Arrays Introduction

Welcome to the Arrays section of the Java Master Class, and congratulations for making it this far.

I hope that as you are progressing through more and more of the course, you're feeling more confident and getting better at writing Java code.

There's still a lot to learn, and I'll admit that many of my favorite features of Java are still ahead.

In the last two sections, we learned about different ways to create classes, which can have any number and type of data elements.

Arrays Introduction

What I haven't really discussed though, is a way to have multiple values, all of the same type.

We had a taste of this kind of problem in the Bill's Burger challenge.

We had several toppings for our burger, and we had to create individual attributes for each. This led to inefficient code and other limitations. For example, what if we needed ten toppings? There are much better ways to handle this.

Java provides us with many types of containers to store multiple values of the same type.

These start with the most basic, which is the array, and that's what this section will cover.

Arrays

Let's look at ways to store and manipulate multiple values of the same type.

The most common way to do this in Java is with an array.

Arrays

An array is a data structure that allows you to store a sequence of values, all of the same type.

You can have arrays for any primitive type, like ints, doubles, booleans, or in fact, any of the 8 primitives we've learned about.

You can also have arrays for any class.

Arrays

Elements in an array are indexed, starting at 0.

If we have an array storing five names, conceptually, it looks as shown here.

Index	0	1	2	3	4
Stored values in an array with 5 elements	"Andy"	"Bob"	"Charlie"	"David"	"Eve"

The first element is at index 0 and is Andy.

The last element in this array is at index 4 and has the String value Eve.

Declaring an Array

When you declare an array, you first specify the type of the elements you want in the array.

Then you include square brackets in the declaration, which is the key for Java to identify the variable as an array.

The square brackets can follow the type as shown in the first two examples.

You can also put the square brackets after the variable name as shown in the last example.

The first approach is much more common.

Note that you don't specify a size in the array declaration itself.

Array Variable Declaration
`int[] integerArray;`
`String[] nameList;`
`String courseList[];`

Instantiating an Array

Array Creation	Object Creation
`int[] integerArray = new int[10];`	`StringBuilder sb = new StringBuilder();`

One way to instantiate the array is with the new keyword, similar to what we've seen when creating instances of classes.

Looking at the left-hand side of this slide, we have an array declaration on the left of the equals sign and then an array creation expression on the right side.

On the right hand side of the slide, I'm showing a typical class object creation statement.

Instantiating an Array

Array Creation	Object Creation
`int[] integerArray = new int[10];`	`StringBuilder sb = new StringBuilder();`

They look pretty similar, but there are two major differences when creating arrays.

Square brackets are required when using the new keyword and a size is specified between them. So, in this example, there will be 10 elements in the array.

An array instantiation doesn't have a set of parentheses, meaning we can't pass data to a constructor for an array.

Using parentheses with an array instantiation gives you a compiler error.

Invalid Array Creation – Compile Error because of ()
`int[] integerArray = new int[10]();`

An Array is NOT Resizable.

The size of an array, once created, is fixed.

In this case, integerArray will have 10 elements.

Array Creation

```
int[] integerArray = new int[10];
```

You can't change the size of an array after the array is instantiated.

You can't add or delete elements. You can only assign values to one of the ten elements in this array, in this example.

Note that Java has other capabilities to store elements that can be resized. We'll be looking at those in the next section.

For now its important to understand these basic arrays.

The array initializer

An array initializer makes the job of instantiating and initializing an array much easier.

The array initializer

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

In this example, you can see I still use the new keyword and have int with the square brackets.

But here, I specify the values I want the array to be initialized to, in a comma-delimited list, within curly braces.

Because these values are specified, the length of the array can be determined by Java, so I don't then need to specify the size of the array in square brackets.

And actually, Java provides an even simpler way to do this.

The array initializer as an anonymous array

Java allows us to drop the new int[] with brackets from the expression, as I'm showing here.

This is known as an anonymous array.

Here, I'm showing examples for both an int array as well as a String array.

The array initializer
`int[] firstFivePositives = {1, 2, 3, 4, 5};`
`String[] names = {"Andy", "Bob", "Charlie", "David", "Eve"};`

An anonymous array initializer can only be used in a declaration statement.

What is an array, really?

An array is a special class in Java.

It's still a class.

The array, like all other classes, ultimately inherits from java.lang.Object.

Array initialization and default element values

When you don't use an array initializer statement, all array elements get initialized to the default value for that type.

For primitive types, this is **zero** for any kind of **numeric primitive**, like int, double, or short.

For **booleans**, the default value will be **false**.

And for any **class** type, the elements will be initialized to **null**.

The Enhanced For Loop, the For Each Loop

This loop was designed to walk through elements in an array or other collection types.

It processes **one element at a time**, from the **first** element to the **last**.

Here is the syntax for the two types of for loop statements, side by side.

Enhanced For Loop	**Basic For Loop**

`for (declaration : collection) {`
`    // block of statements`
`}`
	`for (init; expression; increment) {`
	`    // block of statements`
	`}`

The enhanced for loop only has two components, versus three defined in the parentheses after the for keyword.

The Enhanced For Loop, the For Each Loop

Enhanced For Loop	Basic For Loop

`for (declaration : collection) {`
`    // block of statements`
`}`
	`for (init; expression; increment) {`
	`    // block of statements`
	`}`

It's important to notice that the separator character between components is a **colon** and not a semicolon for the enhanced For Loop.

The **first** part is a **declaration expression** which includes the type and a variable name. This is usually a local variable with the same type as an element in the array.

And the **second** component is the **array** or some other collection variable.

java.util.Arrays

Java's array type is very basic, it comes with very little built-in functionality.

It has a single property or field, named length.

And it inherits java.util.Object's functionality.

Java provides a helper class named java.util.Arrays, providing the common functionality you'd want for many array operations.

These are static methods on Arrays, they are class methods, not instance methods.

Printing elements in an array using Arrays.toStrin

The toString method in this helper class prints out all the array elements, comma delimited, and contained in square brackets.

```
String arrayElementsInAString = Arrays.toString(newArray);
```

The output from this method is shown here conceptually.

It prints the element at index 0 first, followed by a comma, then the element at index 1 next, a comma, and so on, until all elements are printed.

```
[ e[0], e[1], e[2], e[3], ... ]
```

Why use arrays?

We use arrays to manage many items of the same type.

Some common behaviors for arrays include sorting, initializing values, copying the array, and finding an element.

As I mentioned in the last video, this behavior isn't on the array instance itself, but rather provided on a helper class, java.util.Arrays.

Finding a match

There are different algorithms for searching and matching elements in arrays.

Searching Sequentially

I think you can probably imagine that if you were going to start writing code to do this, you might start looping from the first to the last element of an array, checking each element to see if it matches a value that you're looking for.

If you find a match, you'd stop looping and return that a match was found, either with the position you found the element at or just a boolean value: true if it was found, and false if not.

This is called a **linear** or **sequential search** because you're stepping through the elements one after another.

If your elements are sorted, using this type of linear search is unnecessarily inefficient.

Using intervals to Search

You split each section up, testing the values at the boundaries, and based on that, split again into smaller sections, narrowing the number of elements to test each time.

This type of searching in software is called **interval searching**.

Within these two categories, sequential and interval, there are numerous existing algorithms.

One of the most common interval searches is the **binary search**, which is why Java provides this search on so many of its collection classes.

In this search, **intervals** are continually **split into two**, hence the word binary.

Arrays.binarySearch

The static method, binarySearch, is in the Arrays class.

We can use this method to test if a value is already in our array, but there are some **important** things to remember.

- First, the array has to be **sorted**.
- Second, if there are duplicate values in the array, there's no guarantee which one it'll match on.
- Finally, elements must be comparable. Trying to compare instances of different types will lead to errors and invalid results.

Arrays.binarySearch

This method returns:

- **The position of a match**, if found.
- It returns a **-1** when **no match** was found.
- It's important to remember that a positive number **may not be the position of the first match.**
- If your array has duplicate values and you need to find the first element, other methods should be used.

The Array Challenge

Create a program using arrays that **sorts** a list of **integers** in **descending order**. Descending order means from highest value to lowest.

In other words, if the array has the values 50, 25, 80, 5, and 15, your program should return an array with the values in the descending order: 80, 50, 25, 15, and 5.

First, create an **array** of **randomly generated integers**.

Print the array before you sort it.

And print the array after you sort it.

You can choose to create a new sorted array or just sort the current array.

