📚 What is an Array?

An **array** is a non-primitive data structure of fixed size that stores multiple values of the same data type in a single variable. Each value in the array is accessed using an **index**, which starts from 0.

❓ Why Are Arrays Needed?

Arrays are essential for efficiently storing and managing large amounts of related data without the need to create separate variables for each item.

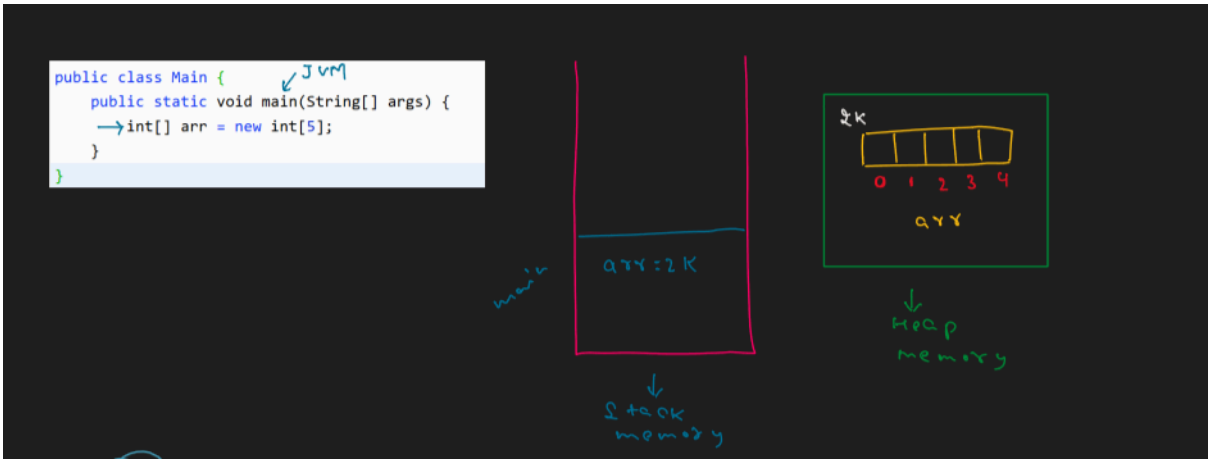
Example Analogy: Imagine an Indian uncle running a coaching class with 50 students. Without arrays, he'd have to create 50 separate variables like `marks1`, `marks2`, ..., `marks50`. That's like writing each student's name on a separate page—tedious and inefficient. With an array, he can simply use `marks[0]`, `marks[1]`, ..., `marks[49]`. Less work, more logic, and fewer pages wasted!

🛏️ Real-Life Example

Think of **dorm beds in a hostel**. Each bed is labeled with a number (like an index), and all beds are arranged in a row (like an array). You can easily find or assign a bed using its number.

🧠 Memory Perspective

- Arrays are **non-primitive** data types.
- When declared, the **reference** to the array is stored in the **stack**, but the actual array elements are stored in the **heap memory**.
- This allows arrays to be dynamically allocated and persist beyond the scope of a single method or function.



🧠 Understanding Arrays: Declaration, Memory Storage, Indexing, and Access

📌 1. Declaration of Arrays

Arrays are declared by specifying the data type followed by square brackets and the array name.

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

This creates an array of integers with 5 elements.

