Arrays

eazy bytes

As of now, we learned about primitive data types like int, float, double, char, String. But using primitive data types, we can only store a single value. But what if we want to store multiple values ?

An array is a fixed-length data structure in Java that allows you to store multiple values of the same data type under a single variable name. The elements can be of any data type including primitive types (such as int, float, double, boolean, char) or objects.

Arrays

eazy bytes

Let's say you need to keep track of mobile numbers of your friends. If you only have 3 friends, you could create variables like below to store their mobile numbers. But what if you have more friends, say fifty or even a thousand? It would be impractical and messy to create a separate variable for each friend mobile number.

```
int mobileNumber1, mobileNumber2, mobileNumber3;
```

Here's where arrays come in handy. Instead of declaring individual variables, you can use a single array to store multiple values of the same type (like integers for mobile numbers). Arrays help you manage and organize data more efficiently. In Java, there's a limit to the number of values an array can hold, but it's a very large number (2,147,483,647) which is the maximum value of the int data type. This makes arrays a practical solution for handling large amounts of data.

What turns a regular variable into an array is the addition of square brackets [] after the data type or after the variable name in the declaration.

```
int mobileNumber; // Normal variable declaration of type int
```

```
int[] mobileNumbers; // By adding [] after int, it transforms mobileNumbers into an array variable. This declaration is read as " mobileNumbers is an array of int."
```

```
int mobileNumbers[]; // Alternatively, you can place the [] after the variable name. This syntax is less common but still valid. It also makes mobileNumbers an array variable.
```

Arrays

eazy bytes

We have the flexibility to declare arrays for both primitive and reference types. Here are additional examples of array declarations:

```
double[] prices; // An array holding double values
char[] grades; // An array for characters
String[] names; // An array of String objects
Person[] persons; // An array of objects belonging to a custom class (reference type)
```

Up to this point, we've set the stage for storing multiple values in a single variable using arrays. However, at the initial declaration of an array, you don't specify the exact number of values it can hold. The number of values an array can accommodate is not determined until later. The following sections will elaborate on how to specify and define the size or number of values an array can hold.

Below is the general syntax for array declaration along with the size,

```
dataType[] arrayName = new dataType[size];  // Declaration with size;
```

Below is the general syntax for array declaration and initialization with values,

```
dataType[] arrayName = {value1, value2, ..., valueN}; // Declaration and initialization with values
```

Arrays

eazy bytes

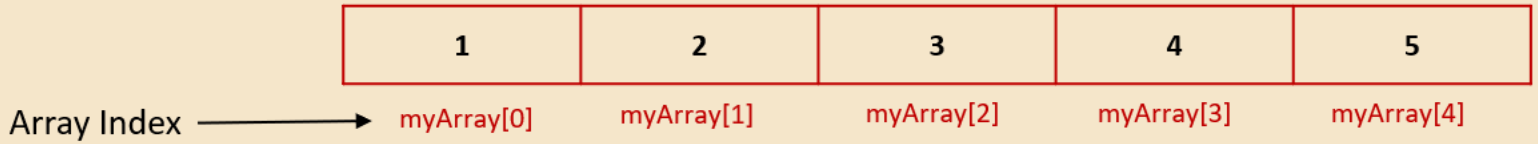
Here's an example of how to declare an array of integers in Java. In this example, we are declaring an array named myArray that can hold 5 integers. The new keyword is used to create the array and the number in the brackets indicates the size of the array.

```
int[] myArray = new int[5];
```

You can also declare an array and initialize it with values at the same time, like below. In this example, we are declaring an array named myArray and initializing it with the values 1, 2, 3, 4, and 5.

```
int[] myArray = { 1, 2, 3, 4, 5 };
```

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



Each item in an array is called an element, and each element is accessed by its numerical index. In Java, array indexes start at 0. If the size of an array is n, then the last element of the array will be at index n-1. So to access the first element of the myArray array, we would use the following code. This code would assign the value of 1 to the firstElement variable.

```
int firstElement = myArray[0];
```

You can also add/update value of an element in an array by its index. For example, to assign the value of the first element in myArray to 10, you would use the following code:

```
myArray[0] = 10;
```

ArrayIndexOutOfBoundsException will be thrown when the code tries to access the element by the index that is equal to, or bigger than, the array length.

