JAVA ARRAY CONCEPTS :

**Declaration and Initialization:**

Declare an array: int[] numbers;

Initialize an array: numbers = new int[5];

Example:

int[] numbers;

numbers = new int[5];

**Initialization with Values:**

Declare and initialize an array with values.

Example:

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

**Accessing Array Elements:**

Access elements using index.

Example:

System.out.println(numbers[2]); // Output: 3

**Array Length:**

Get the length of the array.

Example:

int length = numbers.length;

System.out.println("Array Length: " + length); // Output: 5

**Iteration through an Array:**

Use a loop to iterate through array elements.

Example:

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

System.out.print(numbers[i] + " ");

}

// Output: 1 2 3 4 5

**Multidimensional Arrays:**

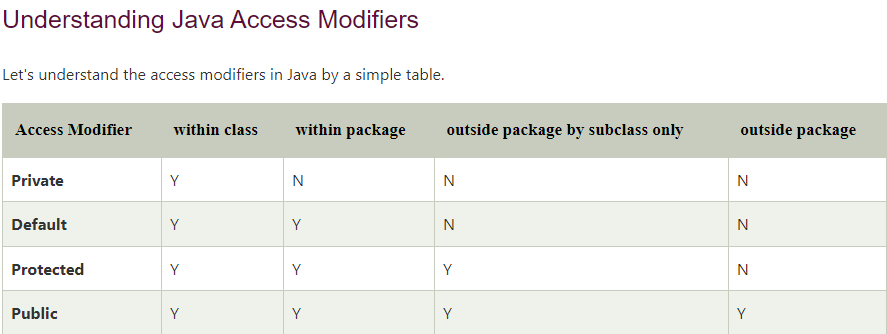
Declare and initialize a 2D array.

Example:

int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

Queries :

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.



* **Access Modifiers** - controls the access level
* **Non-Access Modifiers** - do not control access level, but provides other functionality

**Non-Access Modifiers**

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

String[] cars;

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

int[] myNum = {10, 20, 30, 40};

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo

public class Main {

public static void main(String[] args) {

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

}

## Change an Array Element

To change the value of a specific element, refer to the index number:

}

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

System.out.println(cars[0]);

// Now outputs Opel instead of Volvo

## Array Length

To find out how many elements an array has, use the length property:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

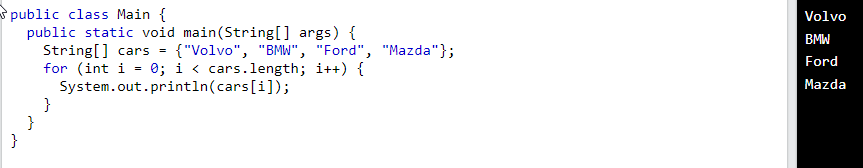
System.out.println(cars.length);

// Outputs 4

## Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:



**for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of**i**.

If you compare the for loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.

public class Main {

public static void main(String[] args) {

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

**for (String i : cars) {**

**System.out.println(i);**

**}**

}

}

## Multidimensional Arrays

A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

To create a two-dimensional array, add each array within its own set of **curly braces**:

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**myNumbers** is now an array with two arrays as its elements.

**Arrays in Methods:**

Pass arrays to methods.

Example:

public static void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

}

**Call the method:**

printArray(numbers); // Output: 1 2 3 4 5

Array Copy:

Copy elements from one array to another.

Example:

int[] copiedArray = Arrays.copyOf(numbers, numbers.length);

**ArrayList in Java with simple examples suitable for beginners in software testing:**

Import ArrayList class:

import java.util.ArrayList;

Declare an ArrayList:

ArrayList<String> fruitsList = new ArrayList<>();

Add elements to ArrayList:

fruitsList.add("Apple");

fruitsList.add("Banana");

fruitsList.add("Orange");

Retrieve elements from ArrayList:

System.out.println("Fruits List: " + fruitsList);

Access elements using index:

String firstFruit = fruitsList.get(0);

System.out.println("First Fruit: " + firstFruit);

Modify elements in ArrayList:

fruitsList.set(1, "Grapes");

System.out.println("Updated List: " + fruitsList);

Remove elements from ArrayList:

fruitsList.remove("Orange");

System.out.println("Updated List after removal: " + fruitsList);

Check if an element exists:

if (fruitsList.contains("Banana")) {

System.out.println("Banana is in the list.");

} else {

System.out.println("Banana is not in the list.");

}

Iterate through ArrayList:

for (String fruit : fruitsList) {

System.out.println(fruit);

}

ArrayList size:

int size = fruitsList.size();

System.out.println("Size of the list: " + size);

Check if ArrayList is empty:

if (fruitsList.isEmpty()) {

System.out.println("The list is empty.");

} else {

System.out.println("The list is not empty.");

}

These examples cover the basic operations you might perform with ArrayList in Java. As a software test engineer, you might use these concepts for testing scenarios involving data manipulation and validation.