🗂️ 2. Memory Storage: Call Stack vs Heap

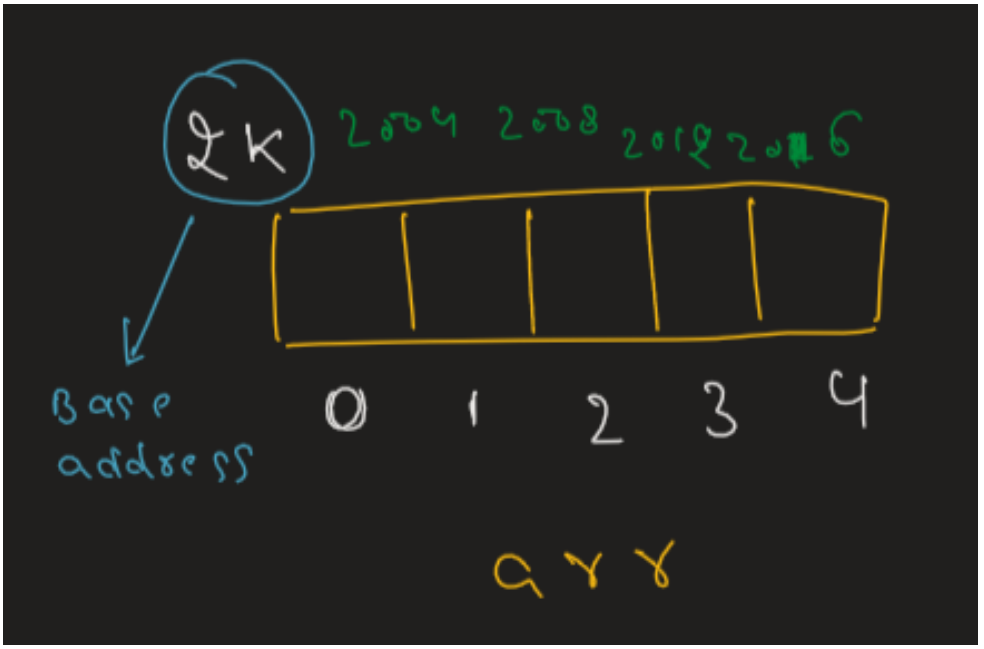
- **Call Stack**: Stores references to arrays when declared inside methods (local variables).
- **Heap Memory**: The actual array elements are stored in the heap. This allows dynamic memory allocation and persistence beyond the method scope.

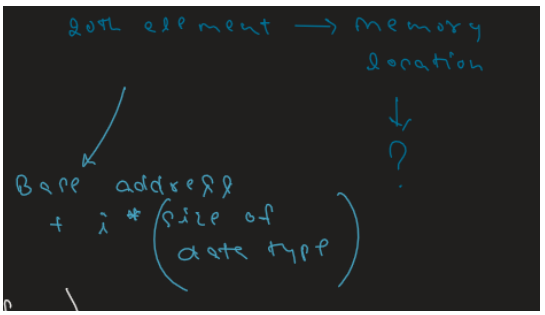
🧭 3. Indexing in Arrays

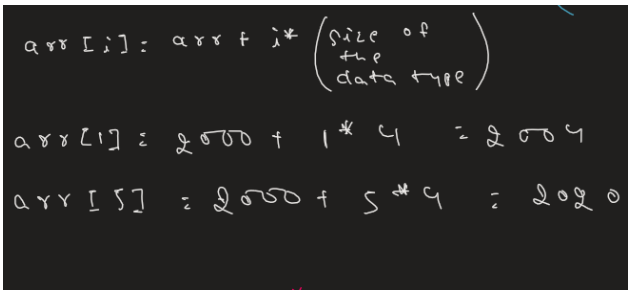
- Arrays use **zero-based indexing**, meaning the first element is at index 0.
- For an array of size `n`, valid indices range from `0` to `n-1`.

🧬 4. How Memory Addressing Works

- The array variable holds a reference (or pointer) to the starting memory address of the array in the heap.
- Each element is stored in contiguous memory locations.
- The address of an element at index `i` is calculated as:
  base_address + (i * size_of_element)







🔍 5. Accessing and Modifying Elements

Getting an element: Use the index to retrieve the value.

```
int x = numbers[2]; // Retrieves the third element
```

Setting an element: Assign a value using the index.

```
numbers[2] = 10; // Sets the third element to 10
```

```
public class Main {

	public static void main(String[] args) {
		int[] arr = new int[5];
		int arr1[] = new int[5]; //C Type Declaration
		System.out.println(arr); //[I@12a3a380
		System.out.println(arr[0]);
		System.out.println(arr[1]);
		int[] other = arr;
		System.out.println(other);
		System.out.println(other.length);
	}

}
```