Arrays

eazy bytes

How are you going to create an array to store 116 mobile numbers? You can do this as follows:

```
int[] mobileNumbers = new int[116];
```

You can also use an expression to specify the length or size of an array while creating the array. Below is the example,

```
int length = 116;
int[] mobileNumbers1 = new int[length];     // mobileNumbers1 can store 116 elements
int[] mobileNumbers2 = new int[length * 2]; // mobileNumbers2 can store 232 elements
```

Arrays

eazy bytes

An array object has a public final instance variable named length which returns the number of elements in the array or the length of the array,

```
int[] mobileNumbers = new int[116]; // Creates an array of length 116
int length = mobileNumbers.length;  // 116 will be assigned to length variable
```

You can use the length property to loop through an array and access all of its elements. Here's an example. This code would loop through the mobileNumbers array and print all the mobile numbers stored inside the array,

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

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

}
```

It's crucial to emphasize that when executing a loop for array processing, the loop condition should verify that the array index is less than the array's length, as in "i < mobileNumbers.length" This is because array indices begin at 0, not 1. Another common error is starting the loop counter at 1 instead of 0.

If you were to change the initialization part of the for loop in the previous code from "int i = 0" to "int i = 1," it wouldn't result in any errors. However, the consequence would be that the first element, wouldn't be processed. The loop would begin with the second element, potentially leading to unintended behavior in your array processing.

Arrays

eazy bytes

We can create an empty array with the length as zero like shown below,

```
int[] emptyArray = new int[0]; or int[] emptyNumList = { }; // An empty array will be created
```

Once an array is created, its length cannot be altered. If there's a need to change the length, the solution involves creating a new array and transferring the elements from the old array to the new one. It's worth noting that an array is allowed to have a length of zero.

```
int[] mobileNumbers = new int[116];  // Creates an array of length 116
mobileNumbers.length = 100; // Compilation fails
```

Elements in an array of primitive data types are set to their respective default values upon initialization. Numeric array elements are initialized to zero, boolean elements to false, and char elements to '\u0000'. Meanwhile, elements in an array of reference types are initialized to null.
Below are the few examples,

```
int[] mobileNumbers = new int[5];  // All the int type elements in array are initialized with a default value 0
String[] names = new String[10]; // All the String type elements in array are initialized with a default value null
Person[] persons = new Person[9]; // All the Person type elements in array are initialized with a default value null
```

for-each loop

eazy bytes

A for-each loop, also known as an enhanced for loop, is a loop in Java that is designed to iterate over elements of an array or a collection. It was introduced in Java 5 and provides a simplified way to iterate through arrays or collections without having to explicitly use an index.

The syntax of a for-each loop in Java is as follows:

```
for (datatype variable : array/collection) {
    // code to be executed for each element
}
```

Here, datatype specifies the type of elements in the array or collection, variable is a temporary variable that is used to refer to the current element in each iteration, and array/collection is the array or collection that is being iterated over. For example, consider the following array of integers:

```
int[] numbers = {1, 2, 3, 4, 5};
```

We can use a for-each loop to iterate through the elements of this array as follows:

```
for (int num : numbers) {
    System.out.println(num);
}
```

eazy bytes

The main difference between a for loop and a for-each loop is the way they iterate through the elements of an array. A for loop uses an index variable to access each element of the array one by one, whereas a for-each loop directly accesses each element of the array without using an index variable.

for loop

```
int[] numbers = {1, 2, 3, 4, 5};

// Using for loop
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

In a for loop, we have to explicitly declare and initialize an index variable, and use it to access each element of the array using the square bracket notation. The loop condition is usually based on the length of the array, and the index variable is incremented or decremented at the end of each iteration.

for-each loop

```
int[] numbers = {1, 2, 3, 4, 5};

// Using for-each loop
for (int num : numbers) {
    System.out.println(num);
}
```

In a for-each loop, we don't have to declare or initialize an index variable, and we can directly access each element of the array using a temporary variable that is declared in the loop header. The loop condition is implicit, based on the size of the array or collection, and the loop variable is automatically assigned to each element of the array or collection in turn.

