
            // equal to ele at the kth position
            if (arr[i] != ele) {
                arr[k] = arr[i];

                // Increment the count of
                // elements not equal to ele
                k++;
            }
        }
        return k;
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 3, 0, 2, 2, 4, 2};
        int ele = 2;
        System.out.println(removeElement(arr, ele));
    }
}
```



```java
import java.util.Arrays;

class GfG {
    static int removeElement(int[] arr, int ele) {

        // Initialize the counter for the
        // elements not equal to ele
        int k = 0;
        for (int i = 0; i < arr.length; i++) {

            // Place the element which is not
            // equal to ele at the kth position
            if (arr[i] != ele) {
                arr[k] = arr[i];

                // Increment the count of
                // elements not equal to ele
                k++;
            }
        }
        return k;
    }

    public static void main(String[] args) {
        int[] arr = {0, 1, 3, 0, 2, 2, 4, 2};
        int ele = 2;
        System.out.println(removeElement(arr, ele));
    }
}
```

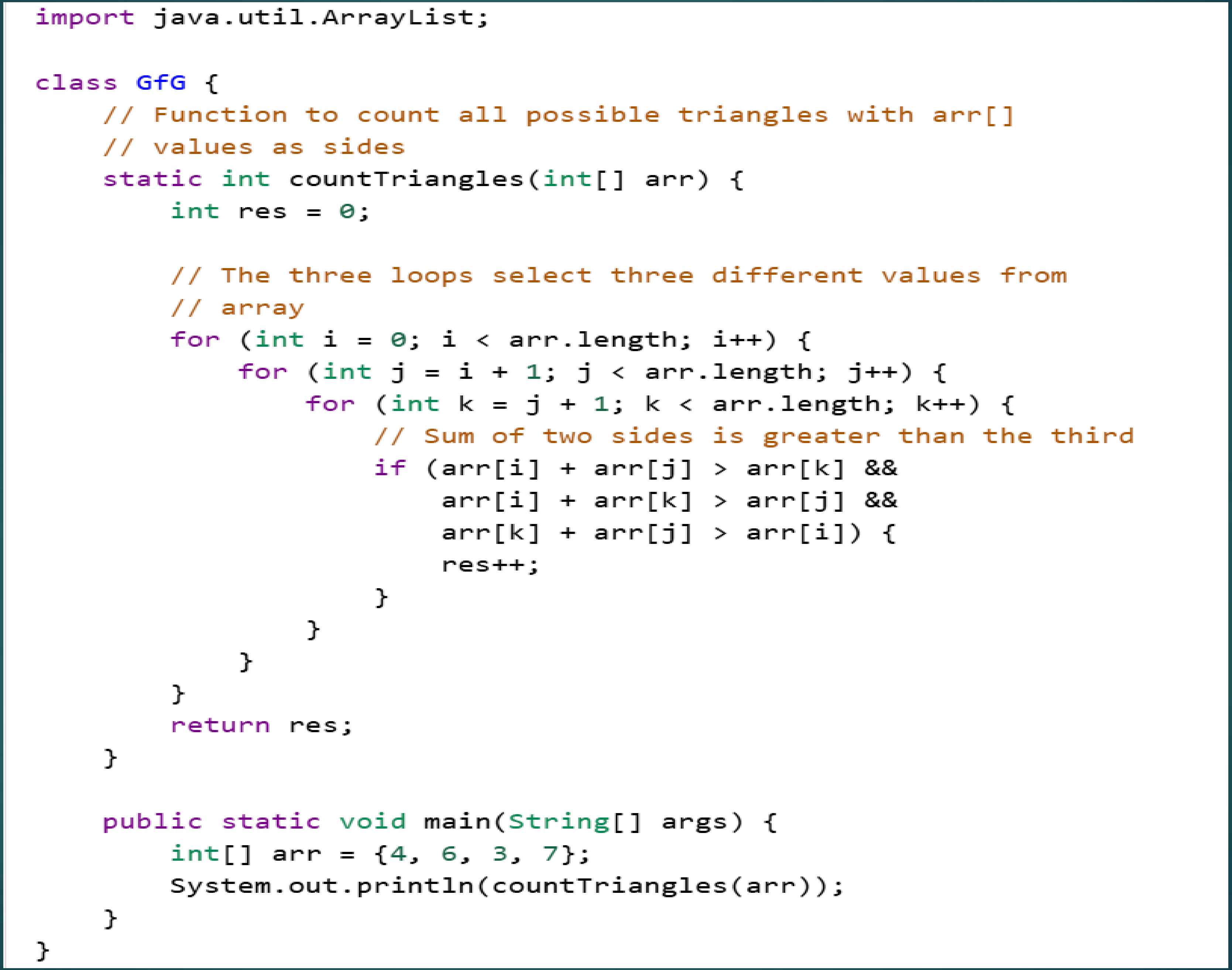
Count possible triangles



Input: *arr[] = [4, 6, 3, 7]*
Output: *3*
Explanation: *There are three triangles possible [3, 4, 6], [4, 6, 7] and [3, 6, 7].*
Note that [3, 4, 7] is not a possible triangle.

Input: *arr[] = [10, 21, 22, 100, 101, 200, 300]*
Output: *6*
Explanation: *There can be 6 possible triangles:*
[10, 21, 22], [21, 100, 101], [22, 100, 101], [10, 100, 101], [100, 101, 200] and [101, 200, 300]

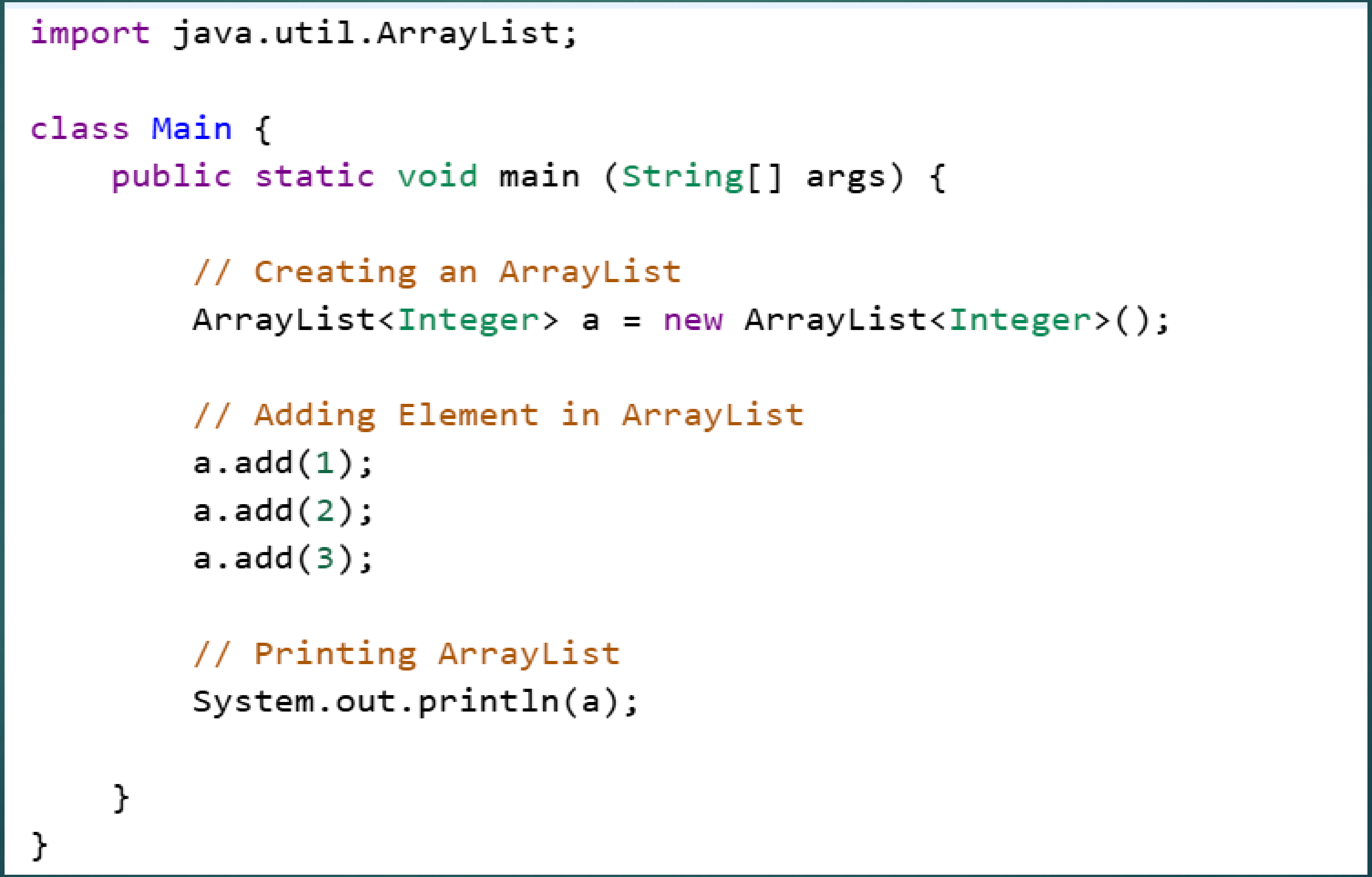
Input: *arr[] = [1, 2, 3]*
Output: *0*
Examples: *No triangles are possible.*



```java
import java.util.ArrayList;

class GfG {
    // Function to count all possible triangles with arr[]
    // values as sides
    static int countTriangles(int[] arr) {
        int res = 0;

        // The three loops select three different values from
        // array
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                for (int k = j + 1; k < arr.length; k++) {
                    // Sum of two sides is greater than the third
                    if (arr[i] + arr[j] > arr[k] &&
                        arr[i] + arr[k] > arr[j] &&
                        arr[k] + arr[j] > arr[i]) {
                        res++;
                    }
                }
            }
        }
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {4, 6, 3, 7};
        System.out.println(countTriangles(arr));
    }
}
```

ArrayList in Java

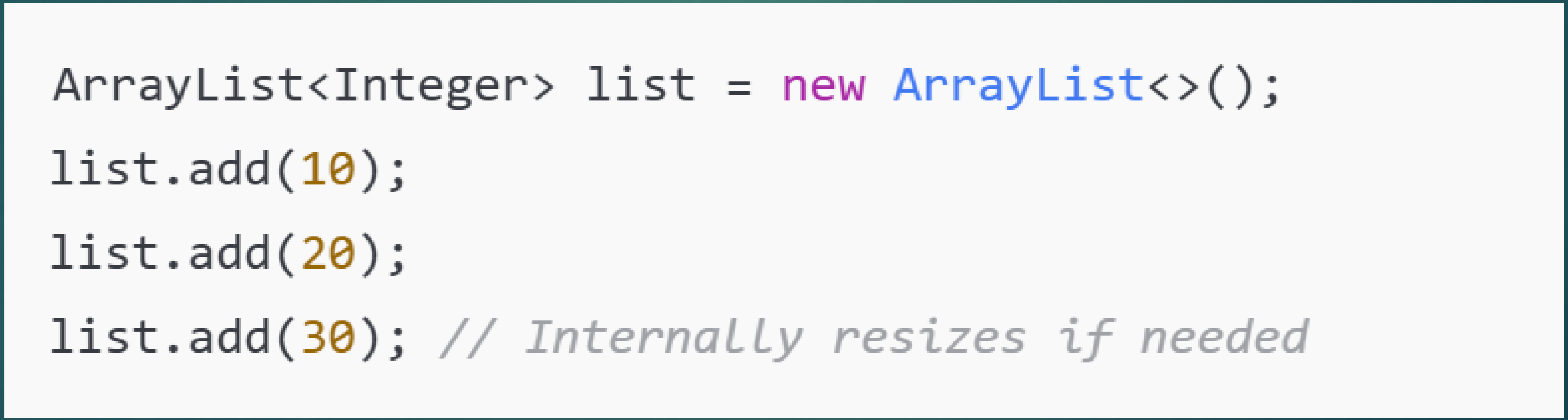
- It provides dynamic-sized arrays in Java
- No need to mention the size when creating ArrayList.



