
🔹 Topic 13: Arrays and Its Types

✅ Definition:

- An **Array** in Java is a container object that holds a fixed number of elements of the same data type.
- It stores elements in contiguous memory locations and provides indexed access to each element.

There are three main types of arrays in Java:

1. **One-dimensional (1D) Array** – Simple linear structure
2. **Two-dimensional (2D) Array** – Array of arrays, resembling a matrix
3. **Multidimensional Array** – Array with more than two levels of nesting

---

✅ Use Case:

Scenario	Array Type
Storing marks of students	1D Array
Storing a chessboard layout	2D Array
Handling multi-level data	Multidimensional

Daily temperatures	1D Array
Employee payroll table	2D Array

✅ Real-Time Usage:

Application Area	Example
Data Analytics	Matrix calculations using 2D arrays
Image Processing	RGB pixels in 2D arrays
Inventory System	Product counts by day (2D)
Banking Transaction Logs	Array of daily transaction amounts
Result Management	1D array for scores, 2D for total data

✅ Architecture Diagram: Array Memory Layout

🔷 1D Array:

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

```
  Index:    0     1     2     3     4
Memory: [___][___][___][___][___]
```

🔷 2D Array:

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

```
    Col → 0   1   2
Row ↓
   0      [0][0][0]
   1      [0][0][0]
```

2 [0][0][0]

✅ Syntax:

1D Array:

```
int[] numbers = new int[5];
numbers[0] = 10;
```

2D Array:

```
int[][] matrix = new int[2][3];
matrix[0][1] = 5;
```

Multidimensional Array:

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

✅ Keywords:

`array`, `index`, `length`, `multidimensional`, `new`, `[]`, `loop`

✅ Simple Example Code:

```
int[] nums = {10, 20, 30};
for (int i = 0; i < nums.length; i++) {
    System.out.println(nums[i]);
}
```

✅ Detailed Example Code with Real-Time Usage:

```
// Sum of all sales over 3 days in a store (2D array)
public class SalesReport {
    public static void main(String[] args) {
        int[][] sales = {
            {100, 200, 150}, // Day 1
            {90, 120, 100},  // Day 2
            {130, 170, 180}  // Day 3
        };

        int total = 0;
        for (int i = 0; i < sales.length; i++) {
            for (int j = 0; j < sales[i].length; j++) {
                total += sales[i][j];
            }
        }

        System.out.println("Total Sales: ₹" + total);
    }
}
```

✅ Sub-Topics with Examples:

🔹 Dynamic Initialization:

```
int[] arr = new int[5];
arr[0] = 10;
```

🔹 Enhanced for-loop:

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

```
}
```

🔹 **Multidimensional Array:**

```
int[][] cube = {
    {1, 2},
    {3, 4}
};
System.out.println(cube[1][1]); // 4
```

✅ Real-Time Example Code:

```
// Calculate average score of students
int[] scores = {75, 85, 90, 70, 80};
int sum = 0;

for (int score : scores) {
    sum += score;
}

double avg = (double) sum / scores.length;
System.out.println("Average Score: " + avg);
```

✅ 15 MCQ Questions (with Answers):

1. What is the index of the first element in an array?

   A) 0

2. Which is the correct declaration of a 1D array?

   B) int[] arr = new int[5];

3. Can arrays hold different data types?

   B) No

4. What does `arr.length` return?

   A) Number of elements in the array

5. How is a 2D array declared?

   C) int[][] matrix = new int[2][2];

6. What will `arr[5]` do if array size is 5?

   D) Throws ArrayIndexOutOfBoundsException

7. Can we store `null` in an array?

   A) Yes, for object arrays

8. How do you assign value to an array?

   C) arr[0] = 10;

9. Is it possible to return an array from a method?

   A) Yes

10. Which loop is best for traversing arrays?

A) for loop

11. Can an array store objects?

A) Yes

12. What is the default value for int[] array elements?

C) 0

13. Are arrays mutable in Java?

B) Yes

14. Can array size be changed after initialization?

D) No

15. Which array type allows matrix representation?

C) 2D array

✅ 20 Interview Questions & Answers:

1. **Q:** What is an array in Java?

A: A container object that holds fixed-size elements of the same type.

2. **Q:** Types of arrays in Java?

   A: 1D, 2D, and multidimensional arrays.

3. **Q:** Difference between `int[] arr;` and `int arr[];`?

   A: Both are valid; former is preferred.

4. **Q:** Can array size be dynamic?

   A: Not after creation; but use collections like `ArrayList`.

5. **Q:** How to access the last element?