The Array Challenge

Create a program using arrays that **sorts** a list of **integers** in **descending order**. Descending order means from highest value to lowest.

In other words, if the array has the values 50, 25, 80, 5, and 15, your program should return an array with the values in the descending order: 80, 50, 25, 15, and 5.

First, create an **array** of **randomly generated integers**.

Print the array before you sort it.

And print the array after you sort it.

You can choose to create a new sorted array or just sort the current array.

Arrays Recap

Lets recap what we've learned so far about arrays and discuss some of the common errors that occur when using them.

Arrays Recap

- An array is a data structure that allows us to store multiple values of the same type in a single variable.
- The default values of numeric array elements are set to zero.
- Arrays are zero-indexed, so an array with n elements is indexed from 0 to n - 1. For example, 10 elements would have the index range from 0 through 9.
- If we try to access an index that is out of range, Java will give us an **ArrayIndexOutOfBoundsException**, which indicates that the index is out of range, in other words, out of bounds.
- To access array elements, we use square braces. This is also known as the array access operator.

Arrays Recap - Creating a New Array

```java
int[] array = new int[5];
```

This array contains the elements from array[0] through array[4].

It has 5 elements and an index range from 0 to 4.

The new keyword is used to create the array and initialize the array elements to their default values.

In this example, all the array elements will default to zero because it is an int array.

For boolean arrays, elements would be initialized to false.

If you use String or other objects, they would be set to a null reference.

Arrays Recap - Creating a new Array

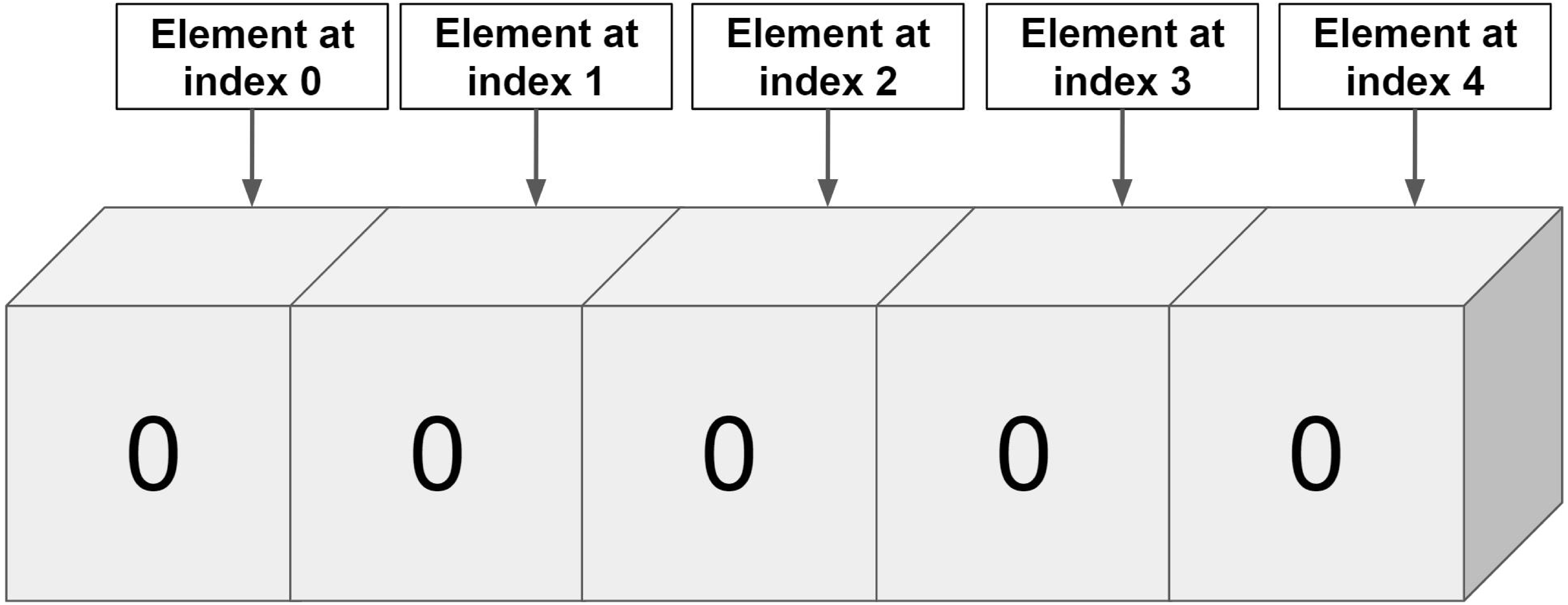
We can also initialize an array inline using an array initializer block, as shown here.

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

I define the values for the array in curly braces, in the order I want them assigned, each value separated by a comma.

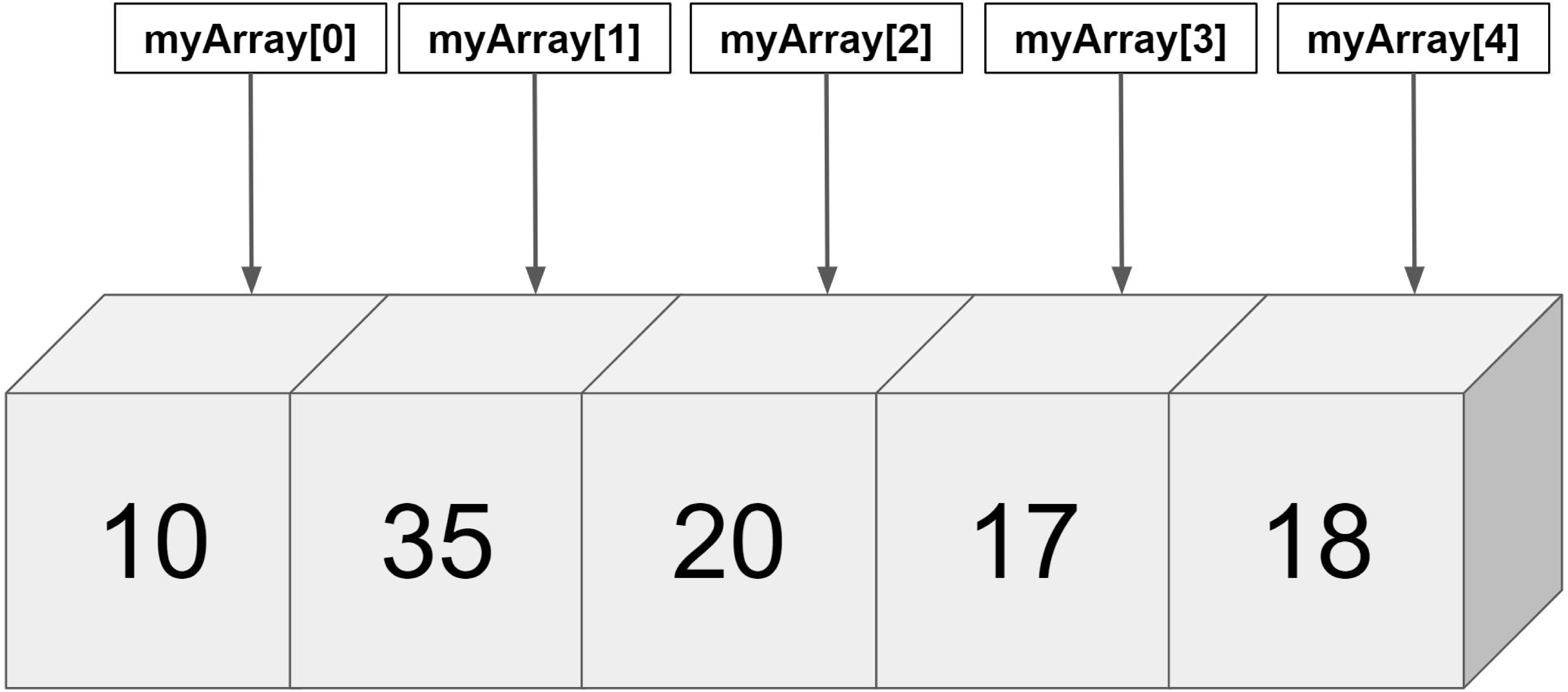
Arrays Recap

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



Arrays Recap

```
int[] myArray = {10, 35, 20, 17, 18};
```



First Common Error

```
int[] myArray = {10, 35, 20, 17, 18};
myArray[5] = 55;      // out of bounds
```

- Accessing index out of range will cause error in other words ArrayIndexOutOfBoundsException
- We have 5 elements and index range is 0 to 4