🔍 What `[I@12a3a380` Means in Java

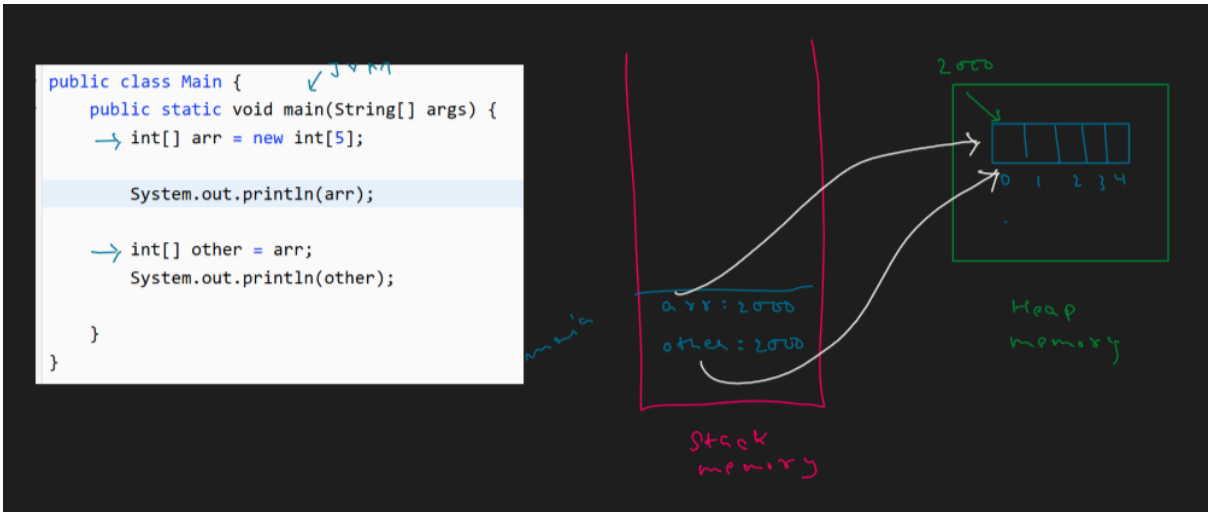
- `[` indicates that it's a one-dimensional array.
- `I` stand for the data type `int`.
- @ Separator between type and memory address.
- 12a3a380 is the hexadecimal representation of the array object's hash code, not the actual memory address.

🧠 Reference Behavior in Arrays

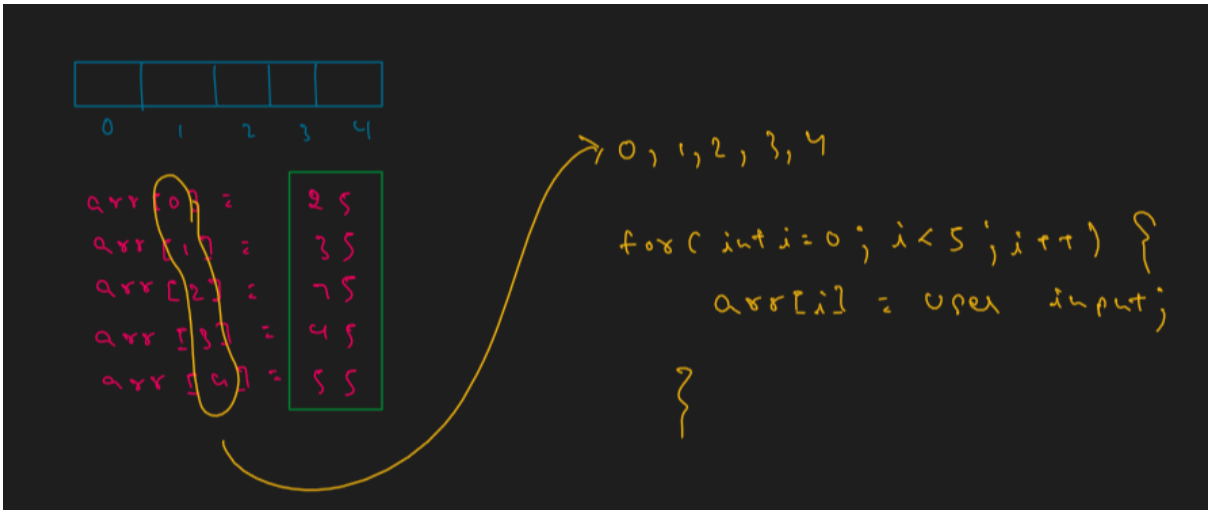
When you assign one array variable to another in Java, both variables point to the **same memory location**. This means any change made through one variable will be reflected when accessed through the other.

This happens because arrays are **reference types** in Java. The variables `arr` and `other` are like two flatmates sharing the same flat. If one of them paints the living room blue, the other will walk in and see the same blue walls. 🎨

So, modifying the array through one reference affects the shared object, and the change is visible to all references pointing to it.



