Class topics

1. Java Methods
    a. Introduction to Methods
    b. Static Methods vs Instance Methods
    c. Access Modifiers

2. Java Arrays
    a. Introduction to Arrays
    b. Arrays Class
    c. Final Arrays
    d. Java Array Programs

Introduction to methods

Java Methods are blocks of code that perform a specific task. A method allows us to reuse code, improving both efficiency and organization. All methods in Java must belong to a class. Methods are similar to functions and expose the behavior of objects.

Example: Java program to demonstrate how to create and use a method.

```
// Creating a method that prints a message
public class MethodExample{
    // Method to print message
    public void printMessage() {
        System.out.println("Hello, Pathshala!");
    }

    public static void main(String[] args) {
        // Create an instance of the Method class
        MethodExample obj = new MethodExample();
        // Calling the method
        obj.printMessage();
    }
}

Output
Hello, Pathshala!
```

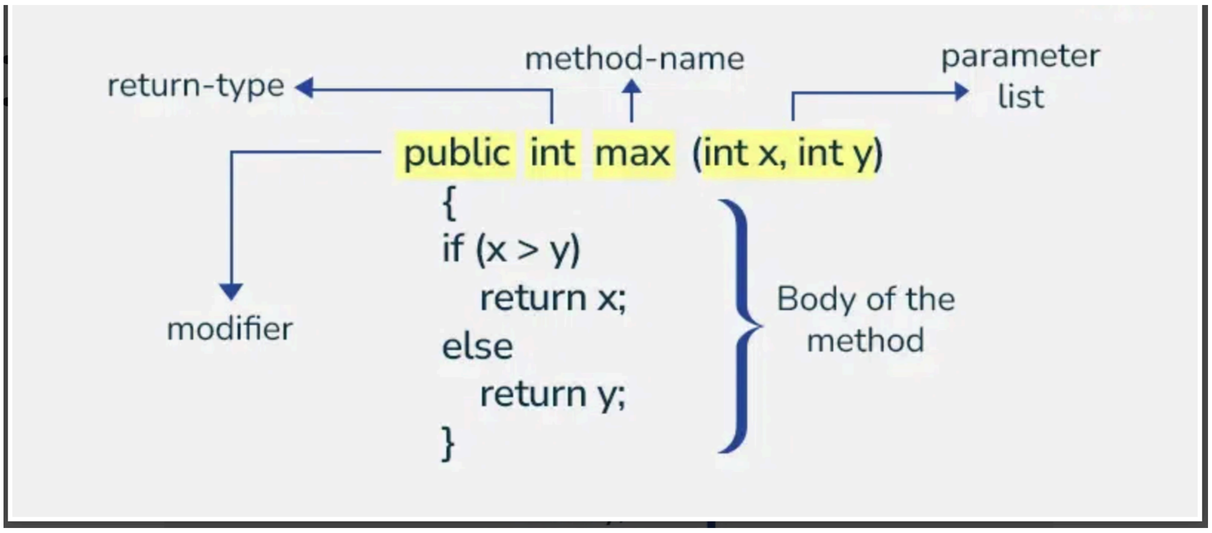
Explanation: In this example, printMessage() is a simple method that prints a message. It has no parameters and does not return anything. Here, first we create a method which prints Hello, Pathshala!.

Syntax of a Method

```
<access_modifier> <return_type> <method_name>( list_of_parameters)
{
    //body
}
```

Key Components of a Method Declaration

- **Modifier:** It specifies the method's access level (e.g., public, private, protected, or default).
- **Return Type:** The type of value returned, or void if no value is returned.
- **Method Name:** It follows Java naming conventions; it should start with a lowercase verb and use camel case for multiple words.
- **Parameters:** A list of input values (optional). Empty parentheses are used if no parameters are needed.
- **Exception List:** The exceptions the method might throw (optional).
- **Method Body:** It contains the logic to be executed (optional in the case of abstract methods).



```
public int max (int x, int y)
{
if (x > y)
   return x;
else
   return y;
}
```

Types of Methods in Java

1. Predefined Method

Predefined methods are the method that is already defined in the Java class libraries. It is also known as the standard library method or built-in method. for example random() method which is present in the Math class and we can call it using the ClassName.methodName() as shown in the below example.

Example:

```
Math.random()    // returns random value
Math.PI()     // return pi value
```

2. User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

```
Example:
sayHello         // user define method created above in the article
Greet()
```

```
setName()
```

Different Ways to Create Java Method

There are two ways to create a method in Java:

1. Instance Method: Access the instance data using the object name. Declared inside a class.

Example:

```
// Instance Method
void method_name() {
    // instance method body
}
```

2. Static Method: Access the static data using class name. Declared inside class with static keyword.

Example:

```
// Static Method

static void method_name() {

    // static method body

}
```

Method Signature

It consists of the method name and a parameter list.

- Number of parameters
- Type of the parameters
- Order of the parameters

Note: The return type and exceptions are not considered as part of it.

Method Signature of the above function:

```
max(int x, int y) Number of parameters is 2, Type of parameter is int.
```

Naming a Method

In Java language method name is typically a single word that should be a verb in lowercase or a multi-word, that begins with a verb in lowercase followed by an adjective, noun. After the first word, the first letter of each word should be capitalized.

Rules to Name a Method:

- While defining a method, remember that the method name must be a verb and start with a lowercase letter.
- If the method name has more than two words, the first name must be a verb followed by an adjective or noun.
- In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example, findSum, computeMax, setX, and getX.

Generally, a method has a unique name within the class in which it is defined but sometimes a method might have the same name as other method names within the same class as method overloading is allowed in Java.

Method Calling

Method calling allows us to reuse code and organize our program effectively. The method needs to be called to use its functionality. There can be three situations when a method is called:

A method returns to the code that invoked it when:

- It completes all the statements in the method.
- It reaches a return statement.
- Throws an exception.