Element at index -1
out of bounds
Element at index 0
Element at index 4
Element at index 5
out of bounds
10
35
20
17
18

Second Common Error

```java
int[] myArray = {10, 35, 20, 17, 18};

for (int i = 1; i < myArray.length; i++) {
    System.out.println("value = " + myArray[i]);
}
```

OUTPUT:

```
value= 35
value= 20
value= 17
value= 18
```

Third Common Error

```java
int[] myArray = {10, 35, 20, 17, 18};

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

OUTPUT:

```
value = 10     // printed when i = 0     -> condition 0 <= 5 is true
value = 35     // printed when i = 1     -> condition 1 <= 5 is true
value = 20     // printed when i = 2     -> condition 2 <= 5 is true
value = 17     // printed when i = 3     -> condition 3 <= 5 is true
value = 18     // printed when i = 4     -> condition 4 <= 5 is true
ArrayIndexOutOfBoundsException when i = 5 since condition 5 <= 5 is true
```

Third Common Error

```
int[] myArray = {10, 35, 20, 17, 18};

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

OUTPUT:

```
value = 10     // printed when i = 0     -> condition 0 <= 5 is true
value = 35     // printed when i = 1     -> condition 1 <= 5 is true
value = 20     // printed when i = 2     -> condition 2 <= 5 is true
value = 17     // printed when i = 3     -> condition 3 <= 5 is true
value = 18     // printed when i = 4     -> condition 4 <= 5 is true
```

Use Enhanced For Loop to avoid some of these erro

Use the enhanced for loop if you're looping through elements from first to last, and you want to process them one at a time, and you're not setting or assigning values

Enhanced For Loop (Preferred for this kind of processing)

```java
int[] myArray = {10, 35, 20, 17, 18};

for (int myInt : myArray) {
    System.out.println("value = " + myInt);
}
```

Traditional For Loop

```java
int[] myArray = {10, 35, 20, 17, 18};

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

Reference Types vs. Value Types

In a previous video, I talked about the differences between a Reference vs. an Object, vs. an Instance, vs. a Class.

I want to revisit this a little and talk about why this matters when we're talking about arrays.

When you assign an object to a variable, the variable becomes a reference to that object.

This is true of arrays, but the array has yet another level of indirection if it's an array of objects.

This means every array element is also a reference.

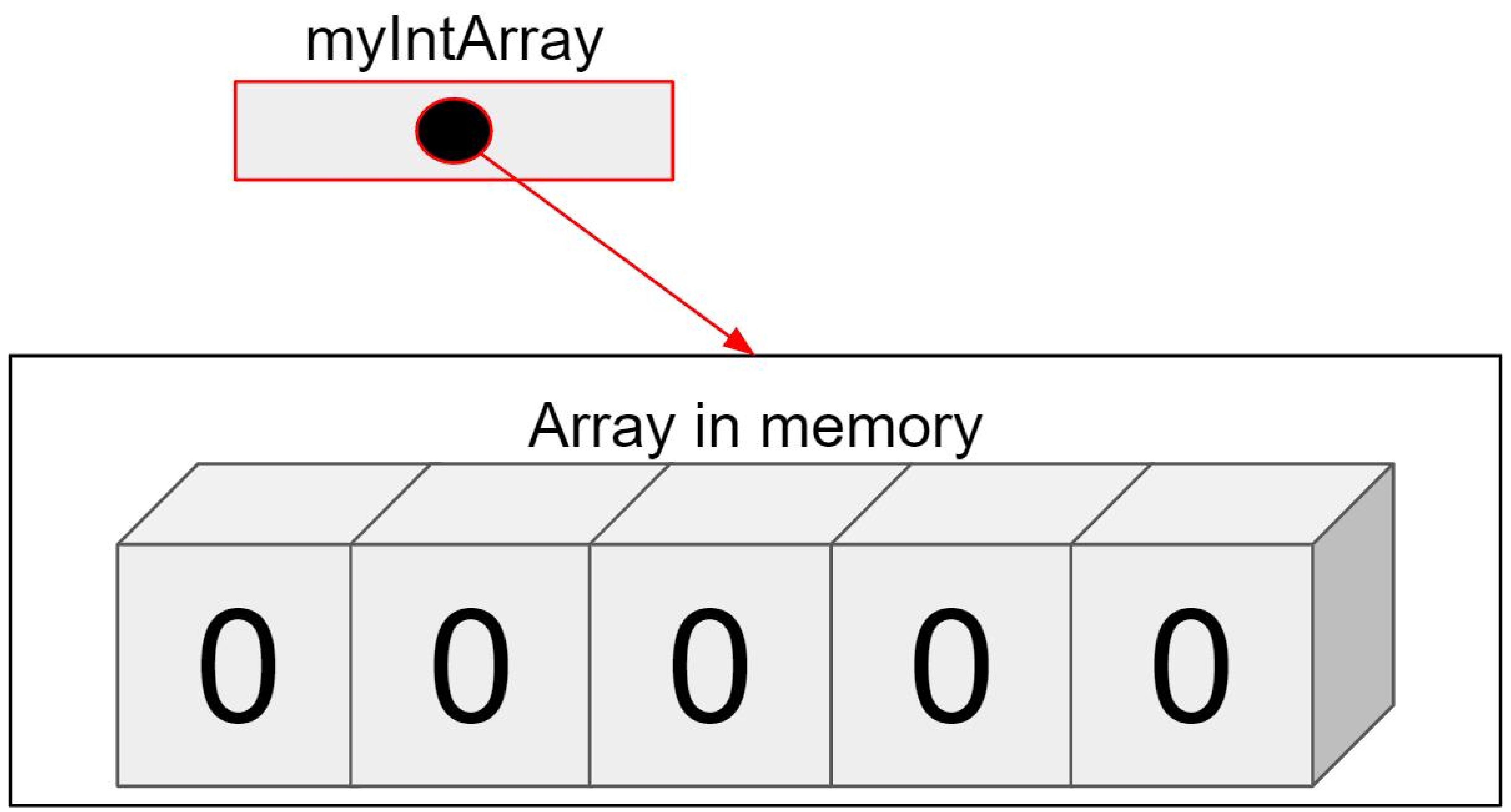
Reference Types vs. Value Types