   A: `arr[arr.length - 1]`

6. **Q:** Can we return an array from a function?

   A: Yes.

7. **Q:** What exception occurs on invalid index?

   A: `ArrayIndexOutOfBoundsException`

8. **Q:** Can arrays hold objects?

   A: Yes, like `String[]`, `Student[]`

9. **Q:** What is the default value in an int array?

   A: 0

10. **Q:** Can we sort an array?

    A: Yes, using `Arrays.sort()`

11. **Q:** What is `arr.length`?

    A: Property for array size.

12. **Q:** Difference between array and ArrayList?

    A: Array is fixed-size; ArrayList is resizable.

13. **Q:** Are arrays primitive or object?

    A: Arrays are objects.

14. **Q:** Is it possible to initialize an array at declaration?

    A: Yes: `int[] a = {1,2,3};`

15. **Q:** Can we use enhanced for loop for 2D arrays?

    A: Yes, nested enhanced for-loops.

16. **Q:** Can arrays store `null`?

    A: Yes, in object arrays.

17. **Q:** How to copy arrays?

    A: `Arrays.copyOf()` or loop.

18. **Q:** Is deep copy possible with arrays?
    A: Yes, manually or with utilities.

19. **Q:** How to create a ragged/jagged array?
    A: `int[][] arr = new int[3][]; arr[0] = new int[2];`

20. **Q:** Can array be used in method parameters?
    A: Yes, e.g., `void display(int[] nums)`

✅ Topic Outcome:

After learning this topic, students will:

- Understand how to declare, initialize, and manipulate arrays
- Handle real-time data in 1D and 2D formats
- Solve matrix-based problems
- Avoid common pitfalls like index errors

✅ Summary:

- Arrays form the backbone of data handling in Java.
- From student scores to image grids, mastering arrays prepares you to solve real-world problems efficiently.

- Though fixed in size, they are fast, reliable, and powerful for structured data storage.

🔹 Topic 14: Array Methods and Its Operations

✅ Definition:

- **Array operations** refer to actions performed on arrays such as sorting, searching, copying, filling, and comparing.
- Java does not support methods **directly on arrays** (like objects), but the `java.util.Arrays` **utility class** provides a set of static methods to manipulate arrays easily.
- These operations are essential for data processing and manipulation in Java programs.

✅ Use Case:

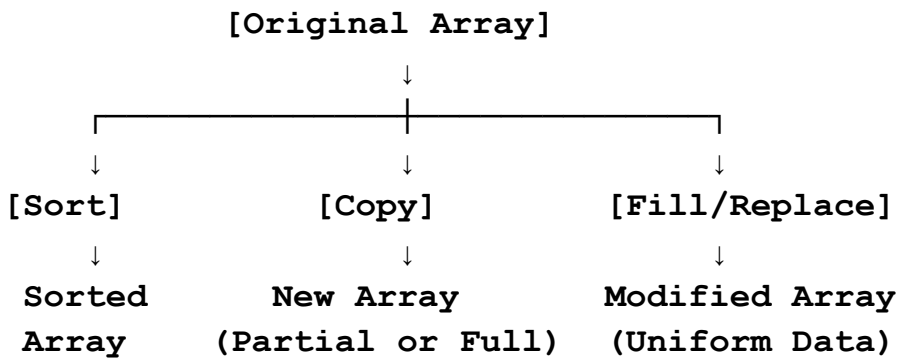
Use Case	Operation
Sorting student marks	`Arrays.sort()`
Searching a name in list	`Arrays.binarySearch()`
Copying contents to another	`Arrays.copyOf()`
Filling default values	`Arrays.fill()`

Comparing two arrays	`Arrays.equals()`

✅ Real-Time Usage:

Application Area	Real-Time Example
Online Test Portal	Sorting leaderboard scores
E-Commerce Cart	Comparing item lists
School Report System	Filling missing grades with default values
Product Inventory	Copying item list to a backup array
Hospital Patient Logs	Searching patient ID in an array

✅ Architecture Diagram: Common Array Operations Flow



✅ Syntax:

```
// Sorting an array
Arrays.sort(array);

// Copying an array
int[] copy = Arrays.copyOf(originalArray, newLength);
// Comparing arrays
Arrays.equals(arr1, arr2);

// Filling array
Arrays.fill(array, value);

// Searching
int index = Arrays.binarySearch(array, key);
```

✅ Keywords:

`Arrays`, `sort`, `fill`, `copyOf`, `equals`, `binarySearch`, `utility`, `static`