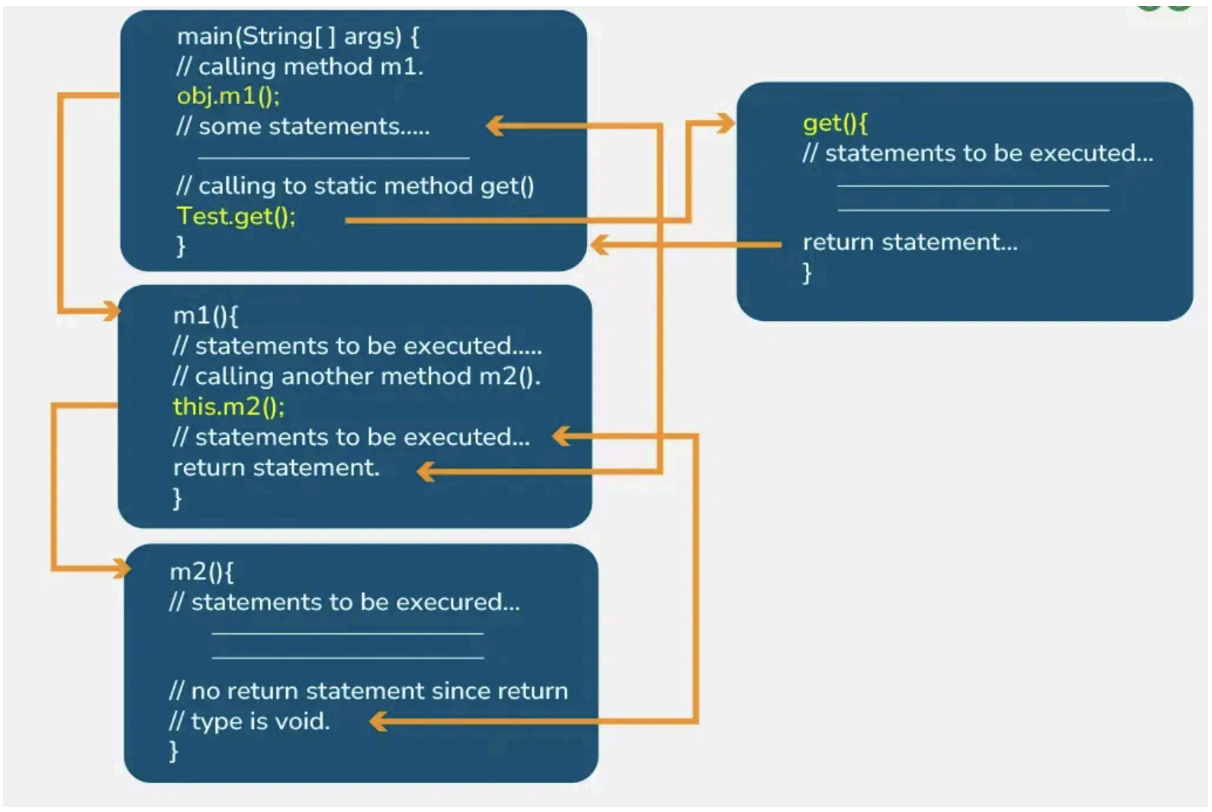
Example 1: Method calling using object of a class.

```
// Java program demonstrates how to call a method
class Add {
    int s = 0;
    public int addTwoInt(int a, int b) {
        s = a + b;
        return s;
    }
}

Class MethodExample {
    public static void main(String[] args) {
        Add a = new Add();
        int res = a.addTwoInt(1, 2);
        System.out.println("Sum: " + res);
    }
}

Output
Sum: 3
```

Example 2: Calling Methods in Different Ways

```
// Java Program to Illustrate Method Calling
import java.io.*;

class Test {
    public static int i = 0;

    Test() {
        i++;
    }

    public static int get() {
        return i;
    }

    public int m1() {
        System.out.println("Inside the method m1");
        this.m2();
        return 1;
    }

    public void m2() {
        System.out.println("In method m2");
    }
}

class MethodExample {
    public static void main(String[] args) {
        Test obj = new Test();

        int i = obj.m1();
        System.out.println("Control returned after m1: " + i);

        int o = Test.get();
        System.out.println("No of instances created: " + o);
    }
}

Output
Inside the method m1
In method m2
Control returned after m1: 1
No of instances created: 1
```

The control flow of the above program is as follows:



```
main(String[ ] args) {
// calling method m1.
obj.m1();
// some statements.....
______________________
// calling to static method get()
Test.get();
}
```

```
get(){
// statements to be executed...
______________________
______________________
return statement...
}
```

```
m1(){
// statements to be executed.....
// calling another method m2().
this.m2();
// statements to be executed...
return statement.
}
```

```
m2(){
// statements to be execured...
______________________
______________________
// no return statement since return
// type is void.
}
```

Passing Parameters to a Method

There are some cases when we don’t know the number of parameters to be passed or an unexpected case to use more parameters than the declared number of parameters. In such cases we can use

- Passing Array as an Argument
- Passing Variable-arguments as an Argument
- Method Overloading.

Memory Allocation for Methods Calls

Method calls are implemented through a stack. Whenever a method is called a stack frame is created within the stack area and after that, the arguments passed to and the local variables and value to be returned by this called method are stored in this stack frame and when execution of the called method is finished, the allocated stack frame would be deleted. There is a stack pointer register that tracks the top of the stack which is adjusted accordingly.

Example: Accessor and Mutator Methods

```java
public class MethodCallExample {
    private int num;
    private String n;
    // Accessor (getter) methods
    public int getNumber() {
      return num;
    }

    public String getName() {
      return n;
    }
    // Mutator (setter) methods
    public void setNumber(int num) {
      this.num = num;
    }

    public void setName(String n) {
      this.n = n;
    }

    public void printDetails() {
        System.out.println("Number: " + num);
        System.out.println("Name: " + n);
    }

    public static void main(String[] args) {
        MethodCallExample g = new MethodCallExample();
        g.setNumber(123);
        g.setName("Pathshala");
        g.printDetails();
    }
}
Output
Number: 123
Name: Pathshala
```

Explanation: In the above example, the MethodCallExample class contains private fields num and n, with getter and setter methods to access and modify their values. The printDetails() method prints the values of num and n to the console. In the main method, the setNumber, setName, and printDetails methods are called to set and display the object's details.

Advantages of Methods

- **Reusability:** Methods allow us to write code once and use it many times.
- **Abstraction:** Methods allow us to abstract away complex logic and provide a simple interface for others to use.
- **Encapsulation:** Allow to encapsulate complex logic and data
- **Modularity:** Methods allow us to break up your code into smaller, more manageable units, improving the modularity of your code.
- **Customization:** Easy to customize for specific tasks.
- **Improved performance:** By organizing your code into well-structured methods, you can improve performance by reducing the amount of code.

Static Method vs Instance Method in Java

In Java, methods are mainly divided into two parts based on how they are connected to a class, which are the static method and the Instance method. The main difference between static and instance methods is listed below:

Static method: A static method is a part of the class and can be called without creating an object.
Instance method: Instance method belongs to an object, we need to create an object to use instance methods.