🎯 Taking User Input in an Array (Java Example)



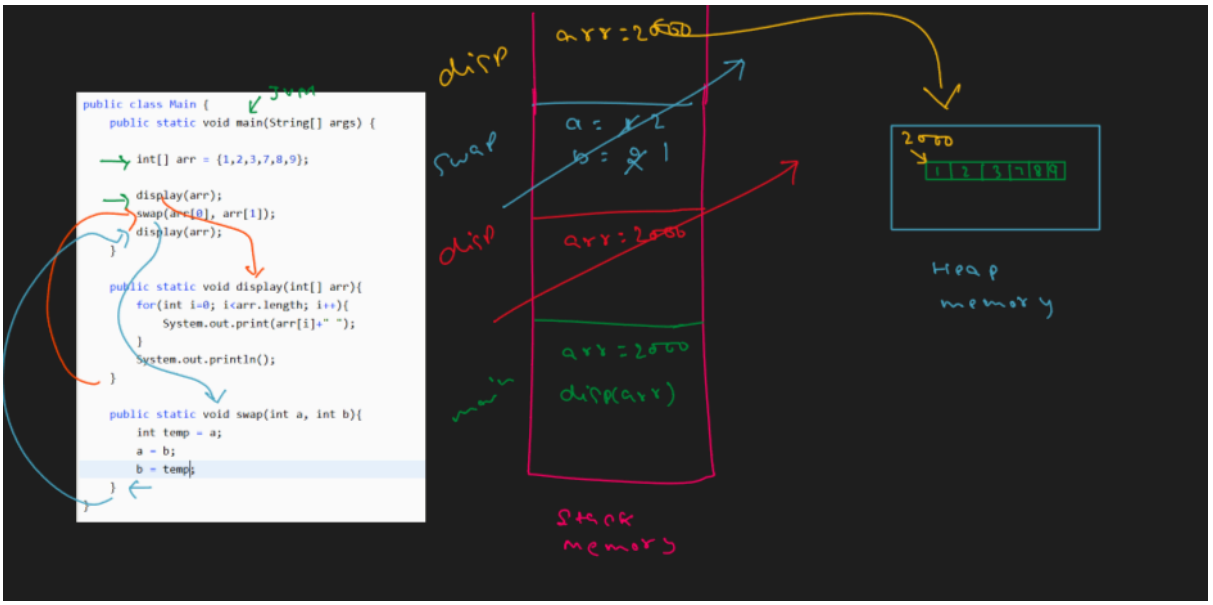
```java
import java.util.Scanner;

public class Main {

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		int n = sc.nextInt();
		int[] arr = new int[n];
		for (int i = 0; i < arr.length; i++) {
			arr[i] = sc.nextInt();
		}
		display(arr);

	}

	public static void display(int[] arr) {
		for (int i = 0; i < arr.length; i++) {
			System.out.print(arr[i] + " ");
		}
		System.out.println();
	}

}
```

🎯 Ways to declare

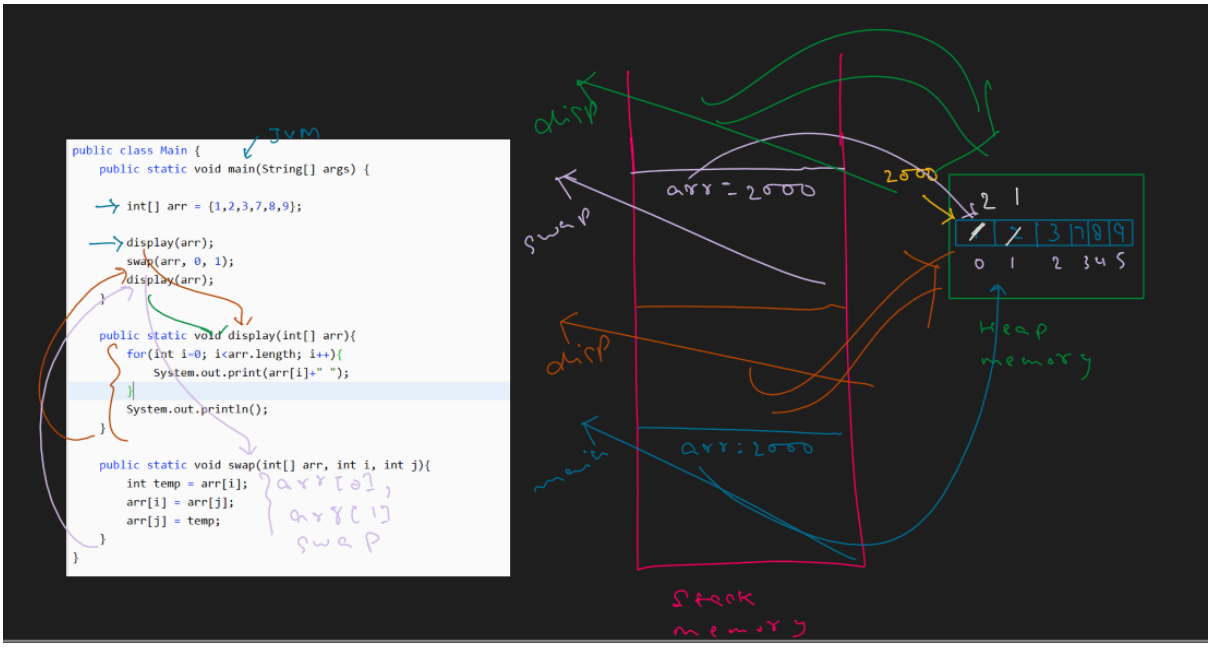
```java
public class Main {
    public static void main(String[] args) {
        int[] arr1 = { 10, 4, 5, 67, 8 };
        int arr2[] = { 10, 4, 5, 67, 8 };
            int[] arr3 = new int []{ 10, 4, 5, 67, 8 };
        int arr4[] = new int []{ 10, 4, 5, 67, 8 };
        int[] arr5 = new int[5];
        int arr6[] = new int[5];
    }
}
```