✅ Simple Example Code:

```
int[] nums = {5, 2, 9, 1};
Arrays.sort(nums); // nums = [1, 2, 5, 9]
```

✅ Detailed Example Code with Real-Time Usage:

```java
// Fill empty scores with default value and sort them
import java.util.Arrays;

public class ArrayOperations {
    public static void main(String[] args) {
        int[] scores = new int[5];

        // Fill all with default score
        Arrays.fill(scores, 40);

        // Update some actual scores
        scores[1] = 88;
        scores[3] = 76;

        // Sort scores
        Arrays.sort(scores);

        System.out.println("Sorted Scores: " +
Arrays.toString(scores));
    }
}
```

✅ Sub-Topics with Examples:

🔹 Arrays.toString():

```java
int[] a = {1, 2, 3};
System.out.println(Arrays.toString(a)); // [1, 2, 3]
```

🔹 Arrays.copyOfRange():

```java
int[] b = Arrays.copyOfRange(a, 1, 3); // [2, 3]
```

🔹 **Arrays.deepEquals() for 2D:**

```
int[][] arr1 = {{1, 2}, {3, 4}};
int[][] arr2 = {{1, 2}, {3, 4}};
System.out.println(Arrays.deepEquals(arr1, arr2)); // true
```

🔹 **Arrays.parallelSort():**

```
int[] bigData = {20, 10, 50, 40};
Arrays.parallelSort(bigData);
```

✅ Real-Time Example Code:

```
// Binary search in sorted product IDs
int[] productIDs = {101, 203, 405, 608};
Arrays.sort(productIDs); // Required before binary search

int index = Arrays.binarySearch(productIDs, 405);
if (index >= 0) {
    System.out.println("Product found at index: " + index);
} else {
    System.out.println("Product not found");
}
```

✅ 15 MCQ Questions (with Answers):

1. Which class provides utility methods for arrays?

   C) Arrays

2. What does `Arrays.sort()` do?

   A) Sorts the array in ascending order

3. Which method checks if two arrays are equal?

   D) Arrays.equals(arr1, arr2)

4. What is the use of `Arrays.fill()`?

   C) Fill array with a specific value

5. Can `Arrays.binarySearch()` be used on unsorted arrays?

   B) No

6. Which method prints array in string format?

   C) Arrays.toString()

7. What will `Arrays.copyOf(arr, 3)` return?

   A) A new array with first 3 elements

8. Can arrays store null values?

   A) Yes, for object arrays

9. Which method is used for partial copying?

   C) copyOfRange()

10. Which method can sort large arrays faster using multi-threading?

D) parallelSort()

11. What is the return type of `binarySearch()`?

B) int (index or -1)

12. What happens if `copyOf()` length is larger than original?

C) Extra elements are filled with default values

13. Which method checks deeply nested arrays?

A) deepEquals()

14. What does `Arrays.equals()` return?

D) true or false

15. What is the output of `Arrays.fill(arr, 5)`?

B) All elements set to 5

✅ 20 Interview Questions & Answers:

1. **Q:** How do you sort an array in Java?

   A: Using `Arrays.sort()`.

2. **Q:** Can you print an array directly?

   A: Use `Arrays.toString()` for readable output.

3. **Q:** What is the use of `Arrays.fill()`?

   A: Fills array with a given value.

4. **Q:** What is `binarySearch()` in arrays?

   A: Searches for an element in a **sorted** array.

5. **Q:** What happens if `binarySearch()` fails?

   A: It returns a negative number.

6. **Q:** Difference between `equals()` and `deepEquals()`?

   A: `equals()` is for 1D; `deepEquals()` for nested arrays.

7. **Q:** Can you copy a part of the array?

   A: Yes, using `copyOfRange()`.

8. **Q:** Does `Arrays.copyOf()` allow resizing?

   A: Yes, creates new array of given size.

9. **Q:** How to compare two object arrays?

   A: Use `Arrays.equals()` or loop.

10. **Q:** When should `parallelSort()` be used?

    A: For large data arrays; it's multi-threaded.

11. **Q:** Can arrays store different data types?

    A: No, only same type.

12. **Q:** What is default value in array copy?

    A: `0`, `false`, or `null` based on type.

13. **Q:** How do you handle missing values in arrays?

    A: Use `fill()` with default data.

14. **Q:** Can you clone an array?

    A: Yes, with `array.clone()`.

15. **Q:** Is `Arrays.sort()` stable?

    A: Yes, uses Dual-Pivot Quicksort for primitives.

16. **Q:** What is the time complexity of `Arrays.sort()`?