Difference Between Static Method and Instance Method

The following table lists the major differences between the static methods and the instance methods in Java.

Features	**Static method**	**Instance method**
Definition	Created using the static keyword and retrieved without creating an object.	Requires an object of its class to be invoked.
Access	Access only static variables and methods.	Can access both static and instance members.
this keyword	Cannot use this keyword within static methods.	Can use this keyword to refer to the current object.
Override	Does not support runtime polymorphism.	Supports runtime polymorphism.

Static Method

A static method can be created using the static keyword. It can be called without creating an object of the class, referenced by the class name itself, or a reference to the object of that class.

Memory Allocation of Static Methods

Static methods belong to the class, not its objects, and they are stored in the Permanent Generation Space of the heap. Their local variables and arguments are stored in the stack. They can be called without creating an instance of the class, using ClassName.methodName(args).

Important Points:

- Static methods are shared among all objects of the class.
- They cannot be overridden as they use static binding at compile time.
- If both superclass and subclass have static methods with the same name, it is called Method Hiding, where the subclass method hides the superclass method.

Example:

```
// Java program to demonstrate the static method
import java.io.*;

class StaticMethodExample {
  // static method
  public static void greet(){
    System.out.println("Hello Pathshala!");
  }

    public static void main (String[] args) {
       // calling the method directly
       greet();
        // calling the method using the class name
        StaticMethodExample.greet();
    }
}

Output
Hello Pathshala!
Hello Pathshala!
```

Explanation: The above example shows a static method greet() inside the StaticMethodExample class, static methods can be called without creating an object. In the main method, we are not creating an object of the StaticMethodExample class, we are calling the method directly by the class name which is StaticMethodExample and then we are printing the output.

Instance Method

Instance methods are the methods that require an object to work. We need to create an object of the class where the method is written, then only we can access the instance method.

Memory Allocation of Instance Method
Instance methods are stored in the Permanent Generation space of the heap (till Java 7, replaced by Metaspace from Java 8 for better efficiency). Their parameters, local variables, and return values are allocated on the stack. They can be called within their class or from other classes, based on their access modifiers.

Important Points:

- Instance methods belong to the object, not the class, and require an object to be called.
- They are stored in one memory location and identify their object through this pointer.
- They can be overridden as they use dynamic binding at runtime.

Example:

```
// Java program to demonstrate the use of instance method
import java.io.*;
class Test {
    String n = "";
    // Instance method
    public void test(String n) {
      this.n = n;
    }
}

class InstanceMethodExample {
    public static void main(String[] args) {
        // create an instance of the class
        Test t = new Test();
        // calling an instance method in the class InstanceMethodExample
        t.test("Pathshala360");
        System.out.println(t.n);
    }
}
Output
Pathshala360
```

Explanation: The above example shows how to use an instance method in Java. We are creating an object of the Test class and calling the test method to set a value and then we are printing the output.

Note: Instance method requires an object to be called.

Access Modifiers in Java

In Java, access modifiers are essential tools that define how the members of a class, like variables, methods, and even the class itself can be accessed from other parts of our program. They are an important part of building secure and modular code when designing large applications. Understanding default, private, protected, and public access modifiers is essential for writing efficient and structured Java programs. In this article, we will explore each modifier with examples to demonstrate their impact on Java development.

Types of Access Modifiers

There are 4 types of access modifiers available in Java:

1. Default - No keyword required
2. Private
3. Protected
4. Public

1. Default Access Modifier

When no access modifier is specified for a class, method, or data member, it is said to have the default access modifier by default. This means only classes within the same package can access it.

Example 1: Demonstrating Default Access Modifier Within the Same Package
In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of the second package.

```java
// default access modifier
package p1;

// Class DefaultExample is having default access modifier
class DefaultExample {
    void display() {
        System.out.println("Hello World!");
    }
}
```

Example 2: Error when Accessing Default Modifier Class across Packages
In this example, the program will show the compile-time error when we try to access a default modifier class from a different package.

```
// error while using class from different package with default modifier
package p2;
import p1.*;    // importing package p1

// This class is having default access modifier
class DefaultExampleNew {
    public static void main(String args[]) {
        // Accessing class DefaultExample from package p1
        DefaultExample o = new DefaultExample();
        o.display();
    }
}
```

2. Private Access Modifier
The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

- Any other class of the same package will not be able to access these members.
- Top-level classes or interfaces can not be declared as private because,
- private means "only visible within the enclosing class".
- protected means "only visible within the enclosing class and any subclasses".
- These modifiers in terms of application to classes, apply only to nested classes and not on top-level classes.

Example: In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
// Error while using class from different package with private access
modifier
package p1;
// Class A
class A {
    private void display() {
        System.out.println("Pathshala360");
    }
}
```

```java
// Class B
class B {
    public static void main(String args[]) {
        A obj = new A();
        // Trying to access private method of another class
        obj.display();
    }
}
```

Explanation: The above code will show a compile-time error when trying to access a private method from class B, even within the same package.

3. Protected Access Modifier

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

Example 1: In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method displayed in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

```java
// protected access modifier
package p1;
// Class A
public class A {
    protected void display() {
        System.out.println("Pathshala360");
    }
}
```

Example 2: In this example, we will create two packages, p1 and p2. Class A in p1 has a protected method display. Class B in p2 extends A and accesses the protected method through inheritance by creating an object of class B.

```java
package p2;
// importing all classes in package p1
import p1.*;
// Class B is subclass of A
class B extends A {
    public static void main(String args[]) {
        B obj = new B();
        obj.display();
    }
}
```

Explanation: The above example demonstrates that a protected method is accessible in a subclass from a different package using inheritance.

4. Public Access Modifier

The public access modifier is specified using the keyword public.

- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

Example 1: Here, the code shows that a public method is accessible within the same package.

```
// public modifier
package p1;
public class A {
    public void display() {
        System.out.println("Pathshala360");
    }
}
```

Example 2: Here, the example shows that a public method is accessible across packages.