🧠 Call by value or call by referrence

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

        int[] arr = {1, 2, 3, 7, 8, 9};

        display(arr);
        swap(arr[0], arr[1]);
        display(arr);
    }

    public static void display(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    public static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }
}
```

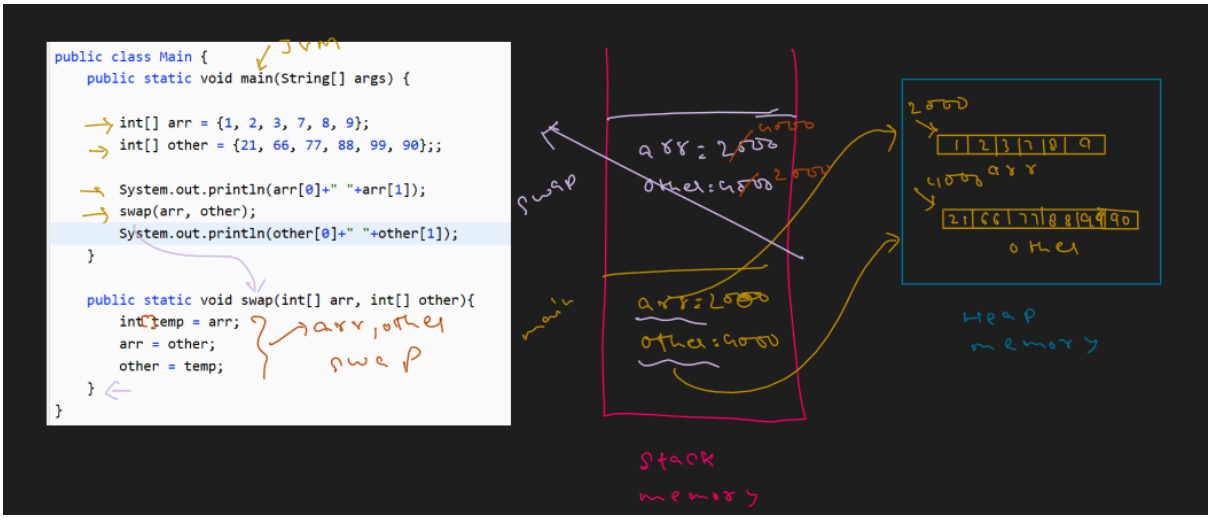


```java
public class Main {
    public static void main(String[] args) {

        int[] arr = {1, 2, 3, 7, 8, 9};

        display(arr);
        swap(arr, 0, 1);
        display(arr);
    }

    public static void display(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```



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

        int[] arr = {1, 2, 3, 7, 8, 9};
        int[] other = {21, 66, 77, 88, 99, 90};