```java
import java.util.ArrayList;

class Main {
    public static void main (String[] args) {

        // Creating an ArrayList
        ArrayList<Integer> a = new ArrayList<Integer>();

        // Adding Element in ArrayList
        a.add(1);
        a.add(2);
        a.add(3);

        // Printing ArrayList
        System.out.println(a);

    }
}
```

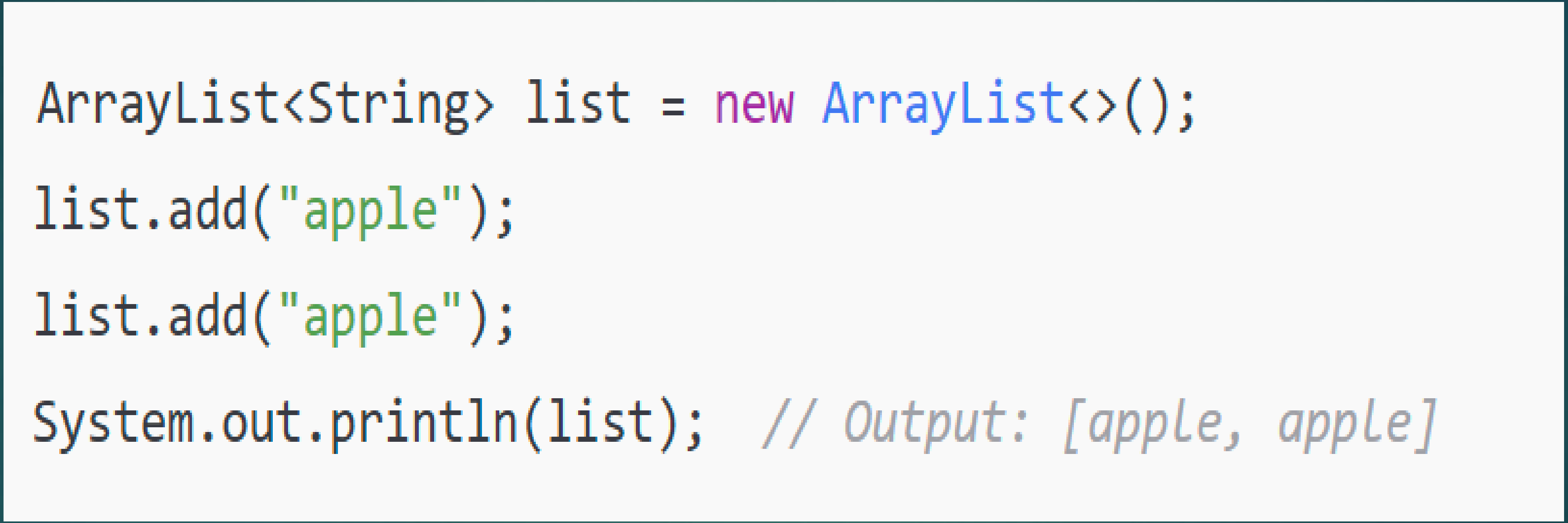
ArrayList in Java

- Java ArrayList allows us to randomly access the list.
- ArrayList is a Java class implemented using the List interface.
- **Key Points of ArrayList :**
  - ArrayList is Underlined Data Structure Resizable Array or Growable Array.



```java
ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30); // Internally resizes if needed
```

ArrayList Duplicates Are Allowed.

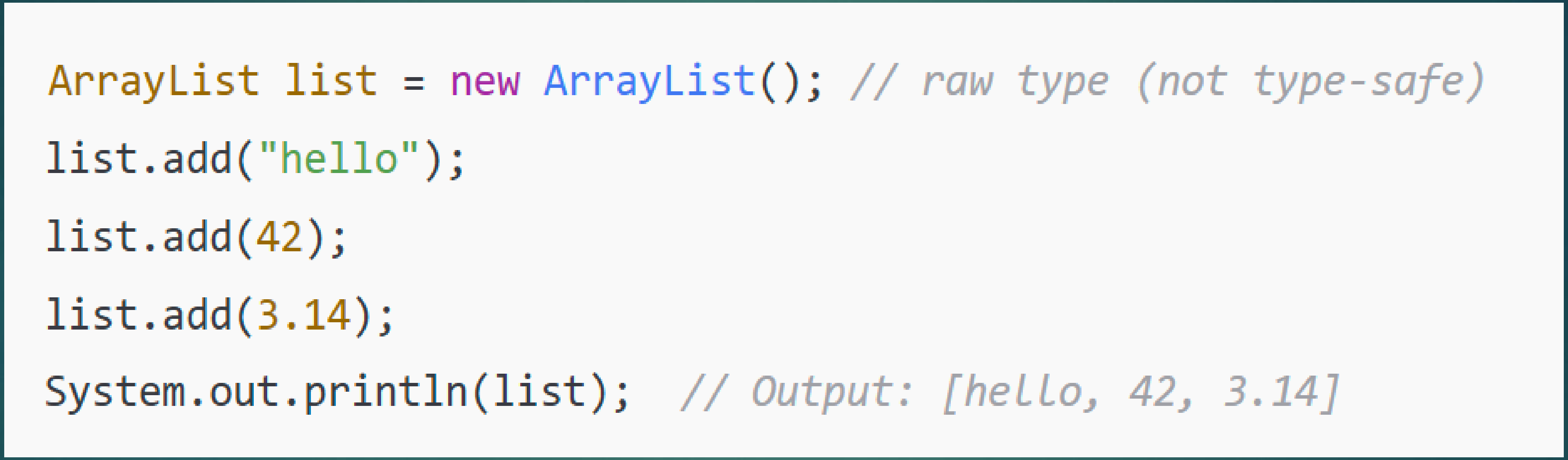


```
ArrayList<String> list = new ArrayList<>();
list.add("apple");
list.add("apple");
System.out.println(list);  // Output: [apple, apple]
```

- Insertion Order is Preserved.

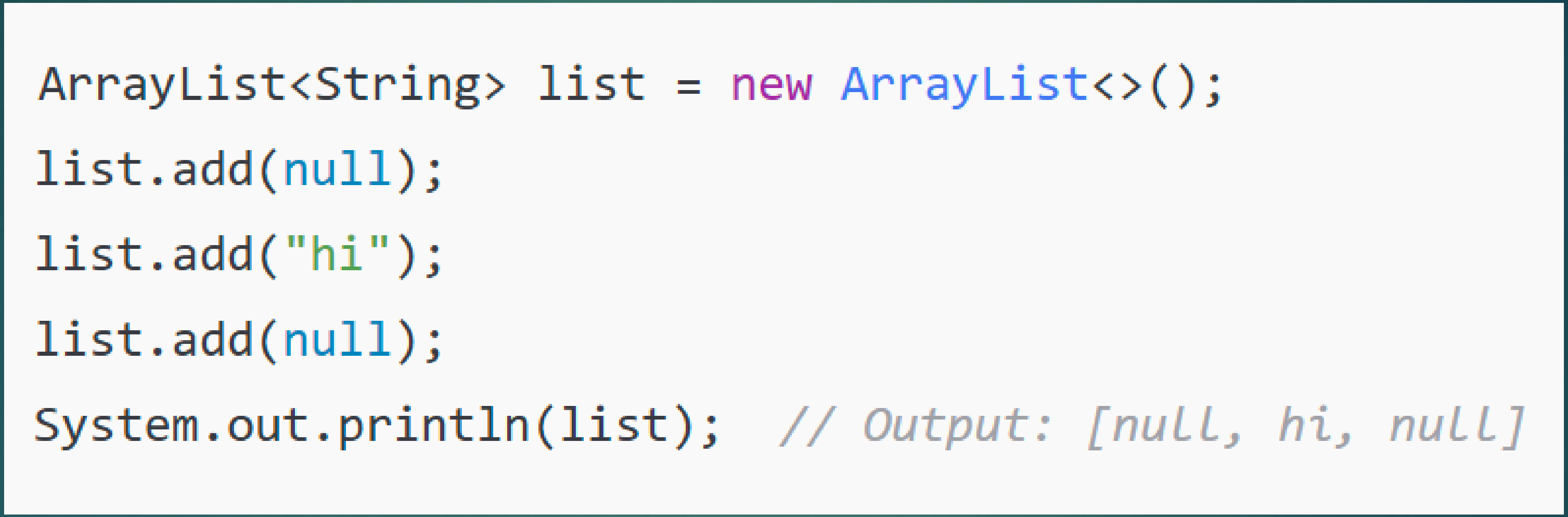
```java
ArrayList<Integer> list = new ArrayList<>();
list.add(5);
list.add(10);
list.add(15);
System.out.println(list);  // Output: [5, 10, 15]
```

❑ Heterogeneous Objects Are Allowed



```
ArrayList list = new ArrayList(); // raw type (not type-safe)
list.add("hello");
list.add(42);
list.add(3.14);
System.out.println(list);  // Output: [hello, 42, 3.14]
```

❑ Null insertion is possible