The for-each loop is often preferred over the traditional for loop when iterating through arrays or collections because it is simpler and less error-prone. It is also faster in many cases because the loop variable is not used to access the array or collection elements. However, the for-each loop is not suitable when we need to modify the array or collection elements during the iteration, as we cannot modify the loop variable directly. In such cases, we have to use the traditional for loop instead.

Arrays advantages and disadvantages

eazy bytes

Every element within an array is stored consecutively in memory. In the scenario of a reference-type array, the array elements store references to objects. These references are stored in contiguous memory, but it's important to note that the actual objects they point to on the heap might not be stored contiguously. Objects on the heap can be scattered, and their locations are typically not contiguous.

```
int[] myArray = { 1, 2, 3, 4, 5 };
```

	Index 0	Index 1	Index 2	Index 3	Index 4
	1	2	3	4	5
Some random memory location →	3415	3416	3417	3418	3419

```
Person[] persons = new Person[3];
```

	Index 0	Index 1	Index 2
	person1	person2	person3
Some random memory location →	4016	4017	4018

Heap memory

Arrays advantages and disadvantages

eazy bytes

Arrays in Java have a fixed size, which can be advantageous when you know the exact number of elements you need to store. The same fixed size can be a limitation if you need a dynamic data structure that can grow or shrink during runtime.

Arrays provide constant-time access to elements using their index. This makes random access operations efficient. Insertion and deletion operations in the middle of the array can be inefficient since elements need to be shifted.

Arrays are memory-efficient because elements are stored in contiguous memory locations. If the array is large and you don't use all the elements, it can lead to wasted memory.

Copying Arrays

eazy bytes

One disadvantage of arrays is their fixed size limitation, as they can only store a specific number of elements. Arrays do not grow in size during runtime, which can be a problem. To address this limitation, Java provides the collection framework, which can automatically grow as needed. We are going to focus on collections later.

For now, imagine you have an array with 500 elements, and later on, you find out you only need to keep 300 of them. Unfortunately, you can't just discard the other 200 elements. Similarly, if you realize you need 550 elements, you can't simply tack on an additional 50 elements to the existing array. If you need to change the size of an existing array, the only solution is to create a new array with the desired length and copy the elements from the original array to the new one.

We can copy array elements from one array to another by using following common approaches,

- Using a loop
- Using the java.lang.System.arraycopy() method
- Using the java.util.Arrays.copyOf() method

Copying Arrays using loops

eazy bytes

Using a loop,

```java
// Original array
int[] sourceArray = {1, 2, 3, 4, 5};

// Creating a new array with a new length of 5 more elements
int[] targetArray = new int[sourceArray.length+5];

// Copying elements using a loop
for (int i = 0; i < sourceArray.length; i++) {
    targetArray[i] = sourceArray[i];
}
```

Copying Arrays using System.arraycopy()

eazy bytes

Using the java.lang.System.arraycopy() method,

```
// Original array
int[] sourceArray = {1, 2, 3, 4, 5};

// Creating a new array with a new length of 5 more elements
int[] targetArray = new int[sourceArray.length+5];

// Using System.arraycopy() to copy elements
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

Below is the signature of the System.arraycopy() method,

public static void arraycopy(Object sourceArray, int sourceStartPosition, Object destinationArray, int destinationStartPosition, int lengthToBeCopied)

Copying Arrays using Arrays.copyOf()

eazy bytes

Using the java.util.Arrays.copyOf() method,

```
// Original array
int[] sourceArray = {1, 2, 3, 4, 5};