    A: O(n log n)

17. **Q:** Can I sort strings with `Arrays.sort()`?

    A: Yes.

18. **Q:** How do you merge two arrays?

    A: Use copy loops or `System.arraycopy()`.

19. **Q:** Can you create utility methods for arrays?
    A: Yes, via helper functions.

20. **Q:** Are array utility methods overloaded?
    A: Yes, for various primitive and object types.

✅ Topic Outcome:

After this topic, learners will:

- Perform advanced operations on arrays using `java.util.Arrays`
- Understand how to sort, search, fill, copy, and compare arrays
- Use arrays effectively in real-world applications like reports, filters, etc.

✅ Summary:

- Java arrays, though basic structures, gain significant functionality using the `Arrays` class.
- Whether you're building dashboards or handling large datasets, operations like sort, fill, and copyOf make arrays powerful and flexible.

🔹 Topic 15: Enum (Enumeration) in Java

✅ Definition:

- An `enum` (short for *enumeration*) in Java is a special data type that enables a variable to be a set of predefined constants.
- It improves type safety, readability, and helps represent fixed sets of values like days, colors, directions, etc.
- Java enums are more powerful than simple C/C++ enums — they are full-fledged classes that can have fields, methods, and constructors.

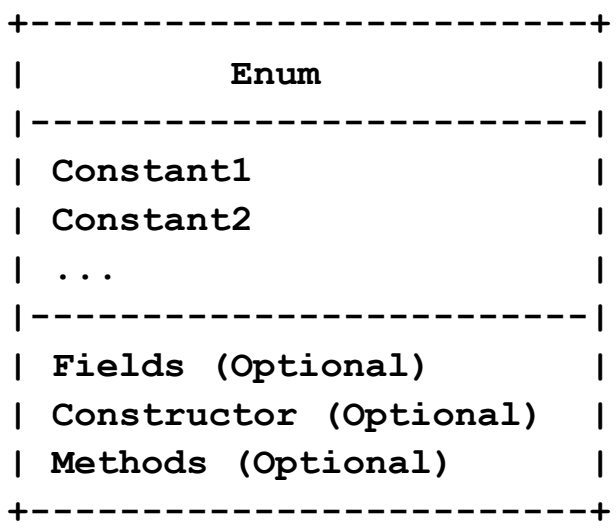
✅ Use Case:

Scenario	Enum Used For
Days of the week	Enum to represent MONDAY, TUESDAY…
Traffic Light System	RED, YELLOW, GREEN
Status Management	PENDING, APPROVED, REJECTED
Order Types in E-Commerce	ONLINE, COD, PICKUP
Access Levels in Software	ADMIN, MODERATOR, USER

✅ Real-Time Usage:

Application	Enum Example
Ticket Booking App	SeatClass { ECONOMY, BUSINESS, FIRST }
Banking System	TransactionType { CREDIT, DEBIT }
University ERP	Grade { A, B, C, D, F }
Online Food App	FoodCategory { VEG, NONVEG, VEGAN }
HR Management System	Gender { MALE, FEMALE, OTHER }

✅ Architecture Diagram: Enum Conceptual Design



Example: Enum Day
- **MONDAY, TUESDAY...SUNDAY**
- **field: isWeekend**
- **method: getIsWeekend()**

✅ Syntax:

Basic Enum Declaration:

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY
}
```

Enum with Fields and Methods:

```
enum Status {
    SUCCESS(200), ERROR(500);

    private int code;

    Status(int code) {
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

✅ Keywords:

`enum`, `values()`, `ordinal()`, `name()`, `switch`, `constructor`, `constant`

✅ Simple Example Code:

```
enum Color {
    RED, GREEN, BLUE
}

public class EnumExample {
    public static void main(String[] args) {
        Color c = Color.RED;
        System.out.println(c); // RED
    }
}
```

✅ Detailed Example Code with Real-Time Usage:

```
enum OrderStatus {
    PLACED, SHIPPED, DELIVERED, CANCELLED
}

public class Order {
    OrderStatus status;

    public Order(OrderStatus status) {
        this.status = status;
    }

    public void checkStatus() {
        switch (status) {
            case PLACED -> System.out.println("Order has
been placed.");
            case SHIPPED -> System.out.println("Order is on
the way.");
            case DELIVERED -> System.out.println("Order
```

```
delivered.");
            case CANCELLED -> System.out.println("Order
cancelled.");
        }
    }

    public static void main(String[] args) {
        Order o = new Order(OrderStatus.SHIPPED);
        o.checkStatus();
    }
}
```