```
// public access modifier
package p2;
import p1.*;
class B {
    public static void main(String args[]) {
        A obj = new A();
        obj.display();
    }
}
```

Important Points:

- If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member.
- Avoid public fields except for constants.

When to Use Each Access Modifier in Real-World Projects

- **Private:** This is used for encapsulating sensitive data and internal helper methods that should not be accessed outside the class.

  Example: Private fields in a model class with getter and setter methods.
- **Default (Package-Private):** This is suitable for classes and methods that should only be accessible within the same package, often used in package-scoped utilities or helper classes.
- **Protected:** This is ideal for methods and fields that should be accessible within the same package and subclasses, commonly used in inheritance-based designs like framework extensions.
- **Public:** This is used for classes, methods, or fields meant to be accessible from anywhere, such as API endpoints, service classes, or utility methods shared across different parts of an application.

Arrays in Java

Arrays in Java are one of the most fundamental data structures that allow us to store multiple values of the same type in a single variable. They are useful for storing and managing collections of data. Arrays in Java are objects, which makes them work differently from arrays in C/C++ in terms of memory management. For primitive arrays, elements are stored in a contiguous memory location, For non-primitive arrays, references are stored at contiguous locations, but the actual objects may be at different locations in memory.

Key features of Arrays:

- **Contiguous Memory Allocation (for Primitives):** Java array elements are stored in contiguous memory locations, which means that the elements are placed next to each other in memory.
- **Zero-based Indexing:** The first element of the array is at index 0.
- **Fixed Length:** Once an array is created, its size is fixed and cannot be changed.
- **Can Store Primitives & Objects:** Java arrays can hold both primitive types (like int, char, boolean, etc.) and objects (like String, Integer, etc.)

Example: This example demonstrates how to initialize an array and traverse it using a for loop to print each element.

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

        // size of array
        int n = arr.length;

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

```
Output
1 2 3 4 5
```

Basics of Arrays in Java

There are some basic operations we can start with as mentioned below:

1. Array Declaration

To declare an array in Java, use the following syntax:

```
type[] arrayName;
```

- type: The data type of the array elements (e.g., int, String).
- arrayName: The name of the array.

Note: The array is not yet initialized.

2. Create an Array

To create an array, you need to allocate memory for it using the new keyword:

```
// Creating an array of 5 integers
int[] numbers = new int[5];
```

This statement initializes the numbers array to hold 5 integers. The default value for each element is 0.

3. Access an Element of an Array

We can access array elements using their index, which starts from 0:

```
// Setting the first element of the array
numbers[0] = 10;
// Accessing the first element
int firstElement = numbers[0];
```

The first line sets the value of the first element to 10. The second line retrieves the value of the first element.

4. Change an Array Element

To change an element, assign a new value to a specific index:

```
// Changing the first element to 20
numbers[0] = 20;
```

5. Array Length

We can get the length of an array using the length property:

```
// Getting the length of the array
int length = numbers.length;
```

Now, we have completed basic operations so let us go through the in-depth concepts of Java Arrays, through the diagrams, examples, and explanations.

In-Depth Concepts of Java Array

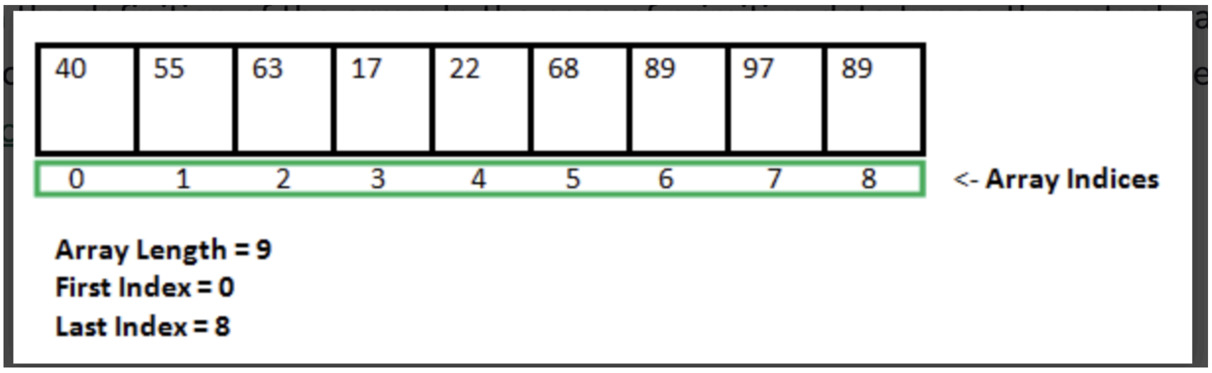
Following are some important points about Java arrays.

Array Properties

In Java, all arrays are dynamically allocated.

- Arrays may be stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using size of.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- Java arrays can also be used as a static field, a local variable, or a method parameter.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class, depending on the definition of the array. In the case of primitive data types, the actual values might be stored in contiguous memory locations (JVM does not guarantee this behavior). In the case of class objects, the actual objects are stored in a heap segment.



Note: This storage of arrays helps us randomly access the elements of an array [Support Random Access].

Creating, Initializing, and Accessing an Arrays in Java

For understanding the array we need to understand how it actually works. To understand this follow the flow mentioned below:

1. Declare
2. Initialize
3. Access

1. Declaring an Array
The general form of array declaration is

```
Method 1:
type var-name[];

Method 2:
type[] var-name;
```

The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class).

Note: It is just how we can create an array variable, no actual array exists. It merely tells the compiler that this variable (int Array) will hold an array of the integer type. Now, Let us provide memory storage to this created array.

2. Initialization an Array in Java
When an array is declared, only a reference of an array is created. The general form of new as it applies to one-dimensional arrays appears as follows:

```
var-name = new type [size];
```

Here, type specifies the type of data being allocated, size determines the number of elements in the array, and var-name is the name of the array variable that is linked to the array. To use new to allocate an array, you must specify the type and number of elements to allocate.

Example:

```
// declaring array
int intArray[];
// allocating memory to array
intArray = new int[20];
// combining both statements in one
int[] intArray = new int[20];
```

Note: The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types). Do refer to default array values in Java.

Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory to hold the array, using new, and assign it to the array variable. Thus, in Java, all arrays are dynamically allocated.

Array Literal in Java
In a situation where the size of the array and variables of the array are already known, array literals can be used.

```
// Declaring array literal
int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

- The length of this array determines the length of the created array.
- There is no need to write the new int[] part in the latest versions of Java.

3. Accessing Java Array Elements using for Loop
Now , we have created an Array with or without the values stored in it. Access becomes an important part to operate over the values mentioned within the array indexes using the points mentioned below:
- Each element in the array is accessed via its index.
- The index begins with 0 and ends at (total array size)-1.
- All the elements of the array can be accessed using Java for Loop.