// Using Arrays.copyOf to copy elements
int[] targetArray = Arrays.copyOf(sourceArray, sourceArray.length+5);
```

In the Arrays.copyOf() method, the first argument is the source array, and the second argument, newLength, represents the number of elements in the new array. Here's how it behaves based on different scenarios:

If newLength is less than the length of the source array, the returned array will be a truncated copy of the source array.
If newLength is greater than the length of the source array, the returned array will contain all elements from the original array. The extra elements will be initialized with default values based on the data type of the array.

If newLength is equal to the length of the source array, the returned array will contain the same number of elements as the source array.

Copying Arrays using Arrays.copyOf()

eazy bytes

Below is the list of copyOf() overloaded methods available in Arrays class,

- boolean[] copyOf(boolean[] original, int newLength)
- byte[] copyOf(byte[] original, int newLength)
- char[] copyOf(char[] original, int newLength)
- double[] copyOf(double[] original, int newLength)
- float[] copyOf(float[] original, int newLength)
- int[] copyOf(int[] original, int newLength)
- short[] copyOf(long[] original, int newLength)
- <T> T[] copyOf(T[] original, int newLength)

The Arrays class provides a copyOfRange() method, which allows you to copy a range of elements from one array to another. The method's signature for an int array is int[] copyOfRange(int[] original, int from, int to). It's important to note that the method is overloaded for all data types.

The parameters from and to represent the initial index (inclusive) and final index (exclusive) of the elements in the source array that are to be copied. These indices must fall within the range of the source array. Consequently, the length of the destination array can, at most, be equal to the length of the source array.

Supported Array operations

eazy bytes

In everyday programming, you may have to do many array operations like sorting, searching, comparing, and copying. The java.util.Arrays class is a utility class equipped with over 150 static convenience methods precisely for handling such array operations. Instead of crafting your own code for these operations, it's advisable to consult the Arrays class API documentation. You might discover a method within the class that perfectly accomplishes what you need.

Don't be overwhelmed by the extensive list of methods in this class. The reason for the abundance of methods in the Arrays class is to facilitate the same operations on arrays of various primitive and reference types. Each method typically has at least nine overloaded versions, covering all eight primitive-type arrays and one for the reference-type array.

Additionally, some operations can be executed on the entire array or on a specified range of elements, doubling the minimum number of methods for one array operation. This comprehensive approach ensures flexibility and compatibility across a diverse range of array types and use cases.

Converting Arrays

eazy bytes

An array in Java can be easily transformed into a String representation using the **toString()** method, converted into a List using **asList()**, and streamed using the **stream()** method.

```java
// Original array
int[] numbers = {1, 2, 3, 4, 5};

// Convert array to String
String arrayAsString = Arrays.toString(numbers);

// Original array
String[] fruits = {"Apple", "Banana", "Orange"};

// Convert array to List
List<String> fruitList = Arrays.asList(fruits);

// Original array
double[] prices = {2.5, 1.8, 3.0, 4.2};

// Convert array to Stream
DoubleStream priceStream = Arrays.stream(prices);
```

List and Stream are related to collections in Java. More details about them will be discussed as part of collections topic.

Sorting Arrays

eazy bytes

Here are examples demonstrating the use of the sort() and parallelSort() methods in the java.util.Arrays class to sort elements of an array. The sort() method is suitable for smaller arrays, while the parallelSort() method is designed for larger arrays:

```java
// Sorting an array of integers using sort()
int[] smallArray = {5, 2, 8, 1, 6};
Arrays.sort(smallArray);

// Displaying the sorted array
System.out.println("Sorted Array (using sort()): " +
Arrays.toString(smallArray));
```

```java
// Sorting a larger array of integers using parallelSort()
int[] largeArray = new int[10000];
Random random = new Random();

// Filling the array with random integers
for (int i = 0; i < largeArray.length; i++) {
    largeArray[i] = random.nextInt(1000000);
}

// Sorting the array in parallel
Arrays.parallelSort(largeArray);

// Displaying the sorted array
System.out.println("Sorted Array (using sort()): " +
Arrays.toString(largeArray));
```

Searching an Array

eazy bytes

To search for a element in an array, we can use the binarySearch() method. But to use binarySearch() method, the array must be sorted. This method returns the index of the search element if it exists in the array. However, if the element is not present, it returns a negative number, specifically equal to -(insertion point) - 1. The insertion point refers to the index where the element would be inserted to maintain the sorted order. This ensures that the returned value is a negative integer when the key is not found in the array.