```
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



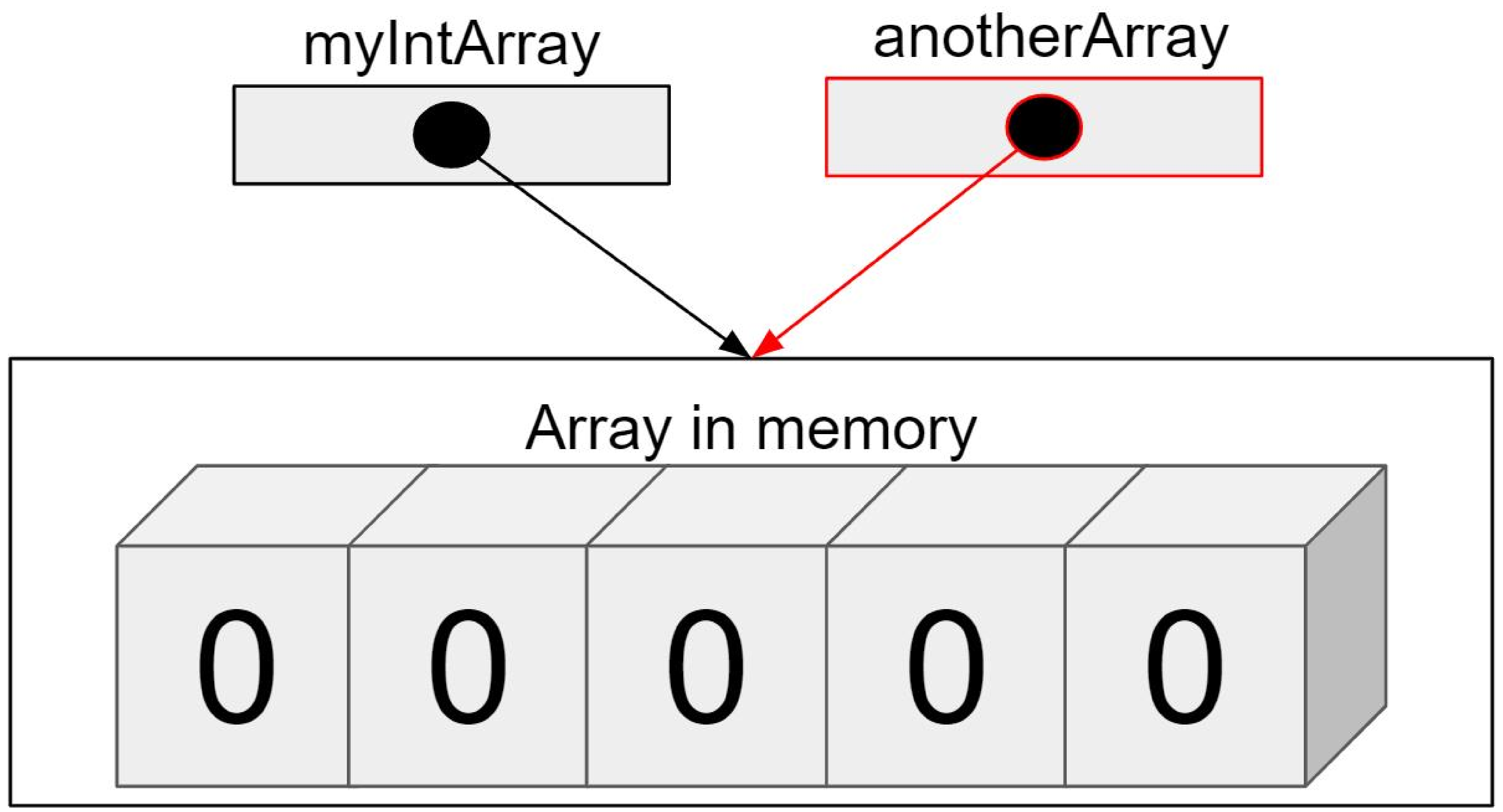
Reference Types vs. Value Types

```java
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



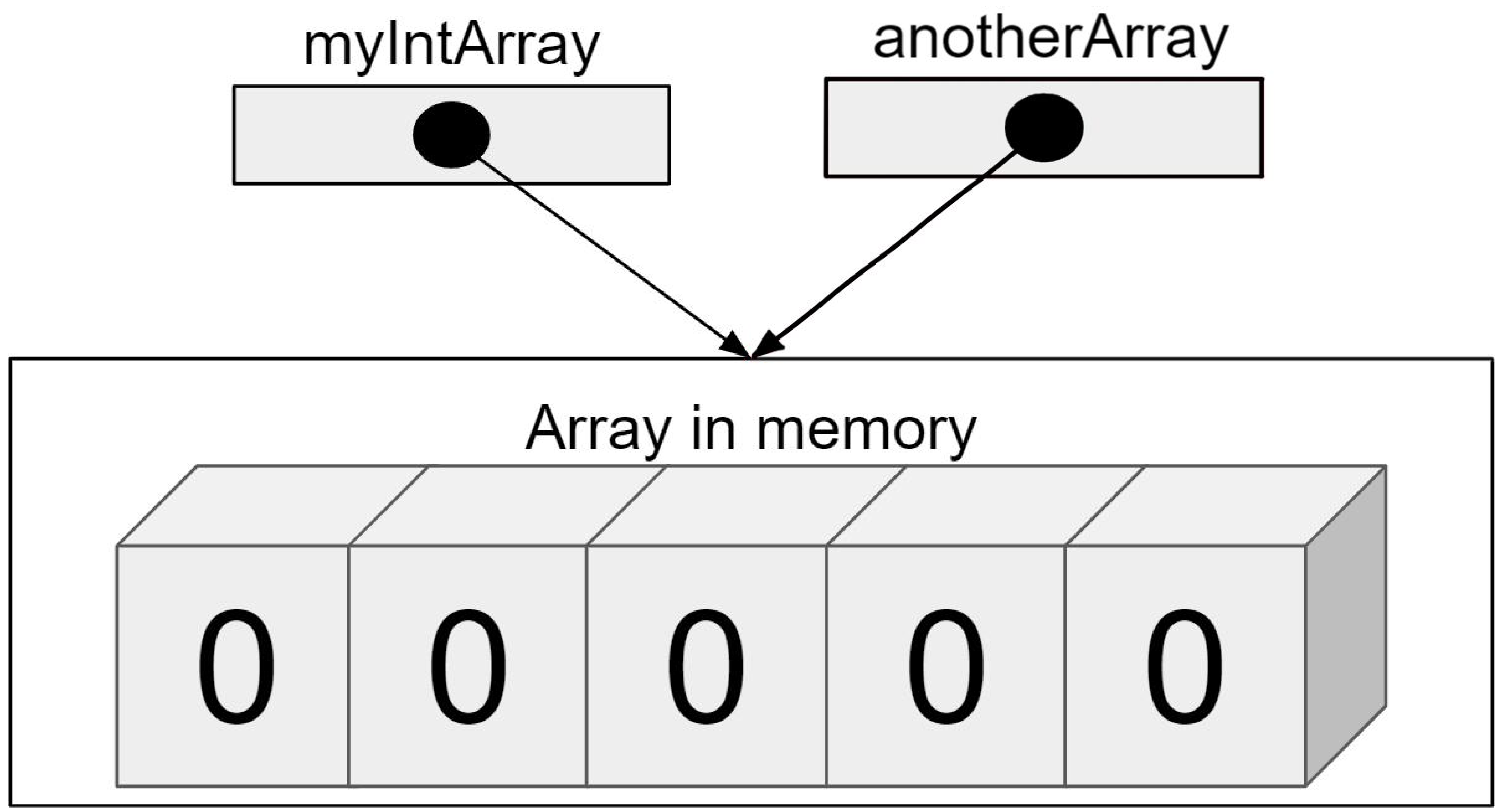
Reference Types vs. Value Types

```
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



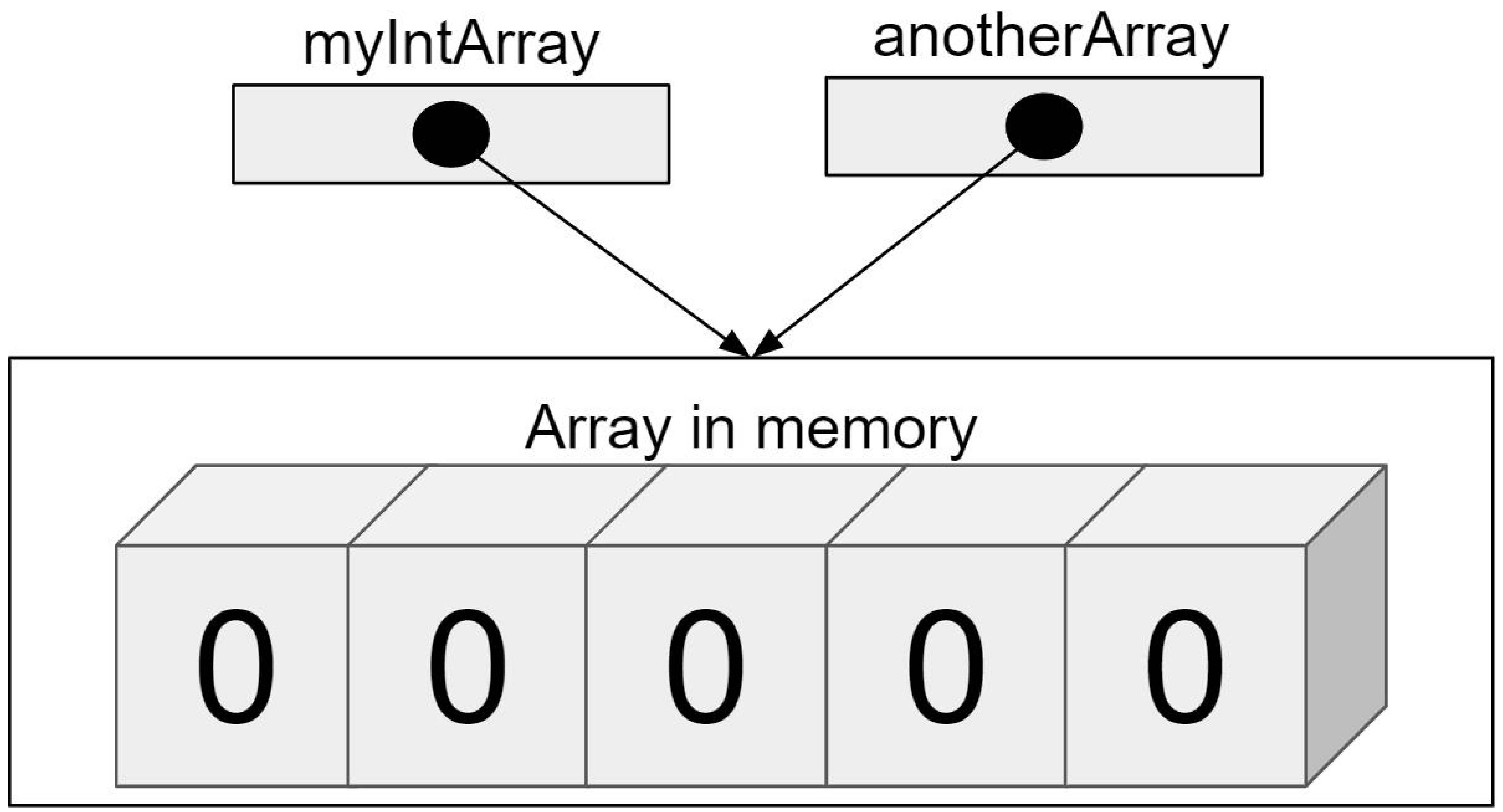
Reference Types vs. Value Types

```java
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



OUTPUT:

```
myIntArray= 0 0 0 0 0
anotherArray= 0 0 0 0
0
```

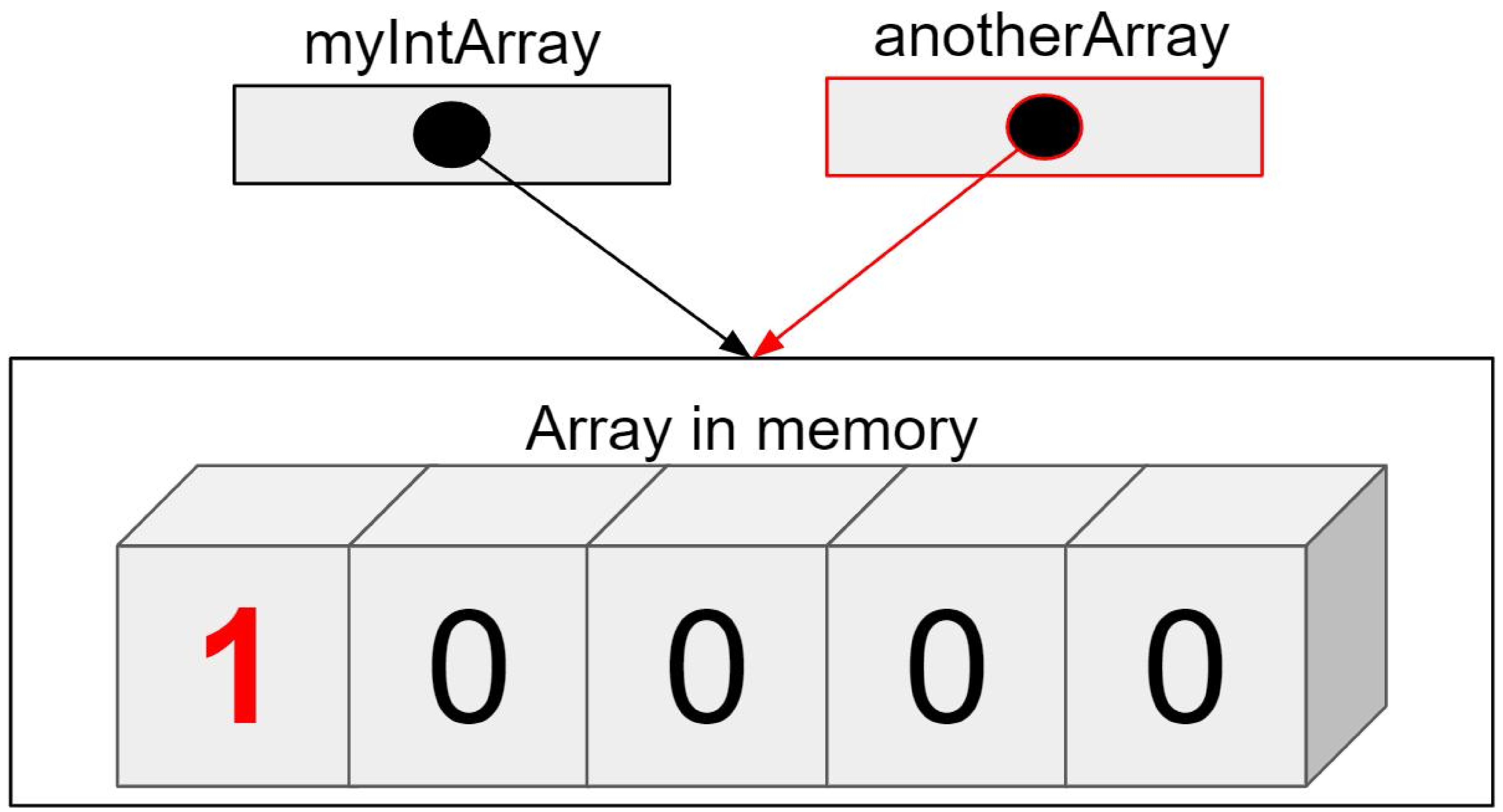
Reference Types vs. Value Types

```java
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



OUTPUT:

```
myIntArray= 0 0 0 0 0
anotherArray= 0 0 0 0
0
```

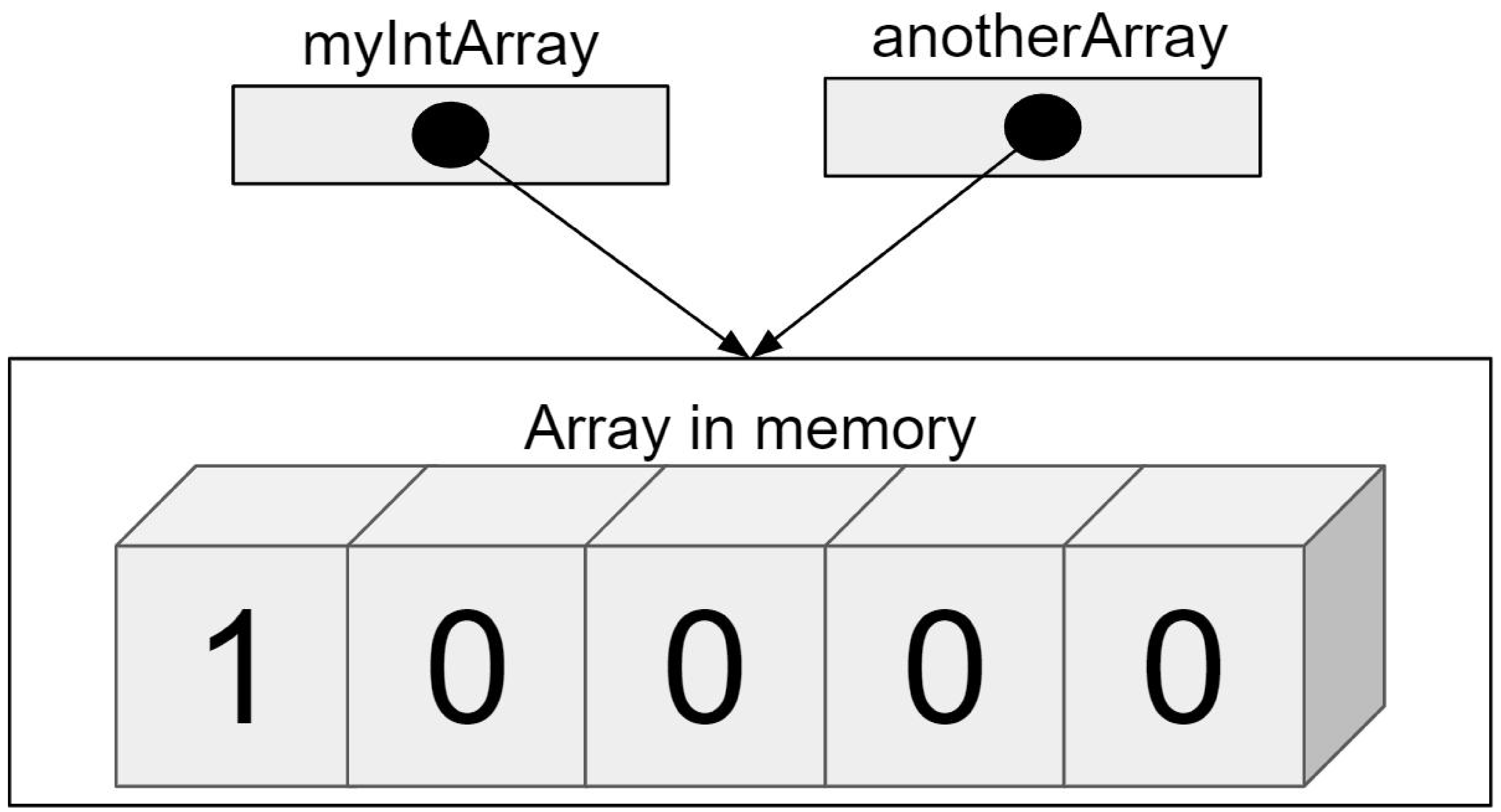
Reference Types vs. Value Types

```java
int[] myIntArray = new int[5];
int[] anotherArray = myIntArray;

System.out.println("myIntArray= " + Arrays.toString(myIntArray));
System.out.println("anotherArray= " + Arrays.toString(anotherArray));

anotherArray[0] = 1;

System.out.println("after change myIntArray= " + Arrays.toString(myIntArray));
System.out.println("after change anotherArray= " + Arrays.toString(anotherArray));
```