```java
ArrayList<String> list = new ArrayList<>();
list.add(null);
list.add("hi");
list.add(null);
System.out.println(list);  // Output: [null, hi, null]
```

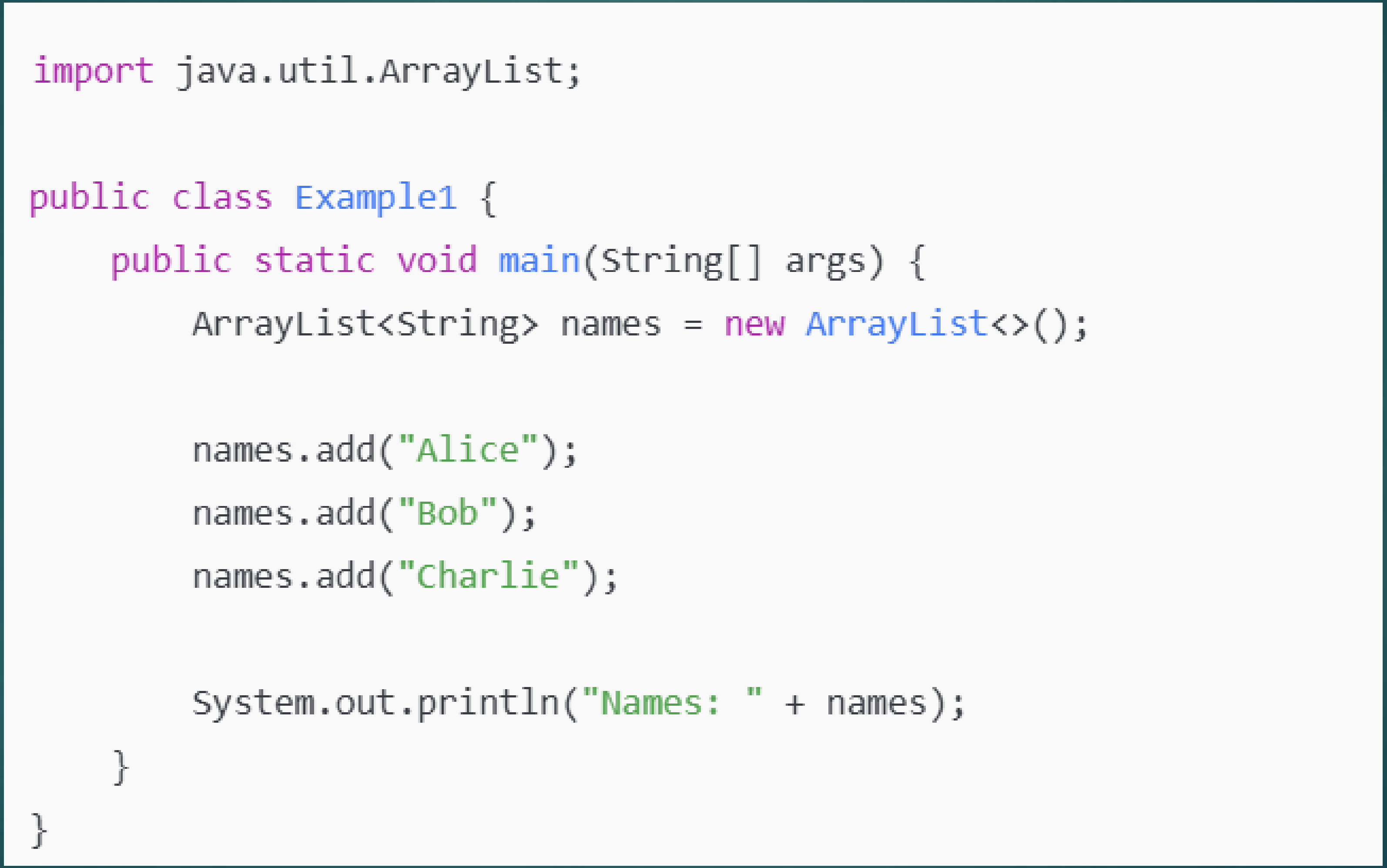
Advantages of Java ArrayList

- **Dynamic size:** ArrayList can dynamically grow and shrink in size, making it easy to add or remove elements as needed.
- **Easy to use**: ArrayList is simple to use, making it a popular choice for many Java developers.
- **Fast access**: ArrayList provides fast access to elements, as it is implemented as an array under the hood.
- **Ordered collection**: ArrayList preserves the order of elements, allowing you to access elements in the order they were added.
- **Supports null values**: ArrayList can store null values, making it useful in cases where the absence of a value needs to be represented.

Disadvantages of Java ArrayList

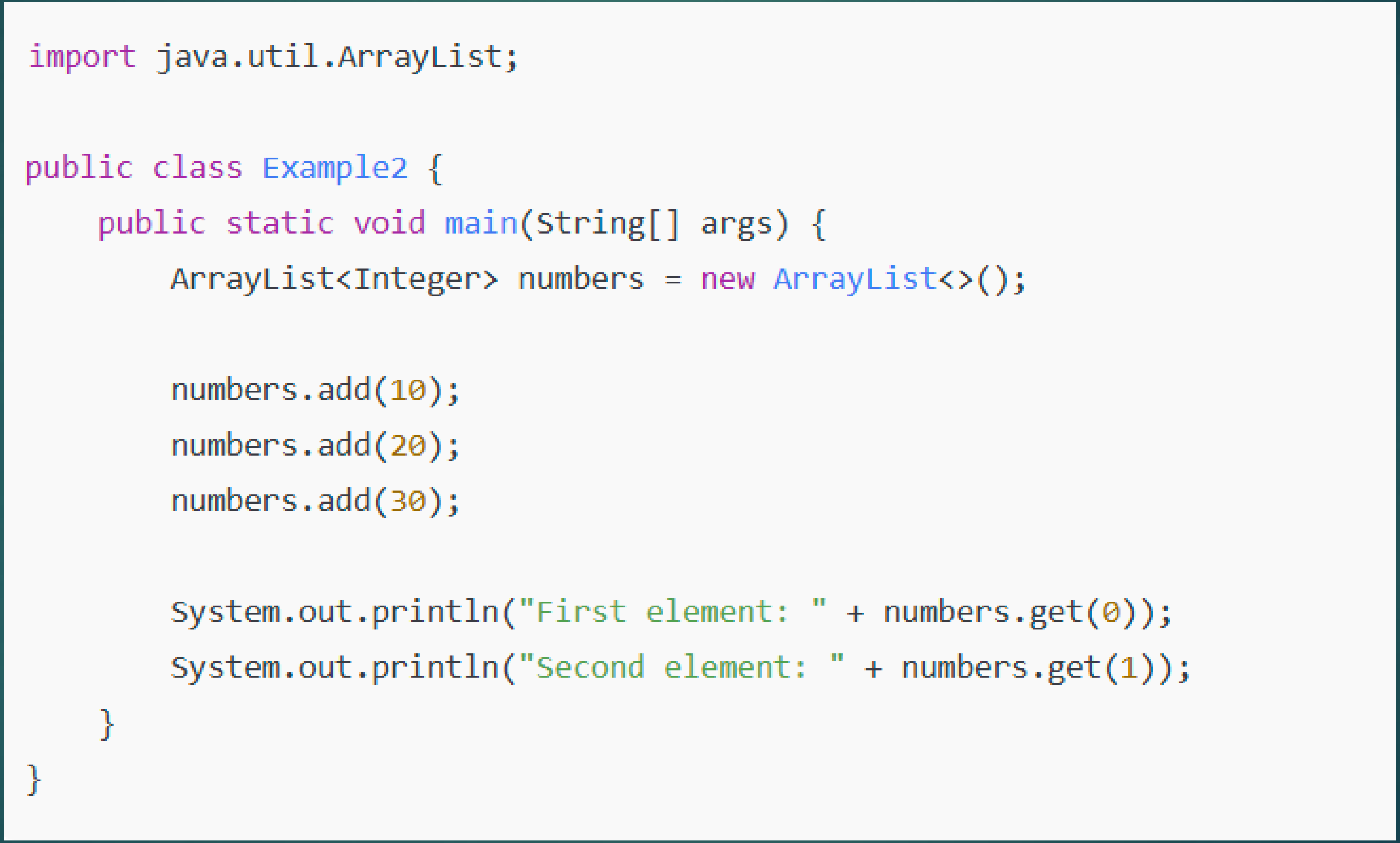
- **Slower than arrays**: ArrayList is slower than arrays for certain operations, such as inserting elements in the middle of the list.
- **Increased memory usage**: ArrayList requires more memory than arrays, as it needs to maintain its dynamic size and handle resizing.
- **Not thread-safe:** ArrayList is not thread-safe, meaning that multiple threads may access and modify the list concurrently, leading to potential race conditions and data corruption.
- **Performance degradation**: ArrayList's performance may degrade as the number of elements in the list increases, especially for operations such as searching for elements or inserting elements in the middle of the list.

1. Add and Display Elements using ArrayList



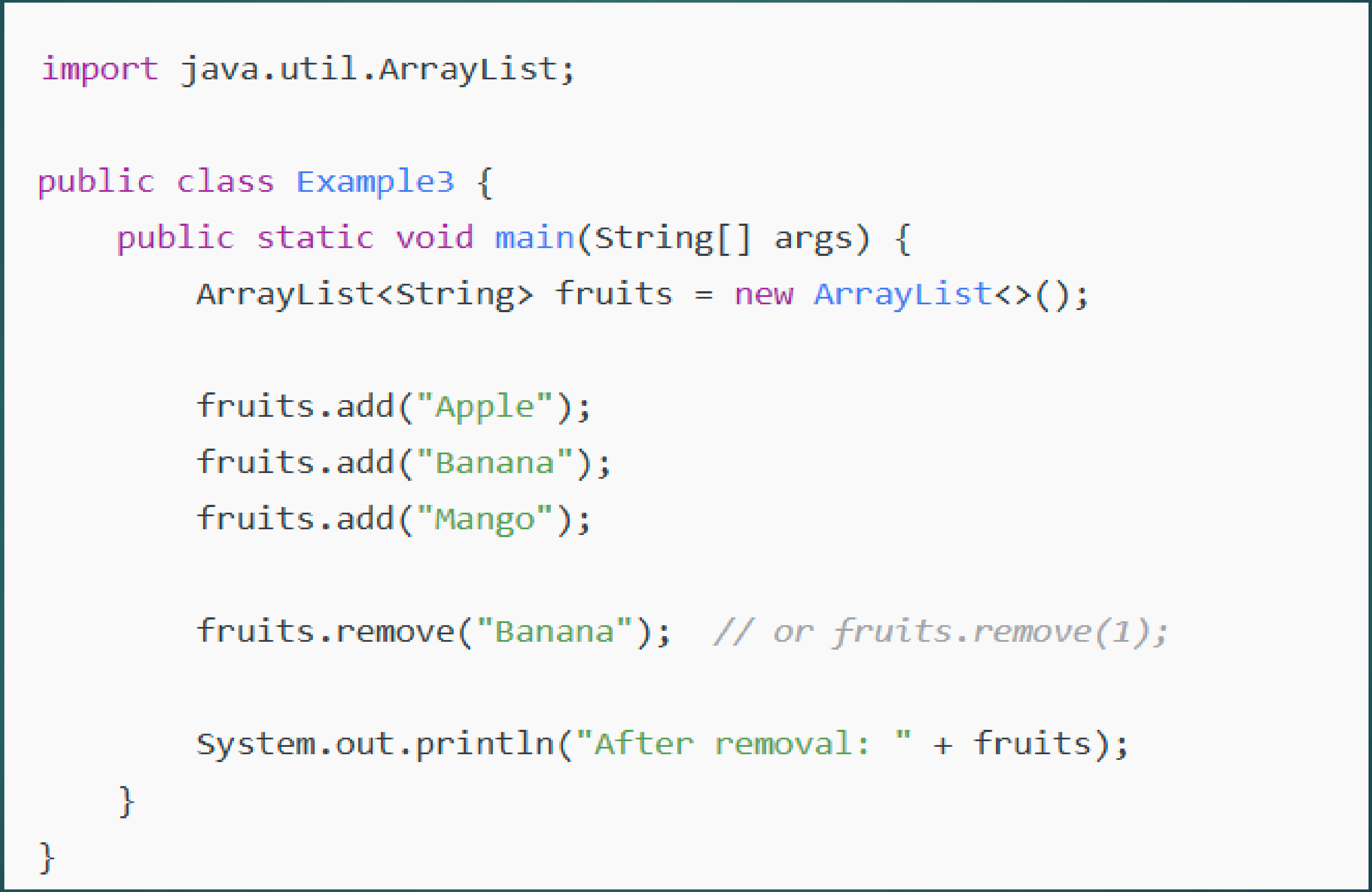
```java
import java.util.ArrayList;

public class Example1 {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();