```java
// Create an array to work with
int[] num = {36, 9, 42, 18, 73};
System.out.println("Original Array: " + Arrays.toString(num));

// Sort the array before using the binary search
Arrays.sort(num);
System.out.println("Array After Sorting: " + Arrays.toString(num));
// Array After Sorting: [9, 18, 36, 42, 73]

// Using binarySearch to find the index of 9
int index = Arrays.binarySearch(num, 9);
System.out.println("Found index of 9: " + index); //Found index of 9: 0

// Using binarySearch to find the index of 999
index = Arrays.binarySearch(num, 999);
System.out.println("Found index of 999: "+ index); //Found index of 999: -6
```

Filling an Array

eazy bytes

Filling arrays involves populating all or part of an array with specific values. Java provides several methods to fill arrays with predetermined values. Here are some commonly used approaches:

Using Arrays.fill() Method: The Arrays.fill() method allows you to fill the entire array or a specified range with a given value.

```
// Example: Filling an array with a specific value
int[] scores = new int[10];
Arrays.fill(scores, 100); // Fills the entire array with the value 100
```

Using Arrays.setAll() Method (Java 8 and later): Introduced in Java 8, the Arrays.setAll() method allows you to fill an array using a lambda function,

```
// Example: Filling an array with index values
int[] indices = new int[5];
Arrays.setAll(indices, i -> i);
```

Multidimensional/Nested Arrays

eazy bytes

A multidimensional or nested array is an array of arrays, where each element of the array can also be an array. In other words, it is an array of arrays. The most common multidimensional array is a two-dimensional array, which is essentially a table or a grid with rows and columns.

Below is the general syntax to declare an multi-dimensional array,

```
dataType[1st Dimension][2nd Dimension]....[Nth Dimension] arrayName;
```

Below is the general syntax to initialize an multi dimensional array,

```
arrayName = new DataTye[length 1][length 2]....[length N];

dataType[1st Dimension][2nd Dimension]....[Nth Dimension] arrayName = new dataTye[length 1][length 2]....[length N];
```

Multidimensional array can be a 2D array, a 3D array, a 4D array, where D stands for Dimension. The higher the dimension, the more difficult it is to access and store elements inside a multi dimensional array.

`Arrays.deepToString` method needs to be used to convert an nested array into a String object

Two-Dimensional or 2D Array

eazy bytes

Numerous computer games incorporate objects positioned within a two-dimensional space. In these types of games, a common approach involves employing a two-dimensional array to represent objects, with indices such as i and j corresponding to specific cells in the array.

The initial index denotes the row number, while the second index denotes the column number. In Java, a two-dimensional array can be visualized as a straightforward table or matrix with defined columns and rows, making it a fundamental structure for handling spatial data in gaming applications.

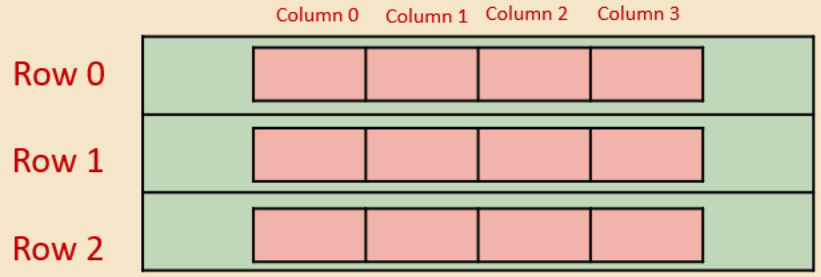
Below is the general syntax to declare an two dimensional array,

```
dataType[][]  arrayName; (or)
dataType [][]arrayName; (or)
dataType  arrayName[][]; (or)
dataType []arrayName[];
```

Below is an example to declare and initialize an two dimensional array,