**Java Interface:**

An interface in Java is a collection of abstract methods. It provides a way to achieve abstraction and multiple inheritance. Here's a simple example:

// Define an interface

interface Shape {

void draw(); // Abstract method

}

// Implement the interface in a class

class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a circle");

}

}

class Square implements Shape {

@Override

public void draw() {

System.out.println("Drawing a square");

}

}

// Main class to test interfaces

public class InterfaceExample {

public static void main(String[] args) {

Shape circle = new Circle();

Shape square = new Square();

circle.draw(); // Output: Drawing a circle

square.draw(); // Output: Drawing a square

}

}

**Java Collections:**

Java Collections Framework provides a set of classes and interfaces to handle and manipulate groups of objects. Here's an example using ArrayList:

import java.util.ArrayList;

public class CollectionExample {

public static void main(String[] args) {

// Create an ArrayList of Strings

ArrayList<String> fruits = new ArrayList<>();

// Add elements to the ArrayList

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

// Access elements using for-each loop

System.out.println("Fruits in the list:");

for (String fruit : fruits) {

System.out.println(fruit);

}

// Check if an element is present

if (fruits.contains("Banana")) {

System.out.println("Banana is in the list");

}

// Remove an element

fruits.remove("Apple");

// Display the updated list

System.out.println("Updated list after removing Apple:");

for (String fruit : fruits) {

System.out.println(fruit);

}

}

}

**In Java, an interface is a collection of abstract methods (methods without a body) and constants. It is used to achieve abstraction, multiple inheritance, and to define a contract that classes must adhere to. Here's a simple example:**

// Define an interface named Animal

interface Animal {

// Abstract method (method without a body)

void makeSound();

// Constant (implicitly public, static, and final)

String TYPE = "Unknown";

}

// Implement the interface in a class

class Dog implements Animal {

@Override

public void makeSound() {

System.out.println("Woof! Woof!");

}

}

class Cat implements Animal {

@Override

public void makeSound() {

System.out.println("Meow!");

}

}

// Main class to test the interface

public class InterfaceExample {

public static void main(String[] args) {

Animal dog = new Dog();

Animal cat = new Cat();

// Call the method defined in the interface

dog.makeSound(); // Output: Woof! Woof!

cat.makeSound(); // Output: Meow!

// Access the constant defined in the interface

System.out.println("Type of dog: " + Dog.TYPE); // Output: Type of dog: Unknown

System.out.println("Type of cat: " + Cat.TYPE); // Output: Type of cat: Unknown

}

}

In this example:

Animal is an interface with one abstract method makeSound() and a constant TYPE.

Dog and Cat are classes that implement the Animal interface. They provide concrete implementations for the makeSound() method.

In the main method, we create instances of Dog and Cat and call the makeSound() method. We also access the constant TYPE defined in the interface.

Interfaces are commonly used in Java to achieve abstraction, allowing you to define a common set of methods that multiple classes can implement in their own way. This promotes code reusability and flexibility in your program design.

**Exception handling in Java is a mechanism to handle runtime errors gracefully, preventing the program from terminating abruptly. It involves the use of try, catch, and finally blocks. Here's a simple example for beginners:**

public class ExceptionHandlingExample {

public static void main(String[] args) {

try {

// Code that may cause an exception

int result = divide(10, 0);

System.out.println("Result: " + result);

// This line won't be executed if an exception occurs

} catch (ArithmeticException e) {

// Catch block to handle the exception

System.out.println("Error: Division by zero");

} finally {

// Finally block, always executed whether an exception occurs or not

System.out.println("Inside the finally block");

}

// Code continues to execute after exception handling

System.out.println("Program continues...");

}

// Method that may throw an exception

static int divide(int a, int b) {

return a / b;

}

}

In this example:

The divide method attempts to perform a division operation that may result in an ArithmeticException if the divisor is zero.

The try block contains the code that might throw an exception.

The catch block catches and handles the exception. In this case, it prints an error message.

The finally block contains code that is always executed, whether an exception occurs or not. It is often used for cleanup operations.

The program continues to execute after the exception handling, demonstrating that the application doesn't crash due to the exception.

This is a basic illustration, and in real-world scenarios, you would handle exceptions more precisely, possibly logging the details of the exception or taking appropriate corrective actions.