Let us check the syntax of basic for loop to traverse an array:

```
// Accessing the elements of the specified array
 for (int i = 0; i < arr.length; i++)
              System.out.println("Element at index " + i + " : "+
arr[i]);
```

Implementation:

```
// Java program to illustrate creating an array of integers,  puts some
values in the array, and prints each value to standard output.

class ArrayExample {
    public static void main(String[] args) {
        // declares an Array of integers.
        int[] arr;
        // allocating memory for 5 integers.
        arr = new int[5];
        // initialize the elements of the array first to last(fifth)
element
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;
        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at index " + i + " : " +
arr[i]);
    }
}

Output
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

Types of Arrays in Java

Java supports different types of arrays:

1. Single-Dimensional Arrays

These are the most common type of arrays, where elements are stored in a linear order.

// A single-dimensional array

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

2. Multi-Dimensional Arrays

Arrays with more than one dimension, such as two-dimensional arrays(matrices).

```
// A 2D array (matrix)
int[][] multiDimArray = {
      {1, 2, 3},
     {4, 5, 6},
     {7, 8, 9} };

You can also access java arrays using for each loops.
```

Arrays of Objects in Java

An array of objects is created like an array of primitive-type data items in the following way.

Syntax:

```
Method 1:
ObjectType[] arrName;

Method 2:
ObjectType arrName[];
```

Example of Arrays of Objects

Example: Here we are taking a student class and creating an array of Student with five Student objects stored in the array. The Student objects have to be instantiated using the constructor of the Student class, and their references should be assigned to the array elements.

```
// Java program to illustrate creating an array of objects
class Student {
    public int rollNo;
    public String name;

    Student(int rollNo, String name){
        this.rollNo = rollNo;
        this.name = name;
    }
}

public class Main {
    public static void main(String[] args){
        // declares an Array of Student
        Student[] arr;
        // allocating memory for 5 objects of type Student.
        arr = new Student[5];
        // initialize the elements of the array
        arr[0] = new Student(1, "Test1");
        arr[1] = new Student(2, "Test2");
        arr[2] = new Student(3, "Test3");
        arr[3] = new Student(4, "Test4");
        arr[4] = new Student(5, "Test15);

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at " + i + " : { "
                               + arr[i].rollNo + " "
                               + arr[i].name+" }");
    }
}

Output
Element at 0 : { 1 Test1 }
Element at 1 : { 2 Test2 }
Element at 2 : { 3 Test3 }
Element at 3 : { 4 Test4 }
Element at 4 : { 5 Test5 }
```

Example: An array of objects is also created like

```
// Java program to illustrate creating an array of objects
class Student{
    public String name;

    Student(String name){
        this.name = name;
    }

    @Override
    public String toString(){
        return name;
    }
}

public class Main{
    public static void main (String[] args){
        // declares an Array and initializing the elements of the array
        Student[] myStudents = new Student[]{
          new Student("Test1"),new Student("Test2"),
          new Student("Test3"),new Student("Test4")
        };

        // accessing the elements of the specified array
        for(Student m : myStudents){
            System.out.println(m);
        }
    }
}

Output
Test1
Test2
Test3
Test4
```

What happens if we try to access elements outside the array size?

JVM throws ArrayIndexOutOfBoundsException to indicate that the array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of an array.

Below code shows what happens if we try to access elements outside the array size:

```
// Code for showing error "ArrayIndexOutOfBoundsException"
public class ArrayExample {
    public static void main(String[] args) {
        int[] arr = new int[4];
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;

        System.out.println(
            "Trying to access element outside the size of array");
        System.out.println(arr[5]);
    }
}

Output
Trying to access element outside the size of array
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 5 out of bounds for length 4
at ArrayExample.main(ArrayExample.java:13)
```

Multidimensional Arrays in Java

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax:

There are 2 methods to declare Java Multidimensional Arrays as mentioned below:

```
// Method 1
datatype [][] arrayrefvariable;
// Method 2
datatype arrayrefvariable[][];
```

Declaration:

```
// 2D array or matrix
int[][] intArray = new int[10][20];
// 3D array
int[][][] intArray = new int[10][20][10];
```

Java Multidimensional Arrays Examples

Example: Let us start with basic two dimensional Array declared and initialized.

```
// Java Program to demonstrate Multidimensional Array
import java.io.*;

class Example {
    public static void main(String[] args){
        // Two Dimensional Array Declared and Initialized
        int[][] arr = new int[3][3];

        // Number of Rows
        System.out.println("Rows : " + arr.length);

        // Number of Columns
        System.out.println("Columns : " + arr[0].length);
    }
}

Output
Rows:3
Columns:3
```

Example: Now, after declaring and initializing the array we will check how to Traverse the Multidimensional Array using a for loop.

```
// Java Program to Multidimensional Array
public class multiDimensional {
    public static void main(String args[]){
        // declaring and initializing 2D array
        int arr[][] = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };

        // printing 2D array
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}

Output
2 7 9
3 6 1
7 4 2
```

Passing Arrays to Methods

Like variables, we can also pass arrays to methods. For example, the below program passes the array to method sum to calculate the sum of the array's values.

```
// Java program to demonstrate passing of array to method
public class Test {
    public static void main(String args[]) {
        int arr[] = { 3, 1, 2, 5, 4 };
        // passing array to method m1
        sum(arr);
    }

    public static void sum(int[] arr) {
        // getting sum of array values
        int sum = 0;
        for (int i = 0; i < arr.length; i++)
            sum += arr[i];

        System.out.println("sum of array values : " + sum);
    }
}

Output
sum of array values : 15
```

Returning Arrays from Methods

As usual, a method can also return an array. For example, the below program returns an array from method m1.