        System.out.println(arr[0] + " " + arr[1]);
        swap(arr, other);
        System.out.println(other[0] + " " + other[1]);
    }

    public static void swap(int[] arr, int[] other) {
        int[] temp = arr;
        arr = other;
        other = temp;
    }
}
```

Java is always pass-by-value (call-by-value).
In the given example, the address of the local variable is being swapped, which will not be reflected in the original method. We can only modify the value at the index, not the reference itself.



🎯 Linear Search

Suppose there are 10 places in my house where I usually keep my phone: the drawer, the second cupboard, on the fridge, near the photo frames in the kitchen, and so on.

Now, I want to install Instagram and watch some reels, so I start looking for my phone. I go to the first place—the drawer—and check if it's there. If I find it, great! I stop searching and start scrolling reels.

If it's not in the drawer, I go to the second place, then the third, then the fourth, and continue like that. As soon as I find my phone at any of these places, I stop right there. That's it. Search ends.

If I've checked all 10 places and still haven't found it, that means my phone is missing—which thankfully never happens in my house!

This is exactly how **linear search** works:

- You start at the first element.
- You check each element one by one.
- The moment you find what you're looking for, you stop.

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

	int[] arr = {1, 2, 3, 6, 7, 8, 9};
	int item = 8;
	int index = -1;

	for(int i=0; i<arr.length; i++){
		if(arr[i] == item){
			index = i;
			break;
		}
	}
	System.out.println(index);
	}
}
```

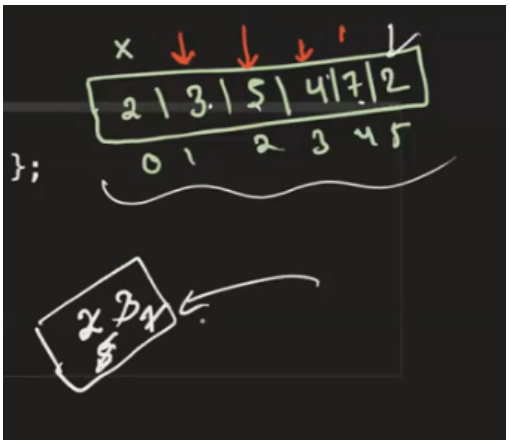
```java
public class Main {
	public static void main(String[] args) {

	Scanner sc = new Scanner(System.in);

	int n = sc.nextInt();
	int[] arr = new int[n];

	for(int i=0; i<n; i++){
		arr[i] = sc.nextInt();
	}

	int item = sc.nextInt();
	int index = -1;

	for(int i=0; i<arr.length; i++){
		if(arr[i] == item){
			index = i;
			break;
		}
	}
	System.out.println(index);

	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int[] arr = { 10, 3, 4, 6, 7, 9 };
		int item = 6;
		System.out.println(linearSearch(arr, item));
	}

	public static int linearSearch(int[] arr, int item) {
		for (int i = 0; i < arr.length; i++) {
			if (arr[i] == item) {
				// System.out.println(i);
				return i;
			}
		}
		// System.out.println(-1);
		return -1;
	}
}
```

🎯 Find Maximun in Array

To explain how to find the maximum value in an array, let's use the example of a piggy bank. Suppose you need to find the largest coin in the piggy bank. Here's how you'd do it:

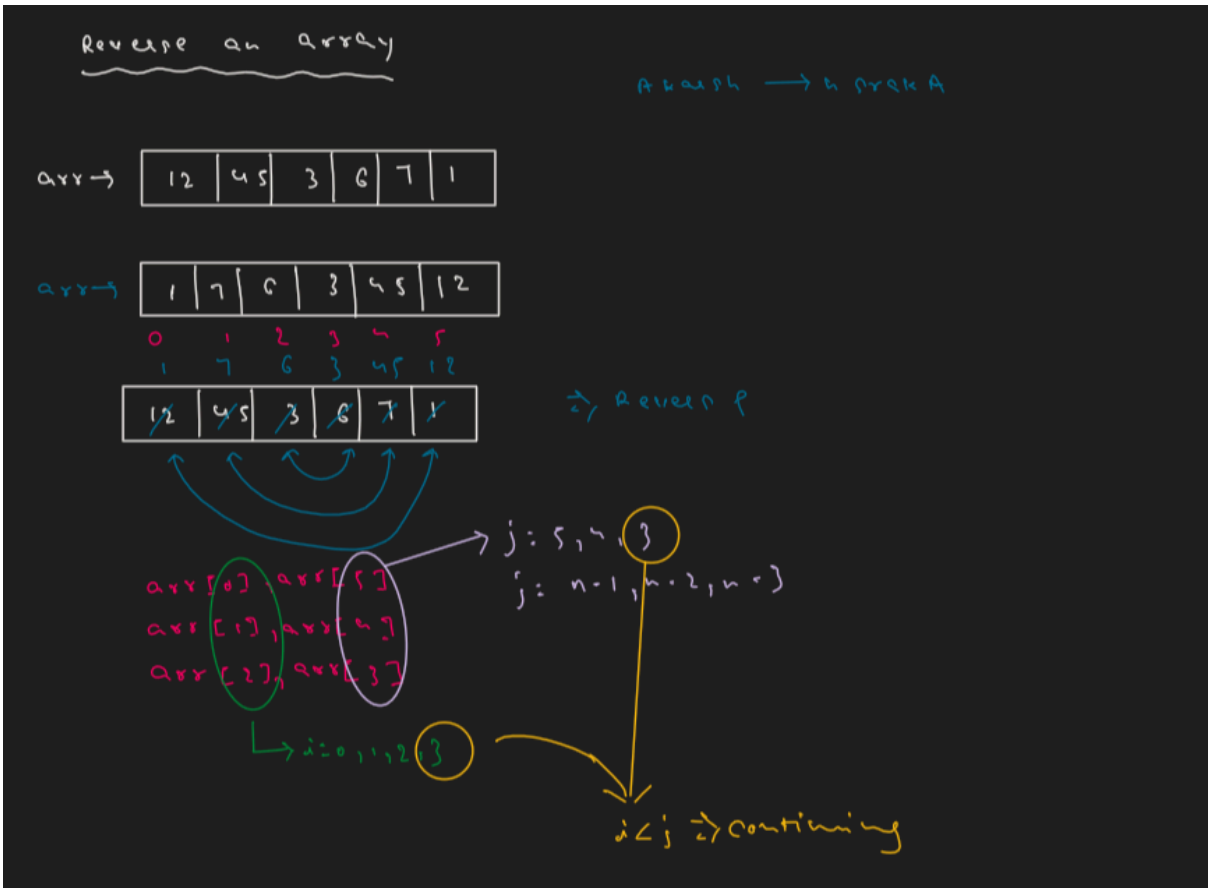
You start by taking the first coin out and writing its value on a piece of paper. Then, you take out the second coin. If it's greater than the value written on the paper, you update the paper with this new value. You continue this process for each coin—if a coin is greater than the current value on the paper, you replace it; otherwise, you leave it as is.

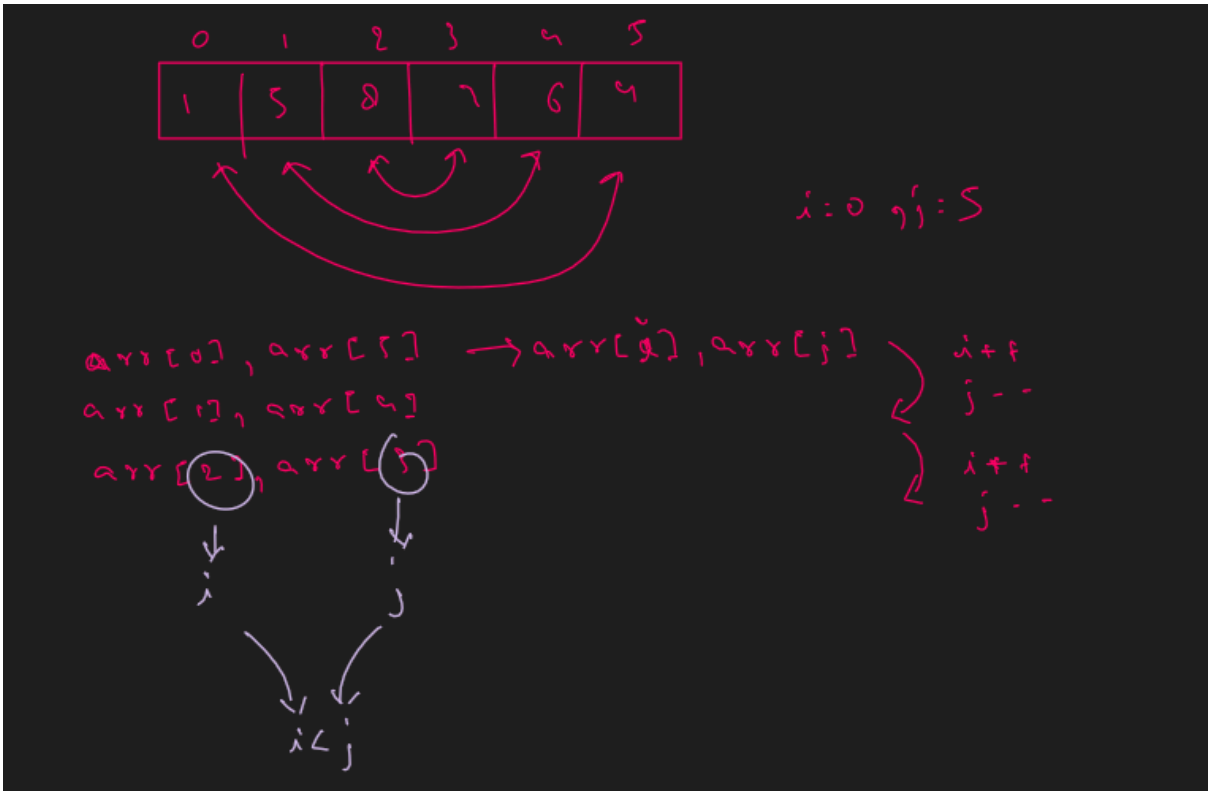



```java
public class Main {
	public static void main(String[] args) {
		int[] arr = { 2, 3, 5, 1, 4, 11, 40, 51, 1, 15 };
		System.out.println(maxValue1(arr));
	}

	public static int maxValue3(int[] arr) {
		int max = Integer.MIN_VALUE;// -2^31
		for (int i = 0; i < arr.length; i++) {
			max=Math.max(max, arr[i]);
		}
		return max;
	}


	public static int maxValue2(int[] arr) {
		int max = Integer.MIN_VALUE;// -2^31
		for (int i = 0; i < arr.length; i++) {
			if (arr[i] > max) {
				max = arr[i];
			}
		}
		return max;
	}
```

```
	public static int maxValue1(int[] arr) {
		int max = arr[0];
		for (int i = 1; i < arr.length; i++) {
			if (arr[i] > max) {
				max = arr[i];
			}
		}
		return max;

	}
}
```