```
// 2D array with 3 rows and 4 columns
int[][] 2DArray = new int[3][4];
```

Size of the 2D array is lenght1 * lenght2. For the above 2D array, the number of elements it can store will be 3*4=12



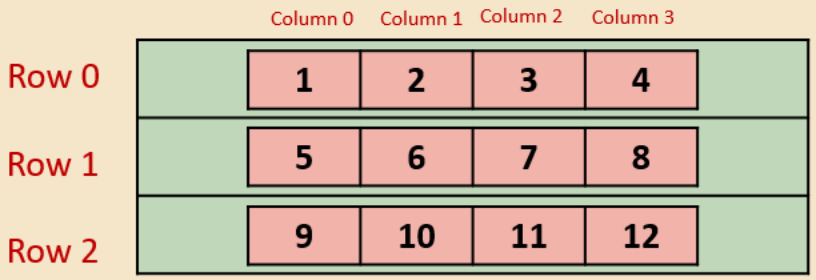
Representation of the above declared `2DArray`

Two-Dimensional or 2D Array

eazy bytes

Lets take an example of a 2D array and try to visualize how it is going to store data inside it,

```
int[][] myArray = {    {1, 2, 3, 4} ,
                       {5, 6, 7, 8} ,
                   {9, 10, 11, 12}
                };
```



Representation of the myArray

In this example, myArray[0][0] would access the first element (1) in the first row and first column, while myArray[1][2] would access the element 7 in the second row and third column.

Row/Column	Column 1	Column 2	Column 3	Column 4
Row 1	myArray[0][0] = 1	myArray[0][1] = 2	myArray[0][2] = 3	myArray[0][3] = 4
Row 2	myArray[1][0] = 5	myArray[1][1] = 6	myArray[1][2] = 7	myArray[1][3] = 8
Row 3	myArray[2][0] = 9	myArray[2][1] = 10	myArray[2][2] = 11	myArray[2][3] = 12

Addition of 2 Matrices

eazy bytes

$$\begin{bmatrix} 4 & 7 & 9 \\ 5 & 7 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 8 & 2 \\ 2 & 6 & 1 \end{bmatrix}$$

The provided Java code demonstrates the creation of two matrices using a 2D array, followed by the creation of another matrix to store the sum of the two original matrices. The addition of the matrices is performed using nested loops, and the result is printed.

```java
//creating two matrices using 2D array
int array1[][]={{4,7,9},{5,7,0}};
int array2[][]={{3,8,2},{2,6,1}};

//creating another matrix to store the sum of two matrices
int sumArray[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<array1.length;i++){
    for(int j=0;j<array1[i].length;j++){
        sumArray[i][j]=array1[i][j]+array2[i][j];
        System.out.print(sumArray[i][j]+" ");
    }
    System.out.println();//new line
}
```

Jagged Arrays

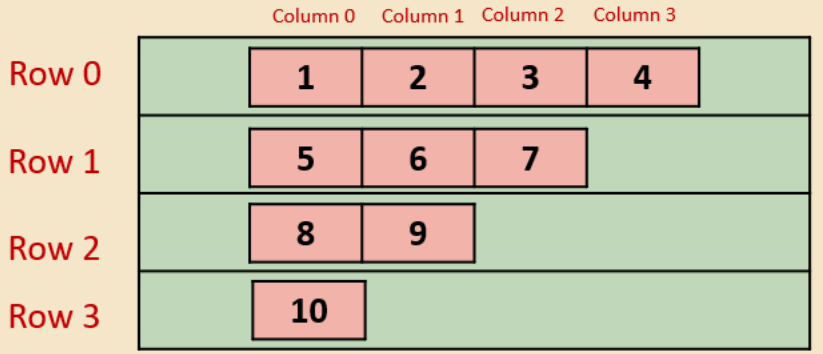
eazy bytes

In the previous examples, we consistently worked with two-dimensional arrays where each row had a fixed number of columns, maintaining uniformity across subarrays. However, Java provides the concept of Jagged Arrays, allowing a two-dimensional array to contain subarrays with varying sizes.

This flexibility enables different rows to have a different number of elements, breaking away from the rigid structure of uniformly sized subarrays.

Below is an example of a jagged array