OUTPUT:

```
myIntArray= 0 0 0 0 0
anotherArray= 0 0 0 0 0
after change myIntArray= 1 0 0 0 0
after change anotherArray= 1 0 0 0
0
```

Arrays as method parameters

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

}
```

Notice here that the parameter to the main method is an array of String.

This means we can pass an array of Strings to this method when it's called.

Or, if we use this method as the entry point to our application, we can pass data on the command line to this method.

Up until now, I've only shown you this particular method signature.

Variable arguments (varargs)

But this signature can be written in a slightly different way.

We can replace the brackets after the String type, which we know tells us this method will take an array of String.

And we can instead replace that with three periods.

This is a special designation for Java that means Java will take zero, one, or many Strings as arguments to this method and create an array with which to process them in the method.

```java
public static void main(String... args) {

}
```

Variable arguments (varargs)

The array will be called args and be of type String.

So, what's the difference then?

The difference is minor within the method body but significant to the code that calls the method.

```
public static void main(String... args) {

}
```

When can you use variable arguments (varargs) ?

There can be only one variable argument in a method.

The variable argument must be the last argument.

Minimum Element Challenge

- Write a method called **readIntegers** that reads a comma delimited list of numbers entered by the user from the console, and then returns an **array** containing the numbers that were entered.
- Next, write a method called findMin that takes the array as an argument and returns the **minimum value** found in that **array**.

Minimum Element Challenge

- In the **main method**:
    - Call the **method readIntegers** to get the array of integers from the user, and print these out using a method found in java.util.Arrays.
    - Next, call the **findMin method**, passing the **array**, returned from the call to the **readIntegers method**.
    - Print the **minimum element** in the **array**, which should be returned from the **findMin** method.
- A tip here. Assume that the user will only enter numbers, so you don't need to do any validation for the console input.

The Reverse Array Challenge

The challenge is to write a method called reverse that takes an int array as a parameter.

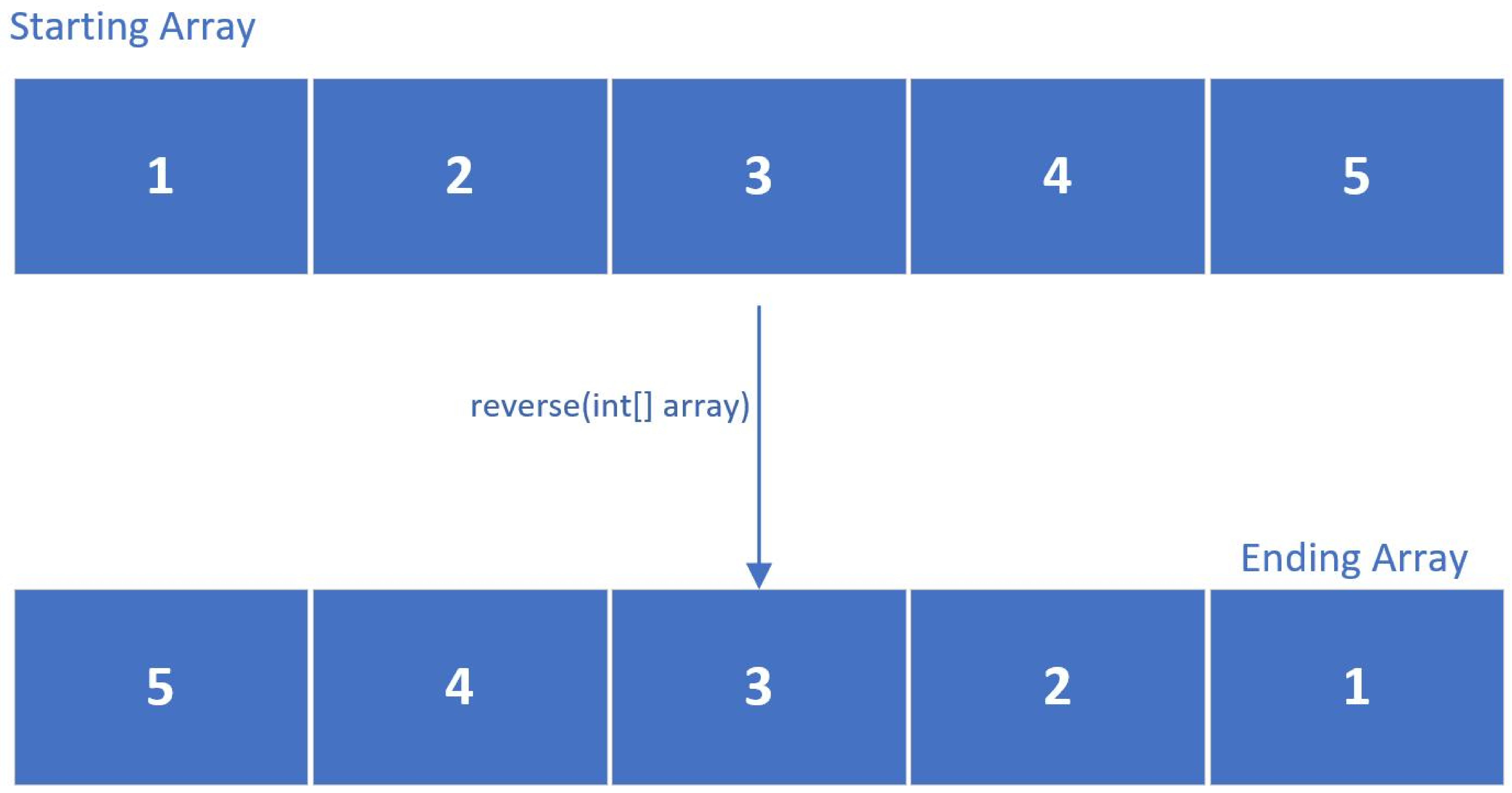