        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println("Names: " + names);
    }
}
```

2. Access Elements by index



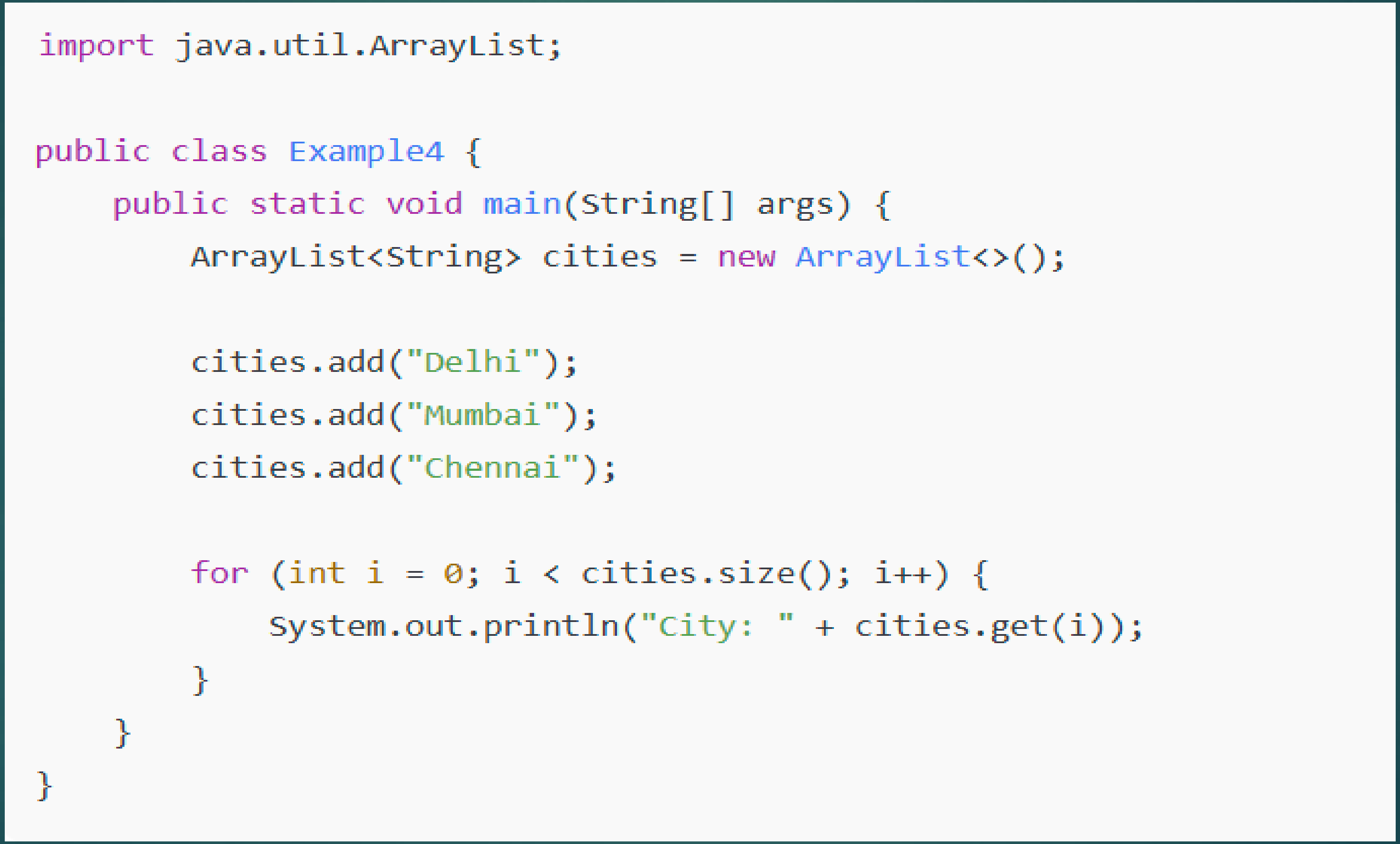
```java
import java.util.ArrayList;

public class Example2 {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();

        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        System.out.println("First element: " + numbers.get(0));
        System.out.println("Second element: " + numbers.get(1));
    }
}
```

3. Remove Elements



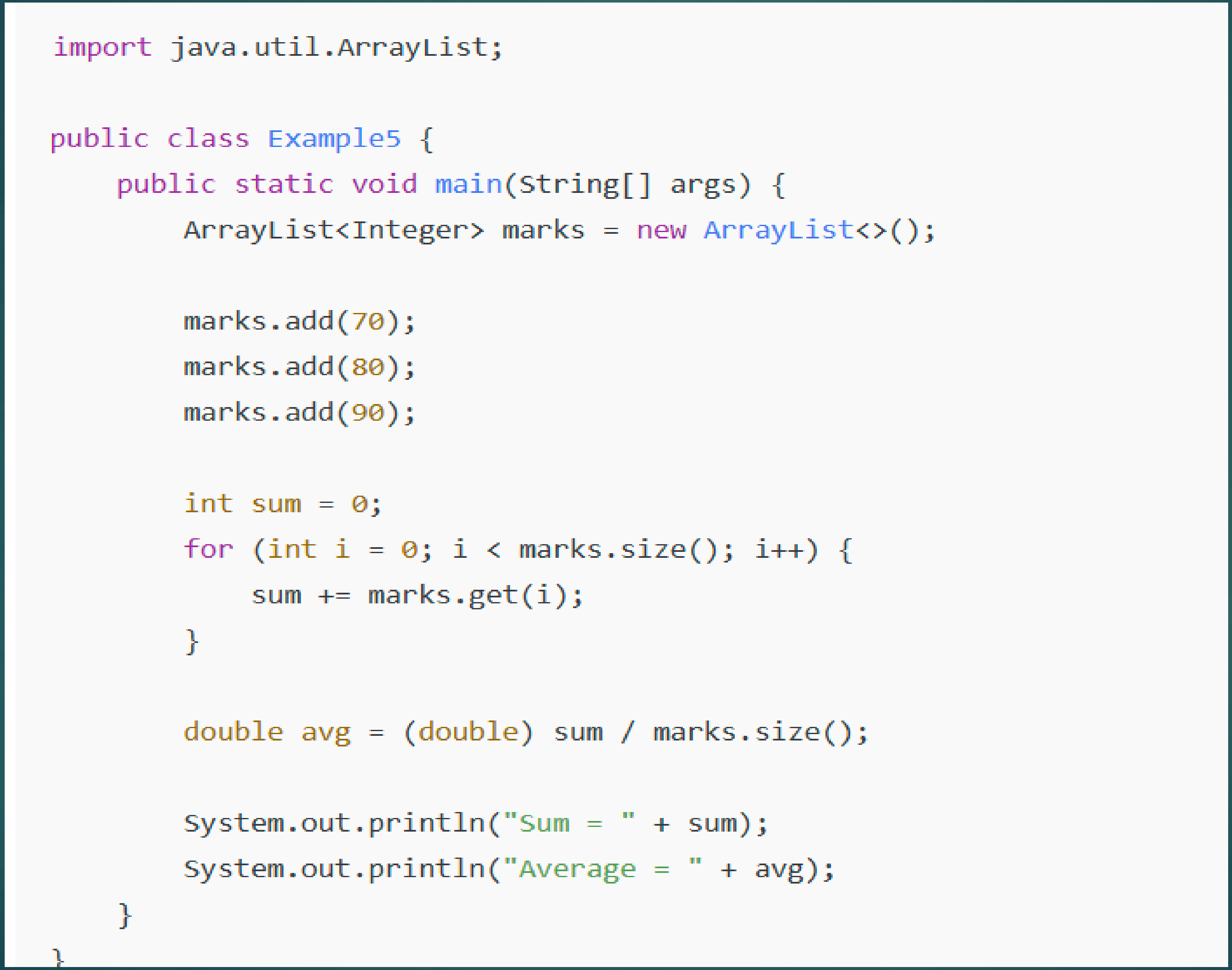
```java
import java.util.ArrayList;

public class Example3 {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();

        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        fruits.remove("Banana");  // or fruits.remove(1);

        System.out.println("After removal: " + fruits);
    }
}
```

4. Loop Through ArrayList



```java
import java.util.ArrayList;

public class Example4 {
    public static void main(String[] args) {
        ArrayList<String> cities = new ArrayList<>();

        cities.add("Delhi");
        cities.add("Mumbai");
        cities.add("Chennai");

        for (int i = 0; i < cities.size(); i++) {
            System.out.println("City: " + cities.get(i));
        }
    }
}
```

5. Find Sum and Average using ArrayList

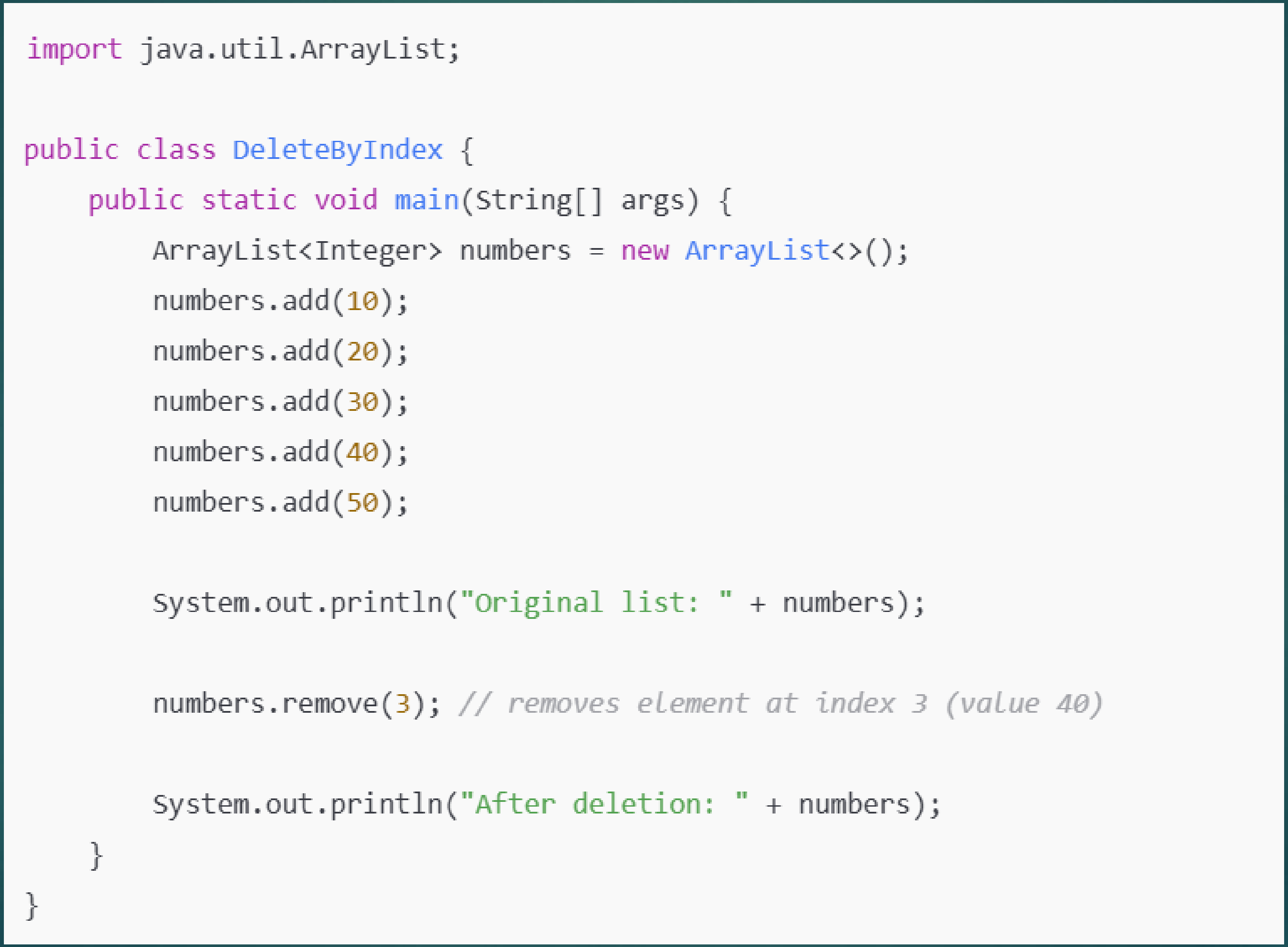


```java
import java.util.ArrayList;

public class Example5 {
    public static void main(String[] args) {
        ArrayList<Integer> marks = new ArrayList<>();

        marks.add(70);
        marks.add(80);
        marks.add(90);

        int sum = 0;
        for (int i = 0; i < marks.size(); i++) {
            sum += marks.get(i);
        }

        double avg = (double) sum / marks.size();