✅ Sub-Topics with Examples:

🔹 `values()` method:

```
for (Day d : Day.values()) {
    System.out.println(d);
}
```

🔹 `ordinal()` method:

```
System.out.println(Day.MONDAY.ordinal()); // 0
```

🔹 `valueOf()` method:

```
Day d = Day.valueOf("SUNDAY");
```

🔹 **Enums in `switch`:**

```
switch (Color.RED) {
    case RED -> System.out.println("It's Red");
}
```

✅ Real-Time Example Code:

```
enum Priority {
    LOW, MEDIUM, HIGH;
}

public class Task {
    public static void main(String[] args) {
        Priority p = Priority.HIGH;
        if (p == Priority.HIGH) {
            System.out.println("Handle immediately!");
        }
    }
}
```

✅ 15 MCQ Questions (with Answers):

1. What is an enum in Java?

   A) A special class with fixed constant values

2. What is the default method provided in enums?

   B) values()

3. What is the return type of `ordinal()`?

C) int

4. Which method returns name of enum constant?

B) name()

5. Can enums have constructors in Java?

A) Yes

6. Can enums implement interfaces?

B) Yes

7. What is the output of `Color.RED.ordinal()`?

C) 0

8. What happens if enum is used in switch?

D) All cases must match enum constants

9. Can enums extend a class?

D) No

10. What type of class is enum in Java?

A) Final

11. Can we override methods in enums?

B) Yes

12. How to iterate over enum?

C) Using values()

13. What is returned by `valueOf("MONDAY")`?

B) Enum constant MONDAY

14. Can enum constants have fields?

A) Yes

15. Which is not valid enum feature?

D) Extending another class

✅ 20 Interview Questions & Answers:

1. **Q:** What is enum in Java?

   A: It's a special class representing constant sets of values.

2. **Q:** Why use enums?

   A: For type safety, readability, and code maintainability.

3. **Q:** Can enums have methods?

   A: Yes, both instance and static methods.

4. **Q:** Can enums be used in switch statements?

   A: Yes.

5. **Q:** Are enums classes?

   A: Yes, implicitly final and extend `java.lang.Enum`.

6. **Q:** What is `values()` method?

   A: Returns an array of enum constants.

7. **Q:** What is `ordinal()`?

   A: Returns position/index of enum constant.

8. **Q:** Difference between `name()` and `toString()` in enums?

   A: Both return name, but `toString()` can be overridden.

9. **Q:** Can enums implement interfaces?

   A: Yes.

10. **Q:** Can we define constructor in enums?

    A: Yes, but it must be private.

11. **Q:** Can enum constants override methods?

    A: Yes, individually.

12. **Q:** Can enums be nested?

    A: Yes, static nested inside classes.

13. **Q:** Are enum constants static or instance?

    A: They are static final.

14. **Q:** What is `EnumSet`?

    A: A high-performance Set for enum types.

15. **Q:** What happens if duplicate constant names exist?

    A: Compile-time error.

16. **Q:** Can enum constants be null?

    A: Yes, but risky.

17. **Q:** How do you get all constants of an enum?

    A: Use `.values()`.

18. **Q:** How to compare enums?

    A: Using `==` or `.equals()`.

19. **Q:** Can enums be serialized?

    A: Yes.

20. **Q:** Can enums be used in annotations?

    A: Yes, common in `@Retention`, `@Target`, etc.

✅ Topic Outcome:

After this topic, learners will:

- Understand how to declare, use, and manipulate enums
- Create readable and maintainable constants in applications
- Integrate enums in switch cases and method calls
- Use enums to enforce type-safe logic

✅ Summary:

- Java Enums provide a clean and structured way to define constant values.
- With capabilities like fields, methods, and switch compatibility, enums are a powerful feature for creating expressive and maintainable Java programs.

🔹 Topic 16: String, String Literal, String Object

✅ Definition:

- A **String** in Java is an object that represents a sequence of characters.
- It is one of the most widely used classes in Java, defined in the `java.lang` package.
- Strings are **immutable**, meaning once a String object is created, it cannot be changed.

In Java, strings can be created in two ways:

1. **String Literal** – Stored in the String Pool (interned).

2. **Using `new` Keyword** – Creates a new object in the heap memory.