🎯 Reverse an Array

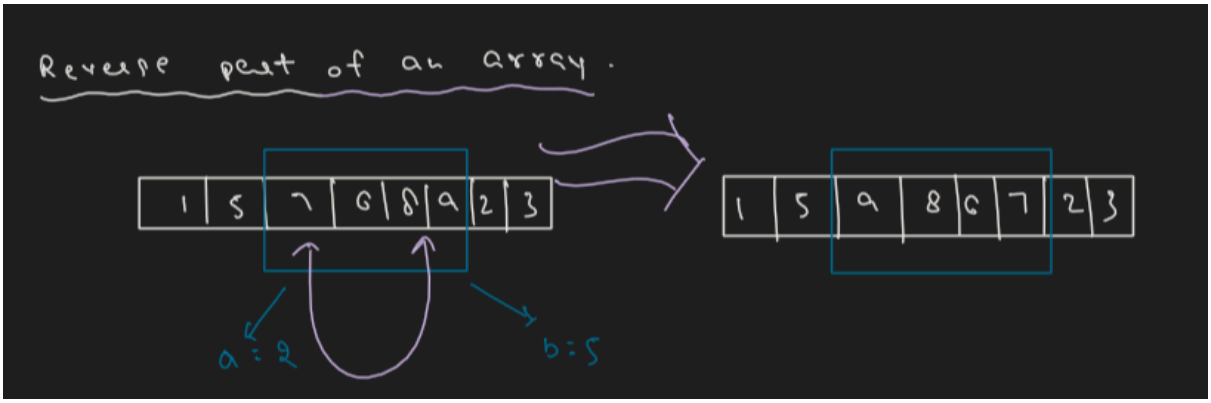




```java
public class Main {

	public static void main(String[] args) {
		int[] arr = { 12, 45, 3, 6, 7, 9, 2 };
		reverseArray(arr);

		// reversePrint(arr);

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

	public static void reverseArray(int[] arr) {
		int i = 0;
		int j = arr.length-1;
		while (i < j) {
			int temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
			i++;
			j--;
		}
	}
```

```
	public static void reversePrint(int[] arr) {
		for (int i = arr.length - 1; i >= 0; i--) {
			System.out.print(arr[i] + " ");
		}
		System.out.println();
	}
}
```

🎯 Reverse part of an Array




```
public class Main {

	public static void main(String[] args) {
		int[] arr = { 3, 5, 1, 7, 8, 6, 9, 11, 15, 17, 18, 16, 23,
27 };

		reverseArrayPart(arr, 3, 11);
		for (int i = 0; i < arr.length; i++) {
			System.out.print(arr[i] + " ");
		}
	}

	public static void reverseArrayPart(int[] arr, int i, int j) {
		while (i < j) {
			int temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
			i++;
			j--;
		}
	}

}
```