```
// Java program to demonstrate return of array from method

class Test {
    public static void main(String args[]) {
        int arr[] = m1();
        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }

    public static int[] m1() {
        return new int[] { 1, 2, 3 };
    }
}

Output
1 2 3
```

Advantages of Java Arrays

- **Efficient Access:** Accessing an element by its index is fast and has constant time complexity, O(1).
- **Memory Management:** Arrays have fixed size, which makes memory management straightforward and predictable.
- **Data Organization:** Arrays help organize data in a structured manner, making it easier to manage related elements.

Disadvantages of Java Arrays

- **Fixed Size:** Once an array is created, its size cannot be changed, which can lead to memory waste if the size is overestimated or insufficient storage if underestimated.
- **Type Homogeneity:** Arrays can only store elements of the same data type, which may require additional handling for mixed types of data.
- **Insertion and Deletion:** Inserting or deleting elements, especially in the middle of an array, can be costly as it may require shifting elements.

Array Class

The Arrays class in the java.util package is a part of the Java Collection Framework. This class provides static methods to dynamically create and access Java arrays. It consists of only static methods and the methods of an Object class. The methods of this class can be used by the class name itself.

The class hierarchy is as follows:

```
java.lang.Object
 ? java.util.Arrays
```

Class Declaration

```
public class Arrays extends Object
```

To use Arrays,

```
Arrays.<function name>;
```

Methods in Java Array Class

The Arrays class of the java.util package contains several static methods that can be used to fill, sort, search, etc in arrays. Now let us discuss the methods of this class which are shown below in a tabular format as follows:

Methods	Action Performed
asList()	Returns a fixed-size list backed by the specified Arrays
binarySearch()	Searches for the specified element in the array with the help of the Binary Search Algorithm
binarySearch(array, fromIndex, toIndex, key, Comparator)	Searches a range of the specified array for the specified object using the Binary Search Algorithm
compare(array 1, array 2)	Compares two arrays passed as parameters lexicographically.
copyOf(originalArray, newLength)	Copies the specified range of the specified array into new Arrays.
copyOfRange(originalArray, fromIndex, endIndex)	Copies the specified range of the specified array into new Arrays.
deepEquals(Object[] a1, Object[] a2)	Returns true if the two arrays are deeply equal to one another.

deepHashCode(Object[] a)	Returns a hash code based on the "deep contents" of the specified Arrays.
deepToString(Object[] a)	Returns a string representation of the "deep contents" of the specified Arrays.
equals(array1, array2)	Checks if both the arrays are equal or not.
fill(originalArray, fillValue)	Assign this fill value to each index of this array.
hashCode(originalArray)	Returns an integer hashCode of this array instance.
mismatch(array1, array2)	Finds and returns the index of the first unmatched element between the two specified arrays.
parallelPrefix(originalArray, fromIndex, endIndex, functionalOperator	Performs parallelPrefix for the given range of the array with the specified functional operator.
parallelPrefix(originalArray, fromIndex, endIndex, functionalOperator)	Performs parallelPrefix for the given range of the array with the specified functional operator.
parallelPrefix(originalArray, operator)	Performs parallelPrefix for complete array with the specified functional operator.
parallelSetAll(originalArray, functionalGenerator)	Sets all the elements of this array in parallel, using the provided generator function.
parallelSort(originalArray)	Sorts the specified array using parallel sort.
setAll(originalArray, functionalGenerator)	Sets all the elements of the specified array using the generator function provided.
sort(originalArray)	Sorts the complete array in ascending order.
sort(originalArray, fromIndex, endIndex)	Sorts the specified range of arrays in ascending order.

sort(T[] a, int fromIndex, int toIndex, Comparator< super T> c)	Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
sort(T[] a, Comparator< super T> c)	Sorts the specified array of objects according to the order induced by the specified comparator.
spliterator(originalArray)	Returns a Spliterator covering all of the specified Arrays.
spliterator(originalArray, fromIndex, endIndex)	Returns a Spliterator of the type of the array covering the specified range of the specified arrays.
stream(originalArray)	Returns a sequential stream with the specified array as its source.
toString(originalArray)	It returns a string representation of the contents of this array. The string representation consists of a list of the array’s elements, enclosed in square brackets (“[]”). Adjacent elements are separated by the characters a comma followed by a space. Elements are converted to strings as by String.valueOf() function.

Implementation

Example 1: asList() method. This method converts an array into a list.

```
// Java Program to Demonstrate Arrays Class via asList() method
// Importing Arrays utility class from java.util package
import java.util.Arrays;
class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };

        // To convert the elements as List
        System.out.println("Integer Array as List: "+
Arrays.asList(intArr));
    }
}

Output
Integer Array as List: [[I@19469ea2]
```

Note: When asList() is used with primitive arrays, it shows the memory reference of the array instead of the list contents. This happens because the asList() method returns a fixed-size list backed by the original array, and for primitive types like int[], it treats the array as an object, not as a list of values.

Example 2: binarySearch() Method. This method searches for the specified element in the array with the help of the binary search algorithm.

```
// Java Program to Demonstrate Arrays Class via binarySearch() method
// Importing Arrays utility class from java.util package
import java.util.Arrays;

public class Main {
    public static void main(String[] args){
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };

        Arrays.sort(intArr);
        int intKey = 22;

        // Print the key and corresponding index
        System.out.println(
            intKey + " found at index = "
            + Arrays.binarySearch(intArr, intKey));
    }
}

Output
22 found at index = 3
```

Example 3: binarySearch(array, fromIndex, toIndex, key, Comparator) Method This method searches a range of the specified array for the specified object using the binary search algorithm.

```
// Java program to demonstrate Arrays.binarySearch() method
import java.util.Arrays;

public class Main {
    public static void main(String[] args){
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        Arrays.sort(intArr);
        int intKey = 22;
        System.out.println(intKey + " found at index = "
            + Arrays.binarySearch(intArr, 1, 3, intKey));
    }
}

Output
22 found at index = -4
```

Example 4: compare(array 1, array 2) Method
This method returns the difference as an integer lexicographically.

```java
// Java program to demonstrate Arrays.compare() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // Get the second Array
        int intArr1[] = { 10, 15, 22 };
        // To compare both arrays
        System.out.println("Integer Arrays on comparison: "
                           + Arrays.compare(intArr, intArr1));
    }
}

Output
Integer Arrays on comparison: 1
```

Example 5: compareUnsigned(array 1, array 2) Method
This method is used to compare two arrays of primitive int[] values (or other primitive array types) lexicographically, but with the values treated as unsigned integers.

```java
// Java program to demonstrate Arrays.compareUnsigned() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Arrays
        int intArr[] = { 10, 20, 15, 22, 35 };
        // Get the second Arrays
        int intArr1[] = { 10, 15, 22 };
        // To compare both arrays
        System.out.println("Integer Arrays on comparison: "
                           + Arrays.compareUnsigned(intArr, intArr1));
    }
}

Output
Integer Arrays on comparison: 1
```

Example 6: copyOf(originalArray, newLength) Method
This method is used to copy the elements of an array into a new array of the specified new length.