✅ Use Case:

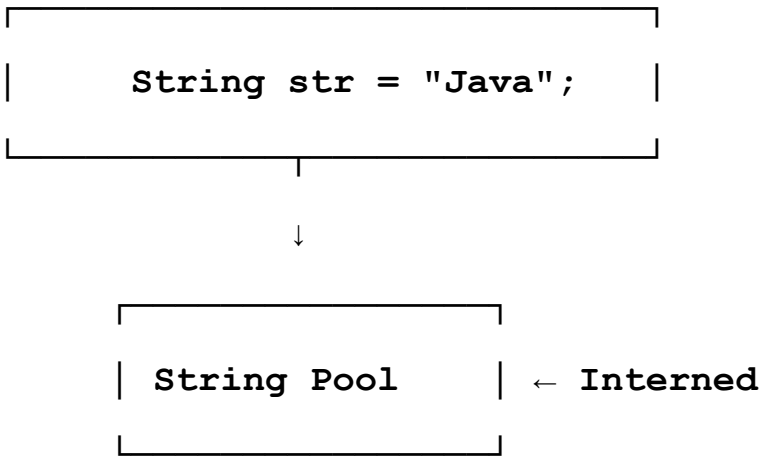
Scenario	String Usage
Usernames/Passwords	Store user input strings
Chat Application	Messages and conversations
Search Engine	Query processing and keyword matching
Banking System	Store account type, name, branch info
Education ERP	Student name, grades, and remarks

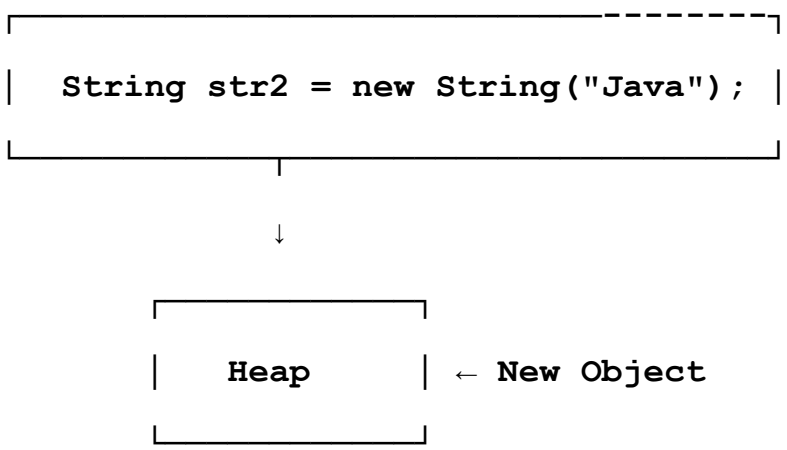
✅ Real-Time Usage:

Application	Real-Time String Use Case
Web Form	Input fields (Name, Email, Password)
Mobile SMS App	Message content and sender/receiver numbers

Online Compiler	Code submitted as String and parsed
Email Service	Subject, body, sender details
Social Media	Post content and usernames

✅ Architecture Diagram: String Creation and Memory Flow





✅ Syntax:

```
String s1 = "Hello";                 // String literal
String s2 = new String("Hello");  // Using new keyword
```

✅ Keywords:

`String`, `literal`, `new`, `immutable`, `pool`, `heap`, `equals`, `==`

✅ Simple Example Code:

```
String name = "Alice";
System.out.println("Hello, " + name);
```

✅ Detailed Example Code with Real-Time Usage:

```
public class WelcomeUser {
    public static void main(String[] args) {
        String username = "admin";

        if (username.equals("admin")) {
            System.out.println("Welcome back,
Admin!");
        } else {
            System.out.println("Access Denied.");
        }
    }
}
```

✅ Sub-Topics with Examples:

- **String Literal:**

```
String s1 = "Hello";
String s2 = "Hello";
System.out.println(s1 == s2); // true (same
reference)
```

- **String Object:**

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2); // false (different
objects)
```

🔹 String Immutability:

```
String s = "Hello";
s.concat(" World");
System.out.println(s); // Output: Hello (original not
changed)
```

✅ Real-Time Example Code:

```
// Check if entered password matches
String correctPassword = "Secret@123";
String inputPassword = "Secret@123";

if (correctPassword.equals(inputPassword)) {
    System.out.println("Login Successful!");
} else {
    System.out.println("Invalid Credentials");
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. Which package contains the String class in Java?