        System.out.println("Sum = " + sum);
        System.out.println("Average = " + avg);
    }
}
```

6. Check is elements exists

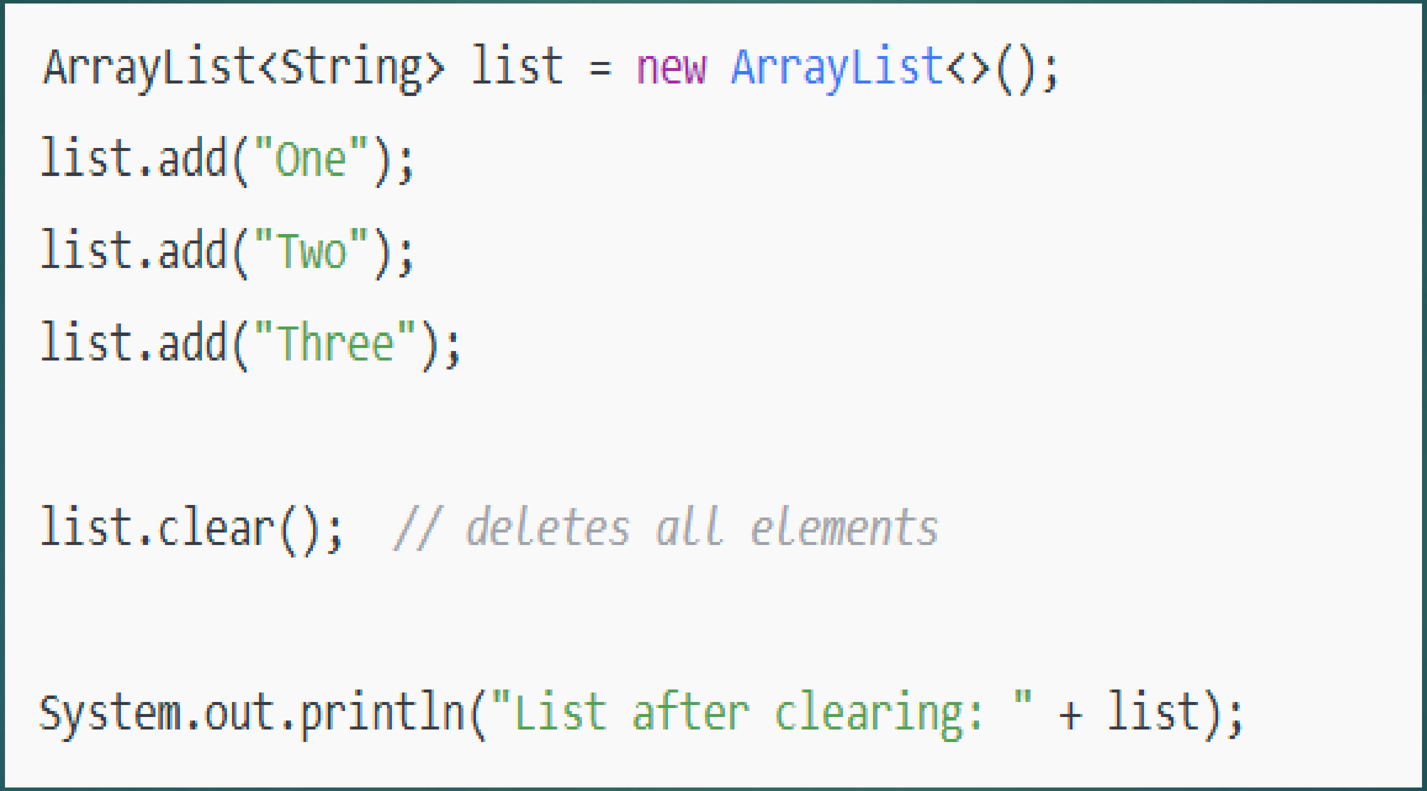
```java
import java.util.ArrayList;

public class Example6 {
    public static void main(String[] args) {
        ArrayList<String> colors = new ArrayList<>();

        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        if (colors.contains("Green")) {
            System.out.println("Green is in the list.");
        } else {
            System.out.println("Green is not in the list.");
        }
    }
}
```

Check is elements exists

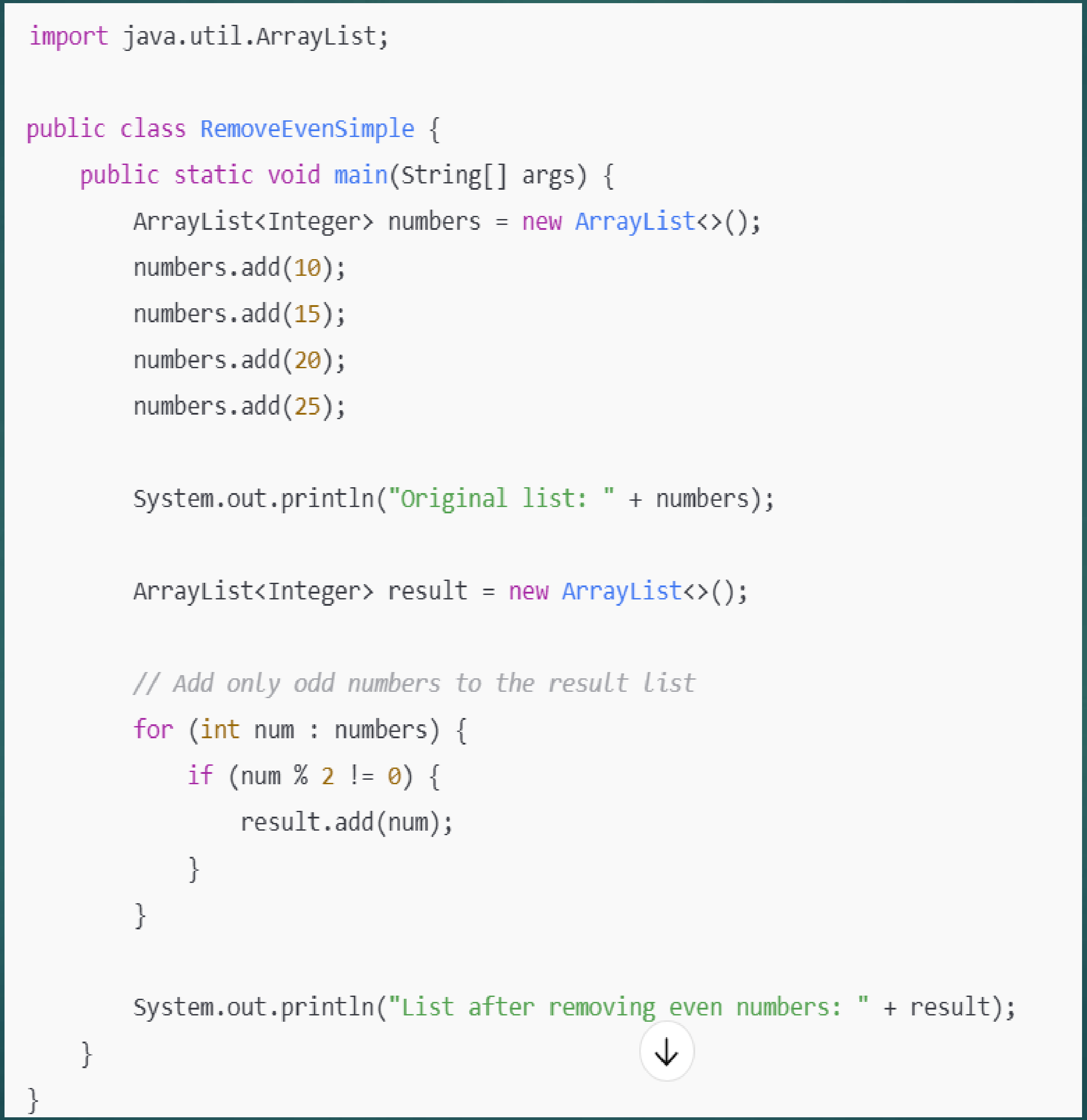


```java
import java.util.ArrayList;

public class DeleteByIndex {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        numbers.add(40);
        numbers.add(50);

        System.out.println("Original list: " + numbers);

        numbers.remove(3); // removes element at index 3 (value 40)

        System.out.println("After deletion: " + numbers);
    }
}
```

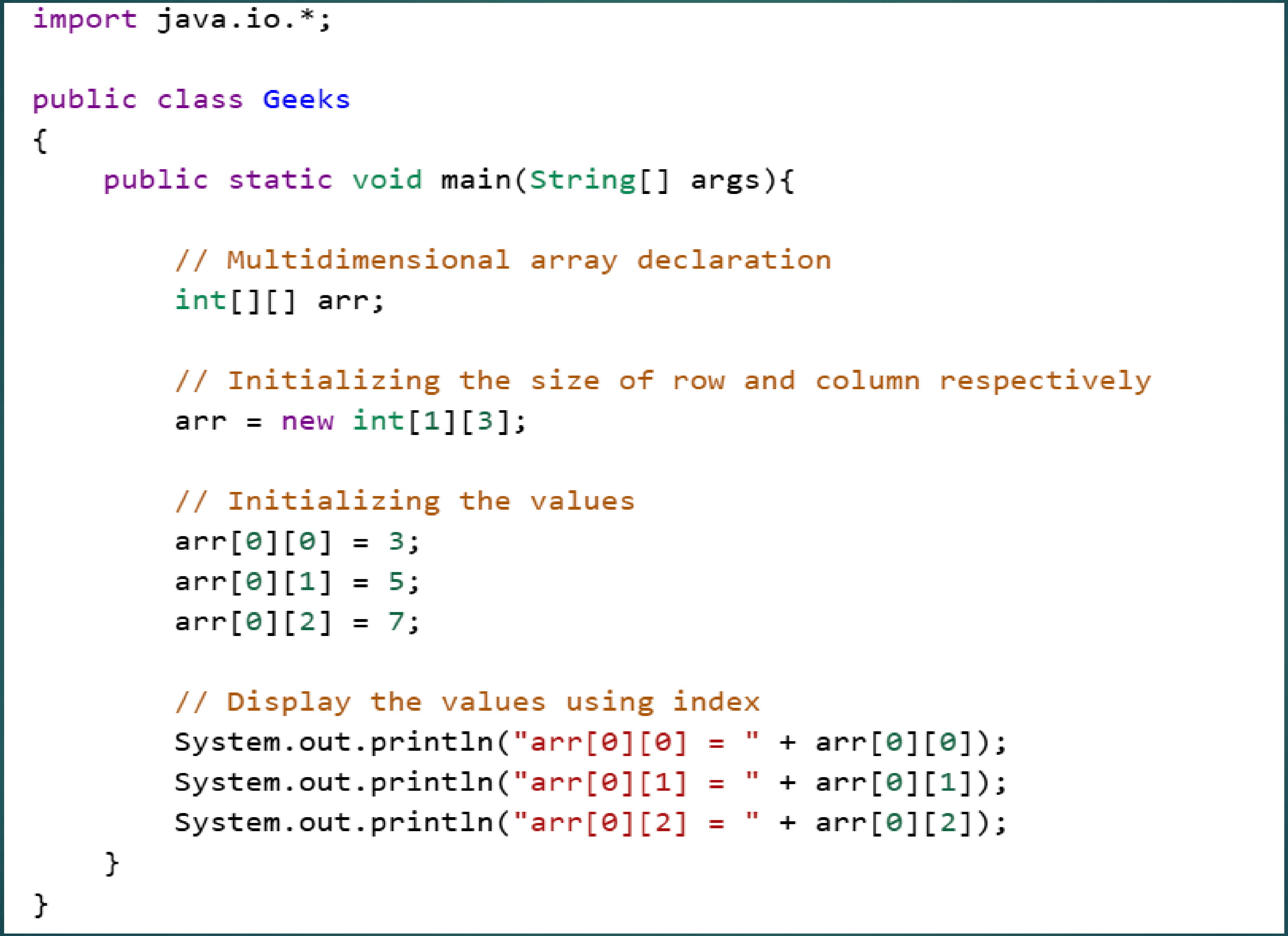
```java
import java.util.ArrayList;

public class DeleteByValue {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Grapes");