```
// Java program to demonstrate Arrays.copyOf() method
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To print the elements in one line
        System.out.println("Integer Array: "
                           + Arrays.toString(intArr));

        System.out.println("\nNew Arrays by copyOf:\n");

        System.out.println("Integer Array: "
                           + Arrays.toString(
                                 Arrays.copyOf(intArr, 10)));
    }
}
Output
Integer Array: [10, 20, 15, 22, 35]
New Arrays by copyOf:
Integer Array: [10, 20, 15, 22, 35, 0, 0, 0, 0, 0]
```

Example 7: copyOfRange(originalArray, fromIndex, endIndex) Method
This method is used to copy a range of elements from an array into a new array.

```
// Java program to demonstrate Arrays.copyOfRange() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        int intArr[] = { 10, 20, 15, 22, 35 };
        System.out.println("Integer Array: " + Arrays.toString(intArr));
        System.out.println("\nNew Arrays by copyOfRange:\n");
        System.out.println("Integer Array: " + Arrays.toString(
                                 Arrays.copyOfRange(intArr, 1, 3)));
    }
}
Output
Integer Array: [10, 20, 15, 22, 35]
New Arrays by copyOfRange:
Integer Array: [20, 15]
```

Example 8: deepEquals(Object[] a1, Object[] a2) Method
This method is used to compare two arrays of objects to check if they are deeply equal.

```
// Java program to demonstrate Arrays.deepEquals() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Arrays
        int intArr[][] = { { 10, 20, 15, 22, 35 } };
        // Get the second Arrays
        int intArr1[][] = { { 10, 15, 22 } };
        // To compare both arrays
        System.out.println("Integer Arrays on comparison: "
                           + Arrays.deepEquals(intArr, intArr1));
    }
}

Output
Integer Arrays on comparison: false
```

Example 9: deepHashCode(Object[] a) Method
This method is used to compute a hash code for an array of objects recursively.

```
// Java program to demonstrate Arrays.deepHashCode() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[][] = { { 10, 20, 15, 22, 35 } };
        // To get the dep hashCode of the arrays
        System.out.println("Integer Array: "
                           + Arrays.deepHashCode(intArr));
    }
}

Output
Integer Array: 38475344
```

Example 10: deepToString(Object[] a) Method \
This method is used to return a string representation of an array recursively.

```
// Java program to demonstrate Arrays.deepToString() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[][] = { { 10, 20, 15, 22, 35 } };
        // To get the deep String of the arrays
        System.out.println("Integer Array: "
                           + Arrays.deepToString(intArr));
    }
}
Output
Integer Array: [[10, 20, 15, 22, 35]]
```

Example 11: equals(array1, array2) Method
This method is used to compare two arrays to check if they are equal.

```
// Java program to demonstrate Arrays.equals() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Arrays
        int intArr[] = { 10, 20, 15, 22, 35 };
        // Get the second Arrays
        int intArr1[] = { 10, 15, 22 };
        // To compare both arrays
        System.out.println("Integer Arrays on comparison: "
                           + Arrays.equals(intArr, intArr1));
    }
}

Output
Integer Arrays on comparison: false
```

Example 12: fill(originalArray, fillValue) Method
This method is used to fill an entire array or a subrange of an array with a specific value.

```
// Java program to demonstrate Arrays.fill() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Arrays
        int intArr[] = { 10, 20, 15, 22, 35 };
        int intKey = 22;
        Arrays.fill(intArr, intKey);
        // To fill the arrays
        System.out.println("Integer Array on filling: "
                           + Arrays.toString(intArr));
    }
}
Output
Integer Array on filling: [22, 22, 22, 22, 22]
```

Example 13: hashCode(originalArray) Method
This method is used to compute a hash code for an array

```
// Java program to demonstrate Arrays.hashCode() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To get the hashCode of the arrays
        System.out.println("Integer Array: "
                           + Arrays.hashCode(intArr));
    }
}
Output
Integer Array: 38475313
```

Example 14: mismatch(array1, array2) Method
This method is used to find the index of the first mismatched element between two arrays

```
// Java program to demonstrate Arrays.mismatch() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Arrays
        int intArr[] = { 10, 20, 15, 22, 35 };
        // Get the second Arrays
        int intArr1[] = { 10, 15, 22 };
        // To compare both arrays
        System.out.println("The element mismatched at index: "
                           + Arrays.mismatch(intArr, intArr1));
    }
}

Output
The element mismatched at index: 1
```

Example 15: parallelSort(originalArray) Method
This method is used to sort an array in parallel.

```
// Java program to demonstrate Arrays.parallelSort() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To sort the array using parallelSort
        Arrays.parallelSort(intArr);
        System.out.println("Integer Array: "
                           + Arrays.toString(intArr));
    }
}

Output
Integer Array: [10, 15, 20, 22, 35]
```

Example 16: sort(originalArray) Method
This method is used to sort an array in ascending order

```
// Java program to demonstrate Arrays.sort() method
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To sort the array using normal sort-
        Arrays.sort(intArr);
        System.out.println("Integer Array: "
                           + Arrays.toString(intArr));
    }
}

Output
Integer Array: [10, 15, 20, 22, 35]
```

Example 17: sort(originalArray, fromIndex, endIndex) Method. This method is used to sort a specified range of an array in ascending order

```
// Java program to demonstrate Arrays.sort() method
import java.util.Arrays;
public class Main{
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To sort the array using normal sort
        Arrays.sort(intArr, 1, 3);
        System.out.println("Integer Array: "
                           + Arrays.toString(intArr));
    }
}

Output
Integer Array: [10, 15, 20, 22, 35]
```

Example 18: sort(T[] a, int fromIndex, int toIndex, Comparator< super T> c) Method. This method is used to sort a specified range of an array using a custom comparator for sorting.

```java
// Java program to demonstrate working of Comparator interface
import java.util.*;
import java.lang.*;
import java.io.*;

class Student {
    int rollNo;
    String name, address;
    public Student(int rollNo, String name, String address) {
        this.rollNo = rollNo;
        this.name = name;
        this.address = address;
    }
    public String toString() {
        return this.rollNo + " " + this.name + " " + this.address;
    }
}

class SortByRoll implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.rollno - b.rollno;
    }
}

class Main {
    public static void main(String[] args) {
        Student[] arr = { new Student(111, "bbbb", "london"),
                          new Student(131, "aaaa", "nyc"),
                          new Student(121, "cccc", "jaipur") };

        System.out.println("Unsorted");
        for (int i = 0; i < arr.length; i++) {
             System.out.println(arr[i]);
        }
        Arrays.sort(arr, 1, 2, new SortByRoll());
        System.out.println("\nSorted by rollNo");
        for (int i = 0; i < arr.length; i++)
            System.out.println(arr[i]);
    }
}
```

```
Output
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollNo
111 bbbb london
131 aaaa nyc
121 cccc jaipur
```

Example 19: sort(T[] a, Comparator< super T> c) Method
This method is used to sort an entire array of objects (T[]) using a custom comparator (Comparator<? super T>).