   A) java.util

   B) java.io

   C) java.lang

   D) java.string

   ✅ **Answer: C**

2. What is the output of `"Java" == "Java"`?

A) true

B) false

C) compile error

D) runtime error

✅ **Answer: A**

3. Strings in Java are:

A) Mutable

B) Immutable

C) Static

D) Private

✅ **Answer: B**

4. Which method is used to compare content of strings?

A) equals()

B) ==

C) compare()

D) check()

✅ **Answer: A**

5. Which memory does the string literal use?

A) Heap

B) Stack

C) String Pool

D) Code Segment

✅ **Answer: C**

6. What is returned by `s1 == s2` for `String s1 = new String("Hi"), s2 = new String("Hi")`?

   A) true

   B) false

   C) error

   D) null

   ✅ **Answer: B**

7. Which method returns length of string?

   A) getLength()

   B) length()

   C) size()

   D) count()

   ✅ **Answer: B**

8. Which is the correct way to create a string object?

   A) String str = "Java";

   B) String str = new String("Java");

   C) Both A & B

   D) None

   ✅ **Answer: C**

9. Which operator is used to join strings?

A) &

B) +

C) .

D) %

✅ **Answer: B**

10. Which method checks if two strings are equal ignoring case?

A) equals()

B) isEqual()

C) equalsIgnoreCase()

D) match()

✅ **Answer: C**

11. Which method returns a character at given index?

A) charAt()

B) getChar()

C) index()

D) at()

✅ **Answer: A**

12. Can string be null?

A) Yes

B) No

✅ **Answer: A**

13. What is the output of `s.toUpperCase()` if `s = "java"`?

A) JAVA

B) Java

C) java

D) Compile error

✅ **Answer: A**

14. What happens when you use `==` on string objects?

A) Compares value

B) Compares references

✅ **Answer: B**

15. String class is:

A) abstract

B) final

C) static

D) interface

✅ **Answer: B**

✅ 20 Interview Questions & Answers:

1. **Q:** What is a String in Java?
   A: An immutable sequence of characters stored as an object.

2. **Q:** Difference between String literal and new String()?
   A: Literal is stored in the String pool; new uses heap memory.

3. **Q:** Why are Strings immutable?
   A: For performance, caching, thread safety, and security.

4. **Q:** What is String Pool?
   A: A special memory region for storing unique string literals.

5. **Q:** How to compare content of two strings?
   A: Use `equals()` method.

6. **Q:** Difference between `==` and `equals()`?
   A: `==` compares references; `equals()` compares content.

7. **Q:** Can you modify a String object?
   A: No, but you can create a new modified one.

8. **Q:** How to convert string to uppercase?
   A: Use `toUpperCase()`.

9. **Q:** How to find length of a string?

   A: Use `length()` method.

10. **Q:** Is String thread-safe?

    A: Yes, because it's immutable.

11. **Q:** What is interning in Strings?

    A: Reusing string objects from the String pool.

12. **Q:** Is `String` final?

    A: Yes, so it can't be extended.

13. **Q:** How to check if string contains another string?

    A: Use `contains()`.

14. **Q:** How to replace part of a string?

    A: Use `replace()` method.

15. **Q:** Can String be null?

    A: Yes, but calling a method on it causes `NullPointerException`.

16. **Q:** What is difference between `substring()` and `subSequence()`?

    A: Both extract parts of a string; `subSequence()` returns

`CharSequence`.

17. **Q:** What is immutability benefit?
    A: Ensures strings are safe for concurrent use and caching.

18. **Q:** How do you concatenate two strings?
    A: Use `+` or `concat()`.

19. **Q:** What happens if you use `==` with literals?
    A: May return true due to interning.

20. **Q:** Can you create multiline string in Java?
    A: Yes, with `"""` (text blocks in Java 13+).

✅ Topic Outcome:

After this topic, learners will:

- Understand different ways of creating strings
- Know the difference between string literal and object
- Use string methods for real-world manipulation
- Handle comparisons and avoid memory pitfalls

✅ Summary:

- Strings are fundamental in Java and appear in nearly all real-world applications.
- Knowing how they are stored, created, and manipulated makes it easier to write secure, efficient, and clean code.

🔹 Topic 17: String Methods in Java

✅ Definition:

- In Java, the `String` class provides a rich set of built-in **methods** to manipulate, analyze, and transform strings.
- Since strings are **immutable**, every method returns a **new String object**, not a modified version of the original.
- String methods handle everything from **measuring length**, **extracting substrings**, and **comparing values**, to **replacing text**, **formatting**, and **splitting strings**.