        System.out.println("Original list: " + fruits);

        fruits.remove("Mango");

        System.out.println("After deleting 'Mango': " + fruits);
    }
}
```



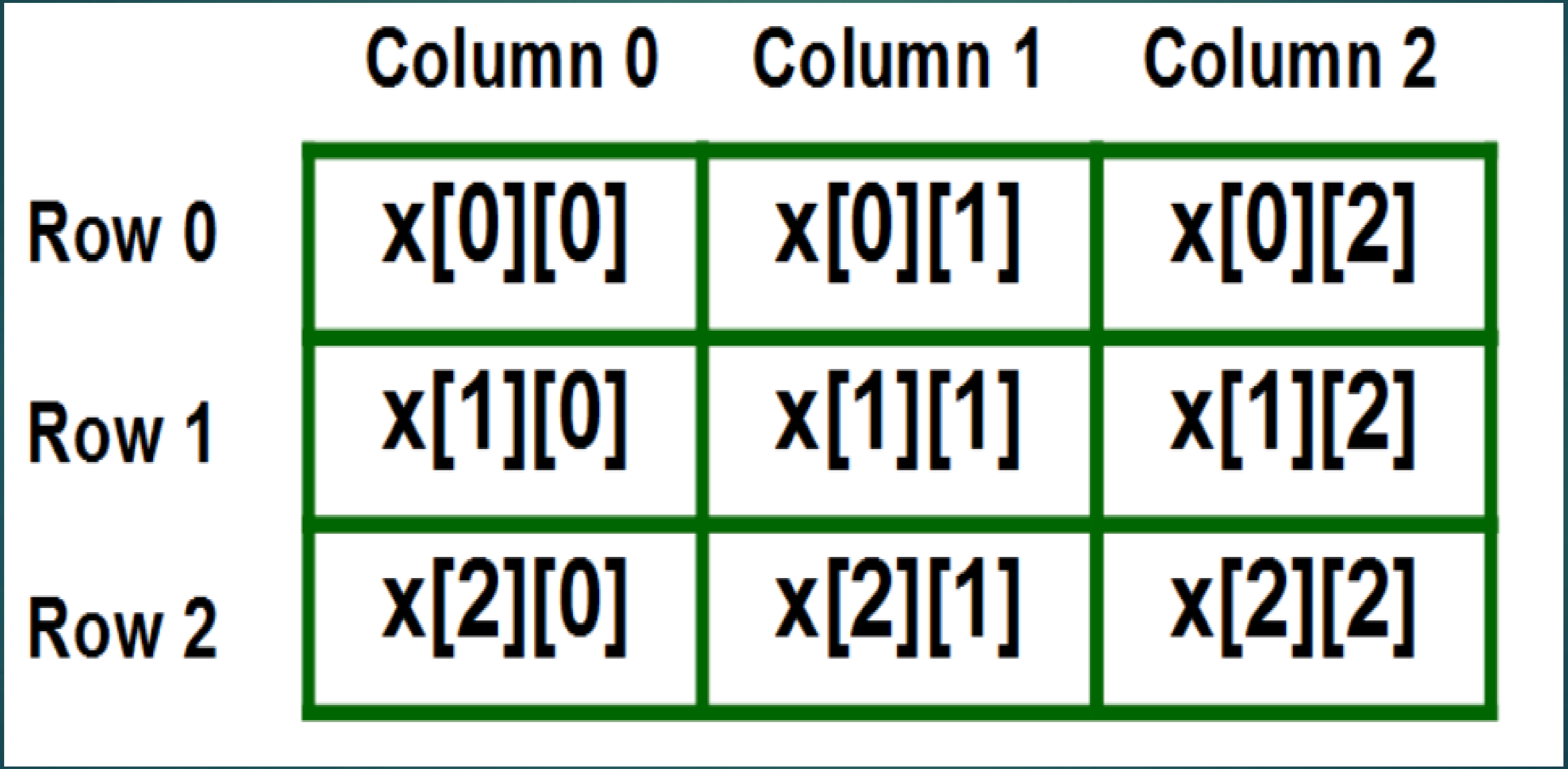
```java
ArrayList<String> list = new ArrayList<>();
list.add("One");
list.add("Two");
list.add("Three");

list.clear();  // deletes all elements

System.out.println("List after clearing: " + list);
```



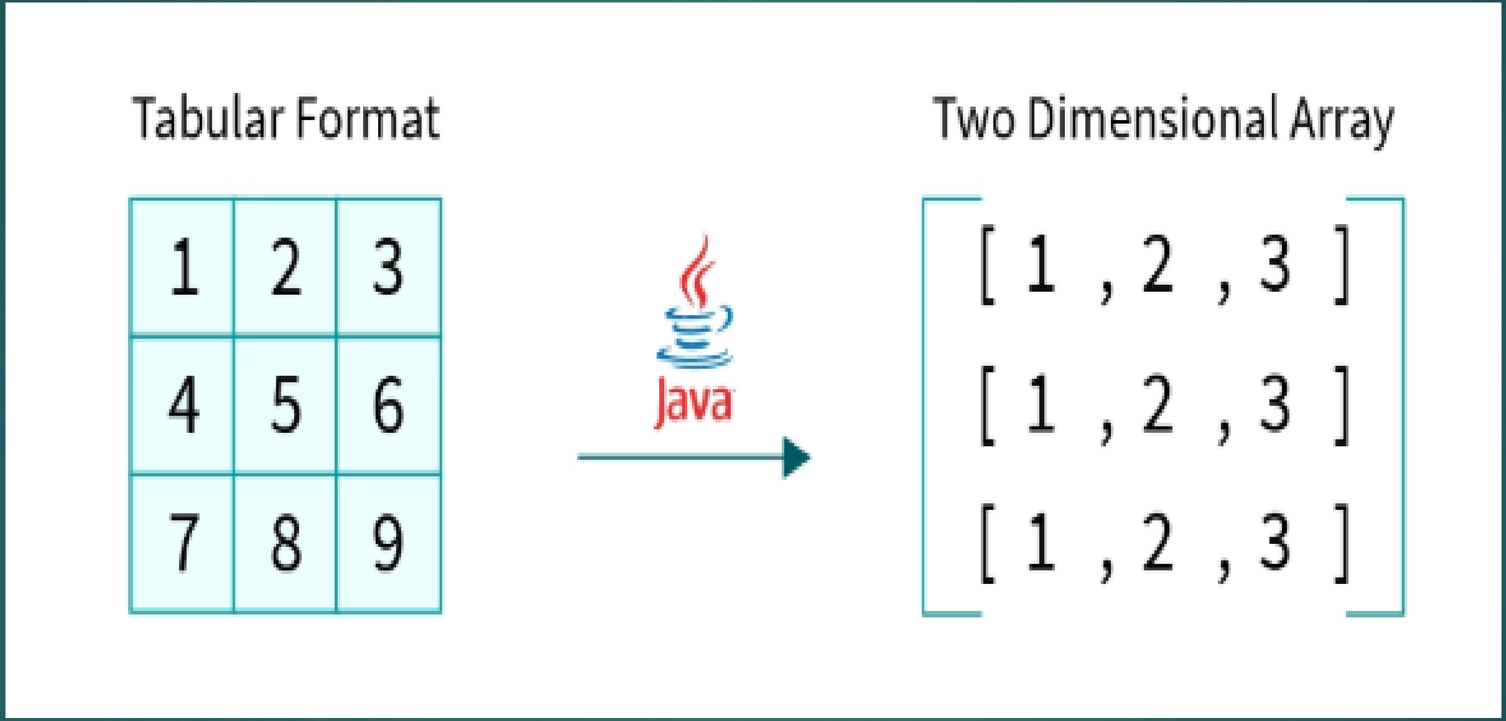
```java
import java.util.ArrayList;

public class RemoveEvenSimple {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(15);
        numbers.add(20);
        numbers.add(25);

        System.out.println("Original list: " + numbers);

        ArrayList<Integer> result = new ArrayList<>();

        // Add only odd numbers to the result list
        for (int num : numbers) {
            if (num % 2 != 0) {
                result.add(num);
            }
        }

        System.out.println("List after removing even numbers: " + result);
    }
}
```



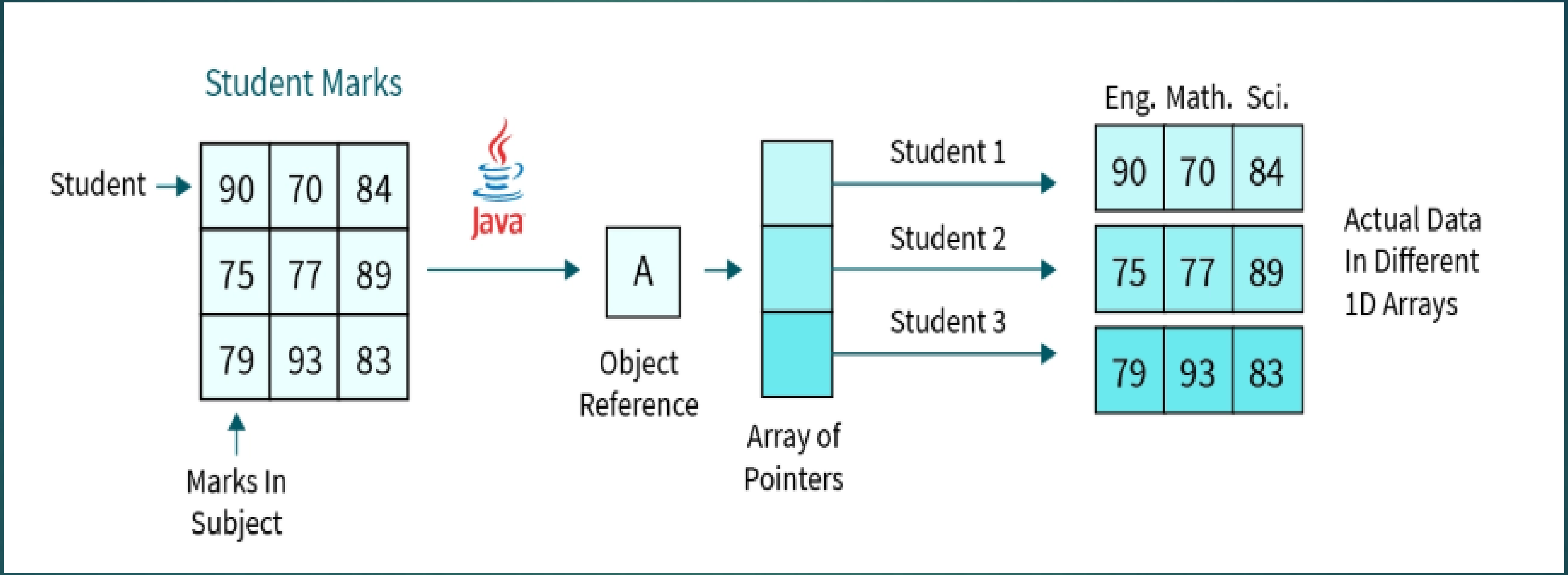
```java
import java.io.*;

public class Geeks
{
    public static void main(String[] args){

        // Multidimensional array declaration
        int[][] arr;

        // Initializing the size of row and column respectively
        arr = new int[1][3];