In the main method, call the reverse method and print the array before and after the reverse method is called.

To reverse the array, you have to swap the elements so that the first element is swapped with the last element and so on.

For example, if the array contains the numbers 1,2,3,4,5, then the reversed array should be, 5,4,3,2,1.

The Reverse Array Challenge

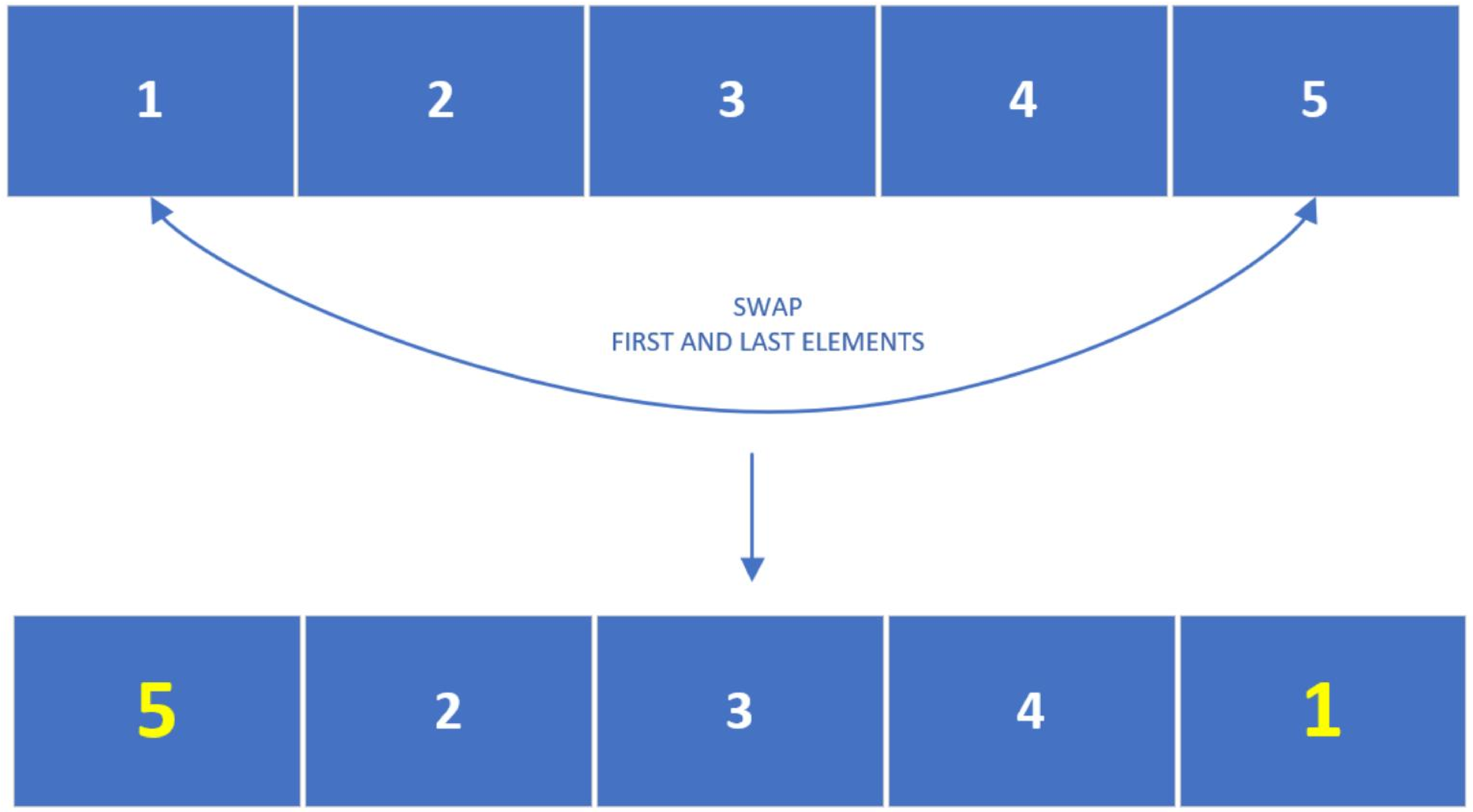
This shows the array before we start reversing the values and the end result we want to achieve:



The Reverse Array Challenge

How would we go about doing this?

We could start swapping the elements at positions 0 and 4, to get this interim result after the first iteration.



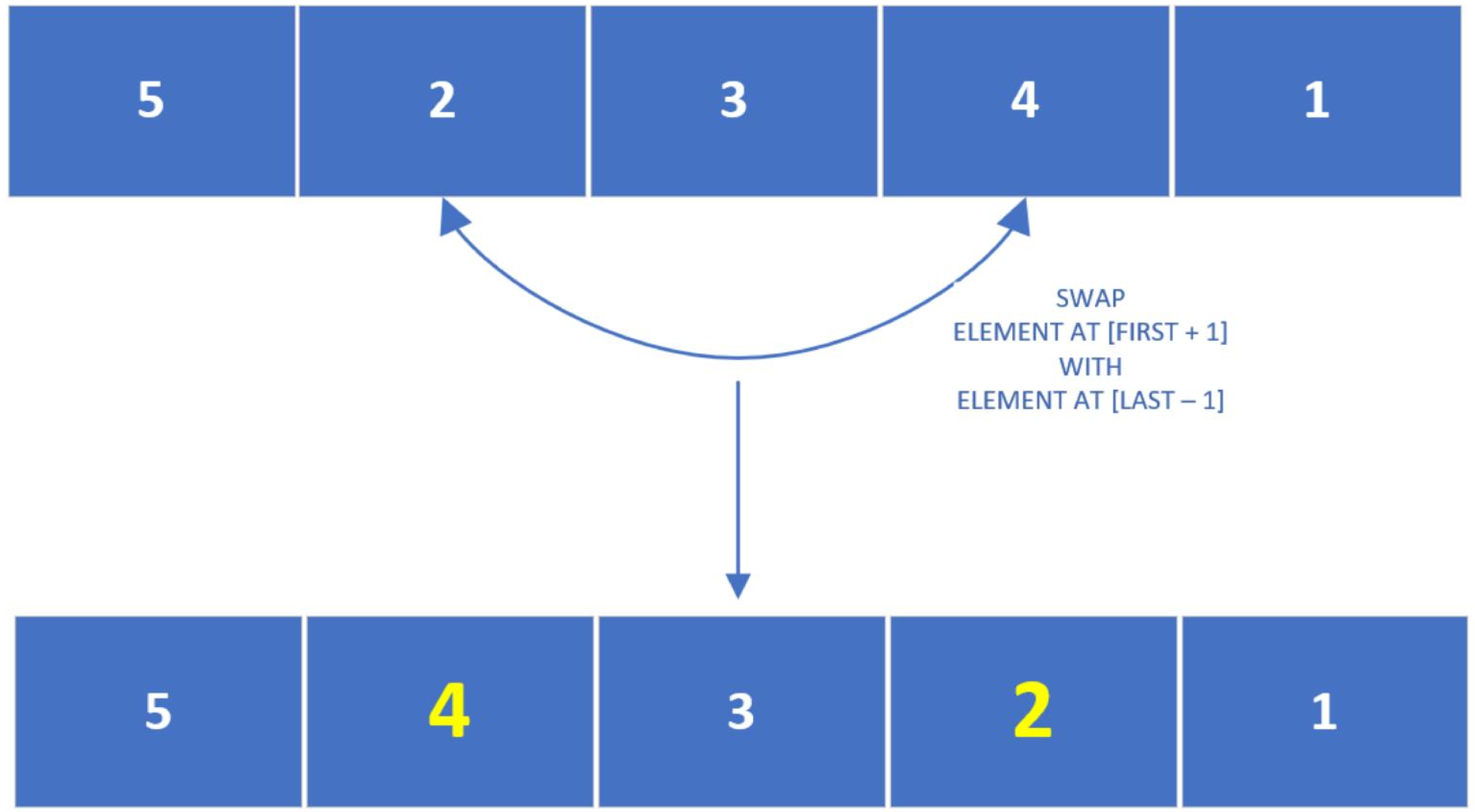
The Reverse Array Challenge

Then swap the elements at the next positions, which we could describe as:

Swapping the element at first plus one position with the element at the last minus one position.

At this point, for a 5 element array, you'd actually be done with the reverse process.

Notice the middle element never had to be swapped at all.



The Reverse Array Challenge

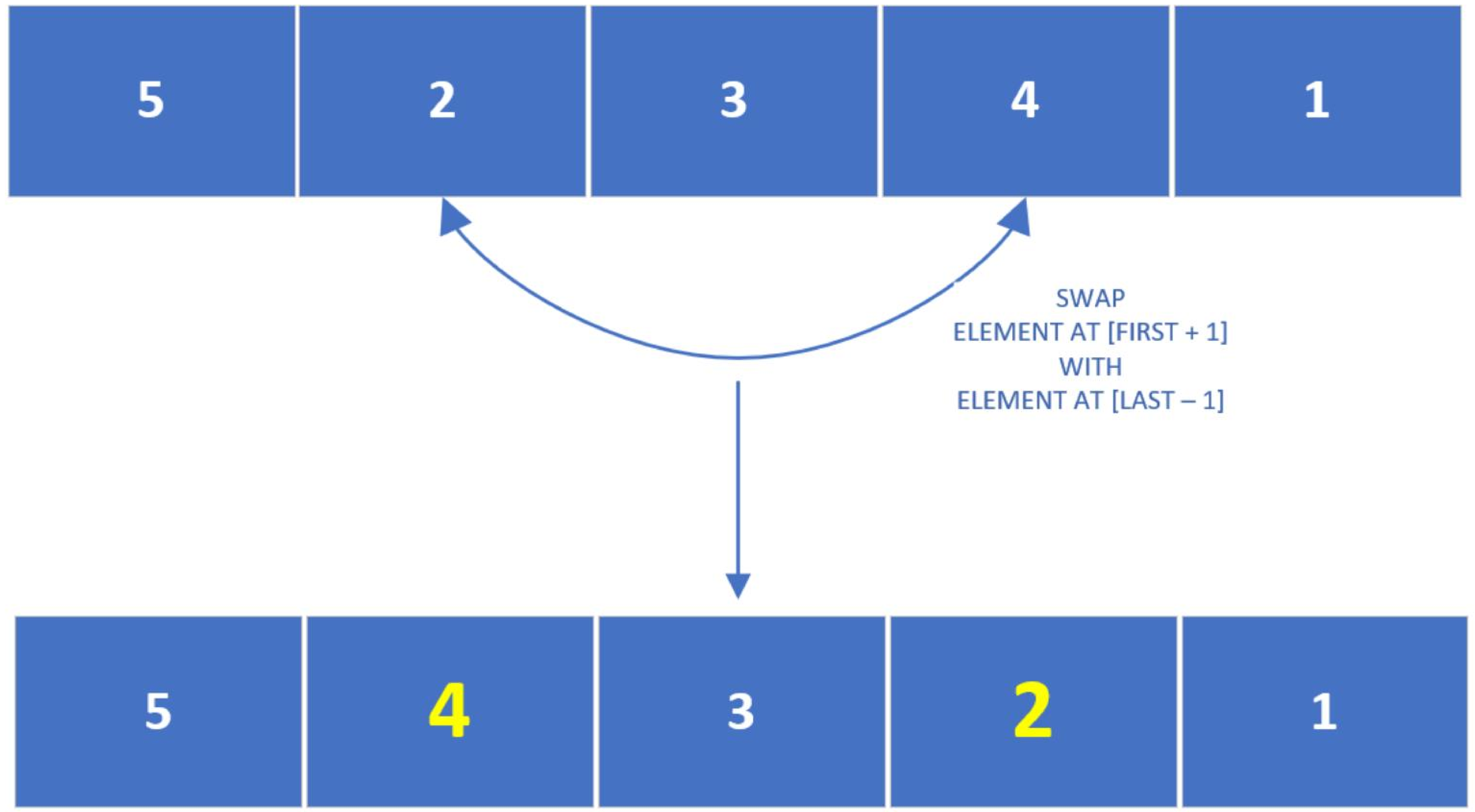
This process would take only two iterations to complete for five numbers.

It would take three iterations for seven numbers, etc.

We start with the outermost elements, swapping them, and work our way towards the center item.

If the number of elements is odd, we can leave the middle element unswapped.

We can think of the middle element as the pivot point.



Java's nested Arrays

An array element can actually itself be an array. It's known as a nested array, or an array assigned to an outer array's element.

This is how Java supports two and three dimensional arrays of varying dimensions.

Two-Dimensional Array

A two-dimensional array can be thought of as a table or matrix of values with rows and columns.

You can use an array initializer for this, which I'm showing on this slide.

Array Initializer formatted over multiple lines

`int[][] array = {`
`                    {1, 2, 3},`
`                    {11, 12, 13},`
`                    {21, 22, 23},`
`                    {31, 32, 33}`
`                    };`

Array Initializer declared on one line
`int[][] array = {{1, 2, 3}, {11, 12, 13}, {21, 22, 23}, {31, 32, 33}};`

Two-Dimensional Array

Notice the two sets of square brackets on the left side of the assignment in the declaration.

Using this type of declaration tells Java we want a two dimensional array of integers.

Array Initializer formatted over multiple lines