```
// Declaration and initialization of a jagged array
int[][] jaggedArray = {
                        {1, 2, 3, 4},
                        {5, 6, 7},
                        {8, 9},
                        {10}
                      };
```



Representation of the Jagged Array

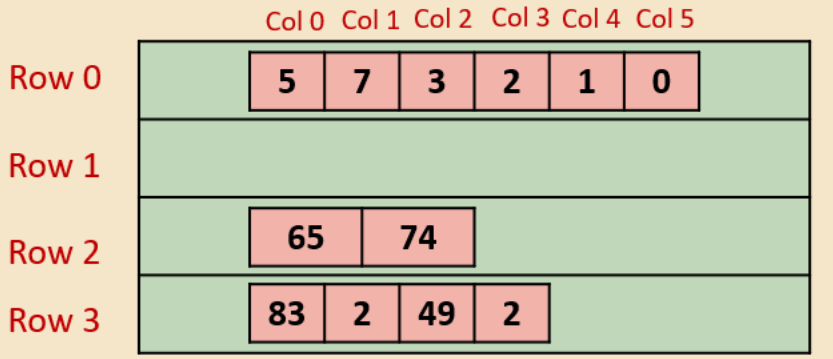
Jagged arrays are useful in scenarios where you need varying lengths for different sets of data, providing a more flexible structure than regular rectangular arrays.

Jagged Arrays

eazy bytes

Another example of Jagged Arrays,

```
// a jagged array with 4 rows and columns are not defined
int[][] jaggedArray = new int[4][];

// array1 with 5 columns
int[] array1 = { 5, 7, 3, 2, 1, 0 };
// empty array2
int[] array2 = {};
// array3 with 2 columns
int[] array3 = { 65, 74 };
// array4 with 4 columns
int[] array4 = { 83, 2, 49, 2 };
// adding all the arrays to jaggedArray
jaggedArray[0] = array1;
jaggedArray[1] = array2;
jaggedArray[2] = array3;
jaggedArray[3] = array4;

// printing jaggedArray
System.out.println(Arrays.deepToString(jaggedArray));
```



Representation of the Jagged Array

Three-Dimensional or 3D Array

eazy bytes

In Java, a 3D array is a data structure that extends the concept of a 2D array to three dimensions. It can be visualized as a collection of 2D arrays, where each 2D array is a "slice" or a two-dimensional layer within the 3D array. To declare and work with 3D arrays in Java, you use three sets of square brackets.

Here's a basic overview of how to declare, initialize, and access elements in a 3D array:

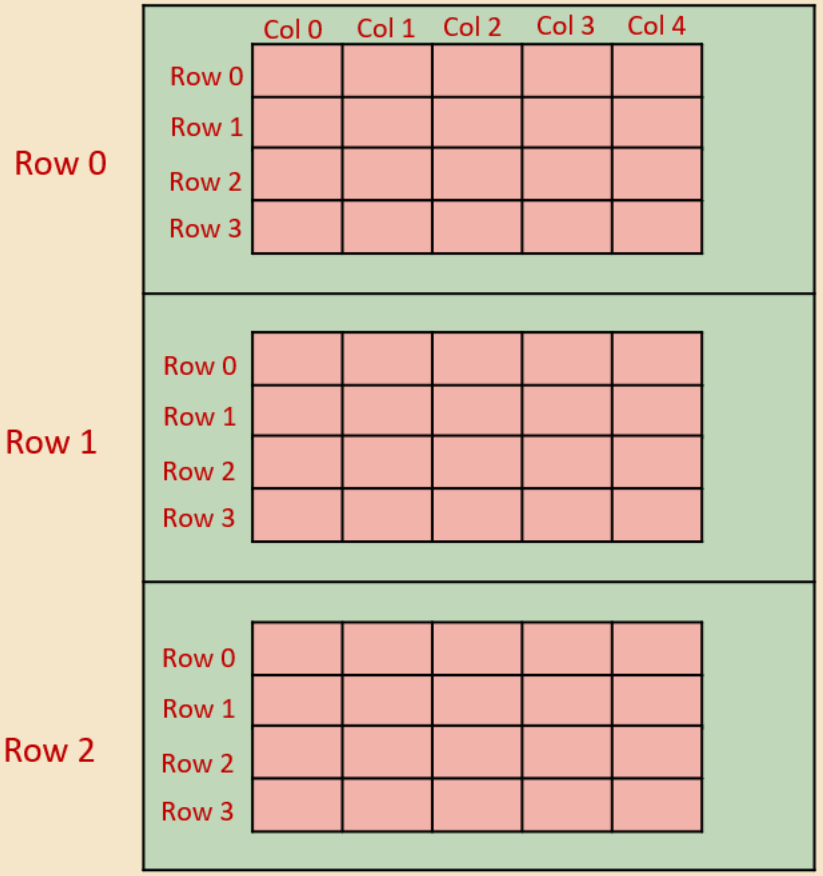
```
// Declaration and initialization of a 3D array
dataType[][][] arrayName = new dataType[size1][size2][size3];