        // Initializing the values
        arr[0][0] = 3;
        arr[0][1] = 5;
        arr[0][2] = 7;

        // Display the values using index
        System.out.println("arr[0][0] = " + arr[0][0]);
        System.out.println("arr[0][1] = " + arr[0][1]);
        System.out.println("arr[0][2] = " + arr[0][2]);
    }
}
```

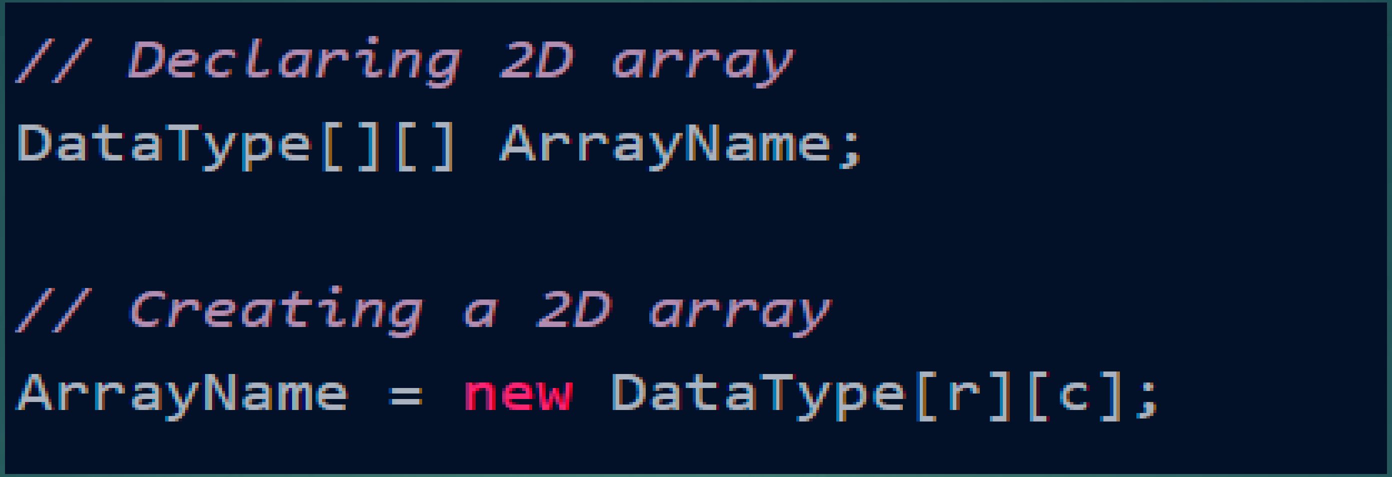


	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

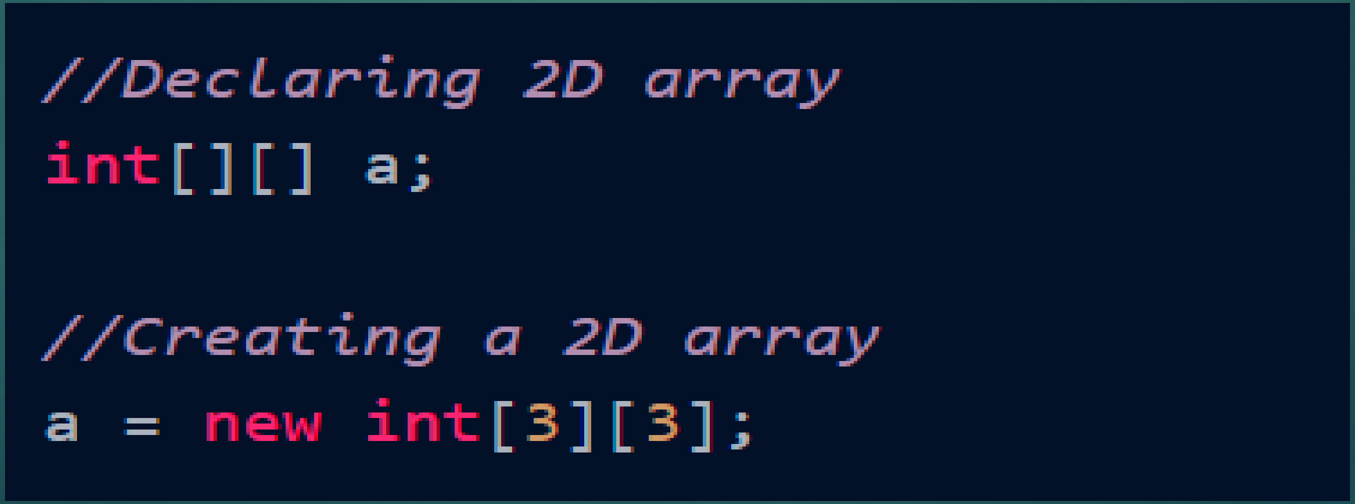


Tabular Format
1 2 3
4 5 6
7 8 9
Java
Two Dimensional Array
[1 , 2 , 3]
[1 , 2 , 3]
[1 , 2 , 3]

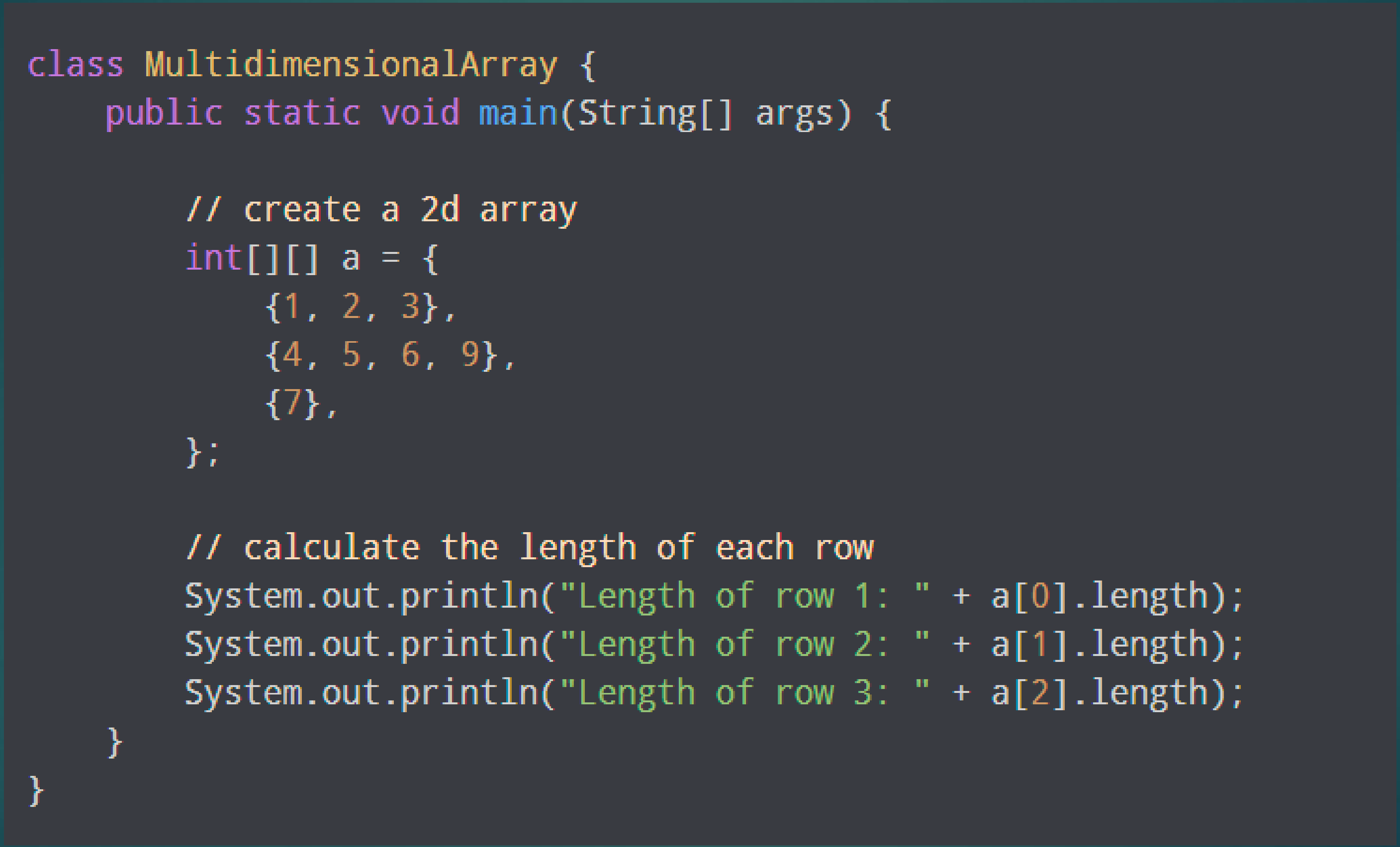

Student Marks
Student
90 70 84
75 77 89
79 93 83
Marks In Subject
Java
A
Object Reference
Array of Pointers
Student 1
Student 2
Student 3
Eng. Math. Sci.
90 70 84
75 77 89
79 93 83
Actual Data In Different 1D Arrays



```
// Declaring 2D array
DataType[][] ArrayName;

// Creating a 2D array
ArrayName = new DataType[r][c];
```

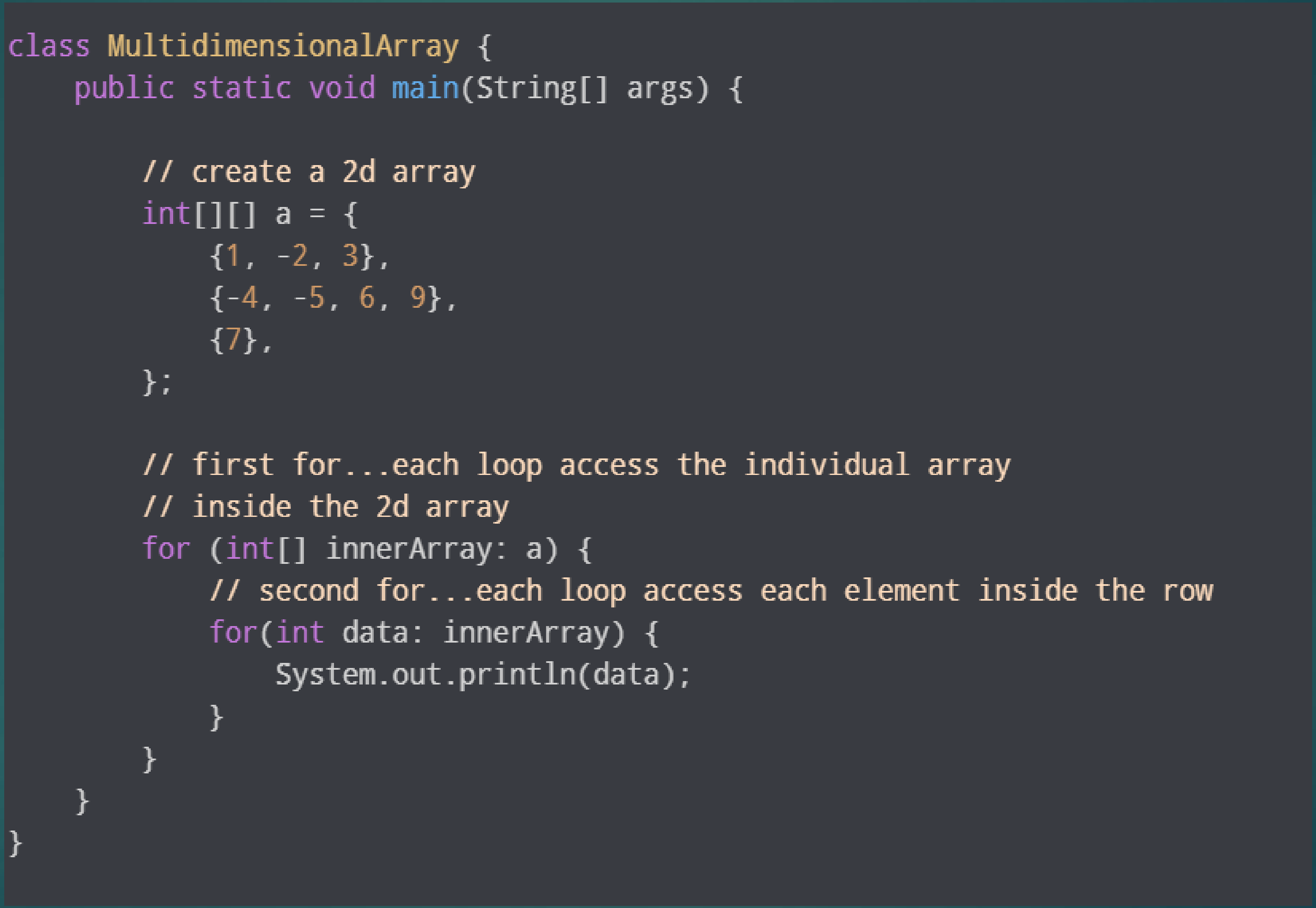