```
int[][] array = {
                    {1, 2, 3},
                    {11, 12, 13},
                    {21, 22, 23},
                    {31, 32, 33}
                    };
```

Array Initializer declared on one line

```
int[][] array = {{1, 2, 3}, {11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

Two-Dimensional Array

Here, I show the same declaration with array initializers that mean the same thing.

The first example just uses white space to make it more readable.

In this example, all the nested arrays have the same length.

Array Initializer formatted over multiple lines

```java
int[][] array = {
                    {1, 2, 3},
                    {11, 12, 13},
                    {21, 22, 23},
                    {31, 32, 33}
                    };
```

Array Initializer declared on one line

```java
int[][] array = {{1, 2, 3}, {11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

Two-dimensional Array

A 2-dimensional array doesn't have to be a uniform matrix though.

This means the nested arrays can be different sizes, as I show with this next initialization statement.

```
int[][] array = {
                                {1, 2},
                                {11, 12, 13},
                                {21, 22, 23, 24, 25}
                                };
```

Here, we have an array with 3 elements.

Each element is an array of integers (a nested array).

Each nested array is a different length.

If you find that confusing, don't worry. It should all make sense shortly.

Two-dimensional Array

You can initialize a two-dimensional array and define the size of the nested arrays, as shown here.

```
int[][] array = new int[3][3];
```

This statement says we have an array of 3 nested arrays, and each nested array will have three ints.

Two-dimensional Array

The result of this initialization is shown in the table on this slide.

Java knows we want a 3x3 matrix of ints and defaults the values of the nested arrays to zeros, as it would for any array.

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0

Two-dimensional Array

You can initialize a two-dimensional array without specifying the size of the nested arrays.

Here, we're specifying only the outer array size by specifying the length, only in the first set of square brackets.

We've left the second set of square brackets empty.

```java
int[][] array = new int[3][];
```

The result of this initialization is an array of 3 null elements.

We are limited to assigning integer arrays to these elements, but they can be any length.

Two-dimensional Array

There are a lot of ways to declare a two-dimensional array.

I'll cover the two most common ways here.

The most common, and in my opinion, the clearest way to declare a two-dimensional array is to stack the square brackets, as shown in the first example.

```
int[][] myDoubleArray;
int[] myDoubleArray[];
```

You can also split up the brackets as shown in the second example, and you'll likely come across this in Java code out in the wild.

Accessing elements in multi-dimensional arrays

When we access a one dimensional array element, we do it with square brackets and an index value.

So this code sets the first element in the array to 50:

```
array[0] = 50;
```

To access elements in a two-dimensional array, we use two indices, so this code sets the first element in the first array to 50.

```
secondArray[0][0] = 50;
```

This next code sets the second element, in the second array to 10.

```
secondArray[1][1] = 10;
```

Accessing elements in multi-dimensional arrays

The code on this slide is similar to the code we have used in IntelliJ, using nested traditional for loops.

In this case, we're not using any local variables, but accessing array elements and variables directly.

```
for (int i = 0; i < array2.length; i++) {
    for (int j = 0; j < array2[i].length; j++) {      // while i = 0, j will loop from 0 to 3
        System.out.print(array2[i][j] + " ");
    }
}
```

Accessing elements in multi-dimensional arrays

This table shows the indices, which are used to access the elements in the two-dimensional array in our code sample.

When we loop through the outer loop, we're accessing each row of elements.

I've highlighted the first row, which would be the elements accessed, when i = 0 for the outer for loop.

When we loop through the inner loop, we're accessing each cell in the array.

A cell in this matrix can be any type.

In our code, each is an integer value, and we know they've all been initialized to zero.

	j = 0	j = 1	j = 2	j = 3
i = 0	[0][0]	[0][1]	[0][2]	[0][3]
i = 1	[1][0]	[1][1]	[1][2]	[1][3]
i = 2	[2][0]	[2][1]	[2][2]	[2][3]

Two Dimensional Array

When we declare multi-dimensional arrays, the declared type can itself be an array, and this is how Java supports two-dimensional arrays:

```
int[][] myArray = new int[3][];      // Declares and instantiates an array of 3 integer arrays,
//      whose sizes are not specified

Dog[][] myDogs = new Dog[3][];       // Declares and instantiates an array of 3 arrays,
//      which will have Dog elements, again, the sizes of the inner arrays aren't specified
```

Type and length of array	Possible Element Values (each element is an array and can be any length)
int[3][]	[5, 7, 9, 10] [3, 6] [11, 21, 31]
Dog[3][]	[pug, rottweiler] [germanShephard, poodle, cavapoo] [beagle, boxer, bulldog, yorkie]

Multi Dimensional Array

We can take that even further, the outer array can have references to any kind of array itself.

In this example, we have an outer array with three elements.

```
Object[] multiArray = new Object[3];
multiArray[0] = new Dog[3];
multiArray[1] = new Dog[3][];
multiArray[2] = new Dog[3][][];
```

The first element is itself a single-dimensional array.

The second element is a two-dimensional array.

And lastly, the third element is a three-dimensional array.