// Example with length values
int[][][] threeDArray = new int[3][4][5];
```

In this example, threeDArray is a 3D array with dimensions 3x4x5.
It means there are three 2D arrays, each with four rows and five columns.

Accessing elements in a 3D array involves specifying the indices for each dimension:

```
// Accessing an element
int element = threeDArray[i][j][k];
```

Here, i, j, and k represent the indices for the three dimensions.



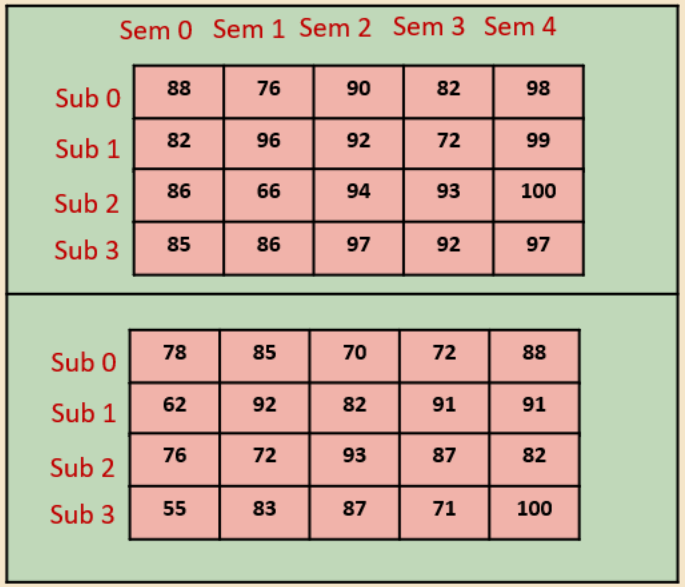
Representation of above declared **threeDArray**

Three-Dimensional or 3D Array

eazy bytes

Let's consider below example data and store it using 3D array,

```
int noOfStudents = 2;
int subjects = 4;
int totalSemesters = 5;

// Student 1 data
int[][] student1 = {
      { 88, 76, 90, 82, 98 },
      { 82, 96, 92, 72, 99 },
      { 86, 66, 94, 93, 100 },
      { 85, 86, 97, 92, 97 },
    };

// Student 2 data
int[][] student2 = {
      { 78, 85, 70, 72, 88 },
      { 62, 92, 82, 91, 91 },
      { 76, 72, 93, 87, 82 },
      { 55, 83, 87, 71, 100 },
    };
```



Representation of Student Data

```
// array for storing data of all three students
int[][][] studentArray = new int[noOfStudents][subjects][totalSemesters];
// storing data of students into 3D array
studentArray[0] = student1;
studentArray[1] = student2;
```