```
//Declaring 2D array
int[][] a;

//Creating a 2D array
a = new int[3][3];
```



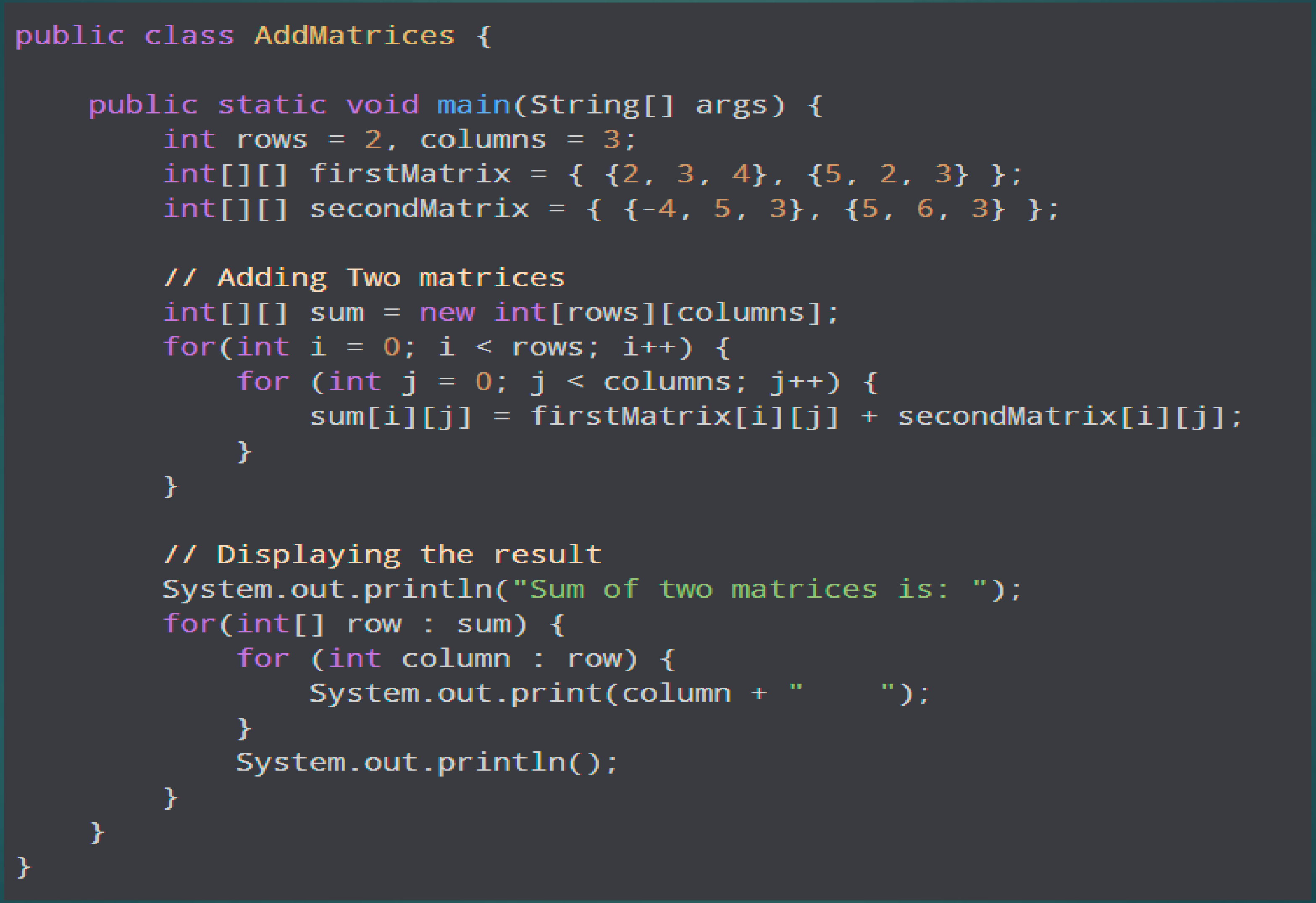
```java
class MultidimensionalArray {
    public static void main(String[] args) {

        // create a 2d array
        int[][] a = {
            {1, 2, 3},
            {4, 5, 6, 9},
            {7},
        };

        // calculate the length of each row
        System.out.println("Length of row 1: " + a[0].length);
        System.out.println("Length of row 2: " + a[1].length);
        System.out.println("Length of row 3: " + a[2].length);
    }
}
```

```java
class MultidimensionalArray {
    public static void main(String[] args) {

        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        for (int i = 0; i < a.length; ++i) {
            for(int j = 0; j < a[i].length; ++j) {
                System.out.println(a[i][j]);
            }
        }
    }
}
```



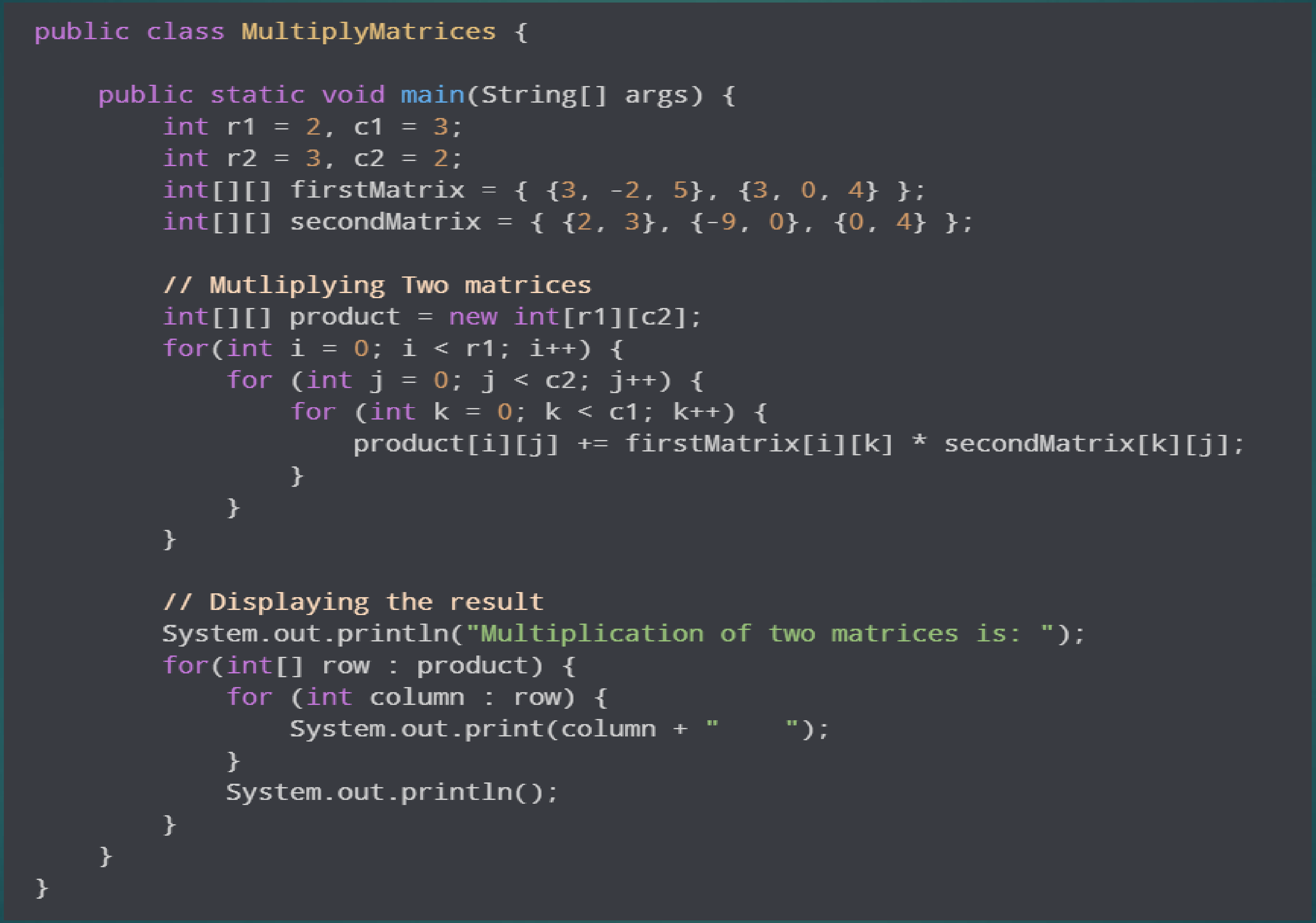
```java
class MultidimensionalArray {
    public static void main(String[] args) {

        // create a 2d array
        int[][] a = {
            {1, -2, 3},
            {-4, -5, 6, 9},
            {7},
        };

        // first for...each loop access the individual array
        // inside the 2d array
        for (int[] innerArray: a) {
            // second for...each loop access each element inside the row
            for(int data: innerArray) {
                System.out.println(data);
            }
        }
    }
}
```

Program to add two matrices



```java
public class AddMatrices {

    public static void main(String[] args) {
        int rows = 2, columns = 3;
        int[][] firstMatrix = { {2, 3, 4}, {5, 2, 3} };
        int[][] secondMatrix = { {-4, 5, 3}, {5, 6, 3} };

        // Adding Two matrices
        int[][] sum = new int[rows][columns];
        for(int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                sum[i][j] = firstMatrix[i][j] + secondMatrix[i][j];
            }
        }

        // Displaying the result
        System.out.println("Sum of two matrices is: ");
        for(int[] row : sum) {
            for (int column : row) {
                System.out.print(column + "    ");
            }
            System.out.println();
        }
    }
}
```

Program to add two matrices



```java
public class MultiplyMatrices {

    public static void main(String[] args) {
        int r1 = 2, c1 = 3;
        int r2 = 3, c2 = 2;
        int[][] firstMatrix = { {3, -2, 5}, {3, 0, 4} };
        int[][] secondMatrix = { {2, 3}, {-9, 0}, {0, 4} };

        // Mutliplying Two matrices
        int[][] product = new int[r1][c2];
        for(int i = 0; i < r1; i++) {
            for (int j = 0; j < c2; j++) {
                for (int k = 0; k < c1; k++) {
                    product[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
                }
            }
        }

        // Displaying the result
        System.out.println("Multiplication of two matrices is: ");
        for(int[] row : product) {
            for (int column : row) {
                System.out.print(column + "    ");
            }
            System.out.println();
        }
    }
}
```