```
// Java program to demonstrate working of Comparator interface
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student {
    int rollNo;
    String name, address;

    // Constructor
    public Student(int rollNo, String name, String address) {
        this.rollNo = rollNo;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString() {
        return this.rollNo + " " + this.name + " " + this.address;
    }
}

class SortByRoll implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.rollNo - b.rollNo;
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Student[] arr = { new Student(111, "bbbb", "london"),
                          new Student(131, "aaaa", "nyc"),
                          new Student(121, "cccc", "jaipur") };

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

        Arrays.sort(arr, new SortByRoll());

        System.out.println("\nSorted by rollNo");
        for (int i = 0; i < arr.length; i++)
            System.out.println(arr[i]);
    }
}

Output
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc
```

Example 20: spliterator(originalArray) Method
This method is used to create a Spliterator for the given array.

```
// Java program to demonstrate Arrays.spliterator() method
import java.util.Arrays;

public class Main{
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To sort the array using normal sort
        System.out.println("Integer Array: "
                           + Arrays.spliterator(intArr));
    }
}

Output
Integer Array: java.util.Spliterators$IntArraySpliterator@4e50df2e
```

Example 21: spliterator(originalArray, fromIndex, endIndex) Method
This method is used to create a Spliterator for a subrange of the given array, starting from fromIndex (inclusive) to toIndex (exclusive).

```
// Java program to demonstrate Arrays.spliterator() method
import java.util.Arrays;

public class Main{
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };

        // To sort the array using normal sort
        System.out.println("Integer Array: "
                           + Arrays.spliterator(intArr, 1, 3));
    }
}

Output
Integer Array: java.util.Spliterators$IntArraySpliterator@4e50df2e
```

Example 22: stream(originalArray) Method
This method is used to convert an array into a stream.

```
// Java program to demonstrate Arrays.stream() method
import java.util.Arrays;

public class Main{
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To get the Stream from the array
        System.out.println("Integer Array: "
                           + Arrays.stream(intArr));
    }
}

Output
Integer Array: java.util.stream.IntPipeline$Head@7291c18f
```

Example 23: toString(originalArray) Method
This method is used to convert an array into a human-readable string representation.

```
// Java program to demonstrate Arrays.toString() method
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        // Get the Array
        int intArr[] = { 10, 20, 15, 22, 35 };
        // To print the elements in one line
        System.out.println("Integer Array: "
                           + Arrays.toString(intArr));
    }
}

Output
Integer Array: [10, 20, 15, 22, 35]
```

Final Arrays in Java

As we all know, the final variable declared can only be initialized once whereas the reference variable once declared final can never be reassigned as it will start referring to another object which makes usage of the final impracticable. But note here that with final we are bound not to refer to another object but within the object data can be changed which means we can change the state of the object but not reference.

Arrays are also an object so final arrays come into play. The same concept does apply to final arrays, that are we can not make the final array refer to some other array but the data within an array that is made final can be changed/manipulated.

Example:

```
// Can Be Reassigned But Not Re-referred
import java.util.*;

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

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

Output
1 2 3 4 1
```

Implementation:

```
// Java Program to Illustrate Final Arrays Importing required classes
import java.util.*;
class Main {
    public static void main(String args[]) {
        final int arr[] = { 1, 2, 3, 4, 5 };
        // Iterating over integer array
        for (int i = 0; i < arr.length; i++) {
            arr[i] = arr[i] * 10;
            System.out.println(arr[i]);
        }
    }
}
```

```
Output
10
20
30
40
50
```

Output Explanation: The array arr is declared as final, but the elements of an array are changed without any problem. Arrays are objects and object variables are always references in Java. So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else.

Example A:

```
class Test {
    int p = 20;
    public static void main(String args[]) {
        final Test t = new Test();
        t.p = 30;
        System.out.println(t.p);
    }
}

Output
30
```

Example B:

```
class Test {
    int p = 20;
    public static void main(String args[]) {
        final Test t1 = new Test();
        Test t2 = new Test();

        // Assigning values into other objects
        t1 = t2;
        System.out.println(t1.p);
    }
}
Output: Compiler Error: cannot assign a value to final variable t1
```

Above program compiles without any error and program 2 fails in compilation. Let us discuss why the error occurred.

So a final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of the array can be modified. Let us propose an example below justifying the same.

Example:

// Java Program to Illustrate Members in Final Array can be Modified

```
class Test{
    public static void main(String args[]) {
        // Declaring a final array
        final int arr1[] = { 1, 2, 3, 4, 5 };
        // Declaring normal integer array
        int arr2[] = { 10, 20, 30, 40, 50 };
        // Assigning values to each other
        arr2 = arr1;
        arr1 = arr2;
        // Now iterating over normal integer array
        for (int i = 0; i < arr2.length; i++)
            System.out.println(arr2[i]);
    }
}
Output: Exception happened
```

Example:

```
// Import Arrays class for toString() method
import java.util.Arrays;

public class FinalArrayExample {
    public static void main(String[] args) {
        final int[] numbers = { 1, 2, 3, 4, 5 };
        numbers[0] = 10;
        System.out.println("Array after modifying first element: "
            + Arrays.toString(numbers));
    }
}
Output: Array after modifying first element: [10, 2, 3, 4, 5]
```

Java Array Programs

1. Java Program to Remove Duplicate Elements From the Array
2. How to Find Length or Size of an Array in Java?
3. How to Convert an Array to String in Java?
4. How to Add an Element to an Array in Java?
5. Check If a Value is Present in an Array in Java
6. Remove an Element at Specific Index from an Array in Java
7. Find the Index of an Array Element in Java
8. Binary Tree (Array implementation)
9. Java - Loop Through an Array
10. Print a 2D Array or Matrix in Java
11. Conversion of Array To ArrayList in Java
12. Array Index Out Of Bounds Exception in Java
13. Array vs ArrayList in Java
14. Compare Two Arrays in Java
15. How to Take Array Input From Users in Java?
16. Java Program to Sort the Elements of an Array in Descending Order
17. Java Program to Find Common Elements Between Two Arrays