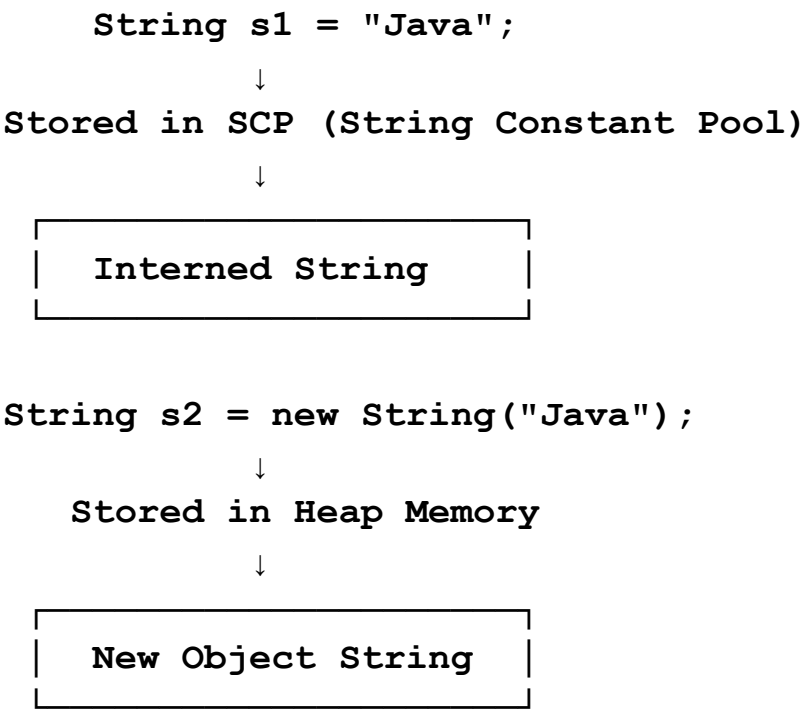
✅ Use Case:

Use Case	String Method(s)
Measure character count	`length()`
Get first or last letter	`charAt()`
Extract name from full name	`substring()`
Convert text case for consistency	`toUpperCase()`, `toLowerCase()`
Clean whitespace in form inputs	`trim()`
Replace symbols or typos	`replace()`
Check for specific keyword	`contains()`
Split CSV records	`split()`

✅ Real-Time Usage:

Application	Real-Time Usage of String Methods
Form Validation	`trim()`, `isEmpty()`, `equalsIgnoreCase()`
Chat App	`toLowerCase()`, `contains()`, `replace()`
Search Feature	`indexOf()`, `substring()`
Data Parsing (CSV/JSON)	`split()`, `replace()`, `trim()`
Authentication System	`equals()`, `equalsIgnoreCase()`

✅ Architecture Diagram: String Pool vs Heap



✅ Syntax & Individual Method Examples:

🔹 1. `length()`

Returns number of characters in the string.

```
String name = "Alice";
System.out.println(name.length()); // Output: 5
```

🔹 2. `charAt(int index)`

Returns character at given index.

```
String s = "Java";
System.out.println(s.charAt(2)); // Output: 'v'
```

🔹 3. `substring(int start, int end)`

Returns substring from start to end-1.

```
String s = "Microcollege";
System.out.println(s.substring(5, 9)); // Output:
"coll"
```

🔹 4. `toUpperCase()`

Converts to uppercase.

```
String s = "hello";
System.out.println(s.toUpperCase()); // Output: "HELLO"
```

🔹 5. `toLowerCase()`

Converts to lowercase.

```
String s = "WELCOME";
System.out.println(s.toLowerCase()); // Output:
"welcome"
```

🔹 6. `trim()`

Removes leading and trailing whitespace.

```
String s = "  hello  ";
System.out.println(s.trim()); // Output: "hello"
```

🔹 7. `replace(String old, String new)`

Replaces all occurrences of a substring.

```
String s = "data123data";
System.out.println(s.replace("data", "info")); //
Output: "info123info"
```

🔹 8. `equals(String another)`

Compares content (case-sensitive).

```
String a = "Test";
String b = "Test";
System.out.println(a.equals(b)); // true
```

🔹 9. `equalsIgnoreCase(String another)`

Compares content (case-insensitive).

```
String a = "Admin";
String b = "admin";
System.out.println(a.equalsIgnoreCase(b)); // true
```

🔹 10. `contains(CharSequence s)`

Checks if string contains a substring.

```
String email = "user@college.com";
System.out.println(email.contains("@")); // true
```

🔹 11. `indexOf(String s)`

Returns index of first occurrence.

```
String s = "Engineering";
System.out.println(s.indexOf("e")); // Output: 1
```

🔹 12. `isEmpty()`

Checks if string is empty (`length == 0`).

```
String s = "";
System.out.println(s.isEmpty()); // true
```

🔹 13. `split(String regex)`

Splits string into array using delimiter.

```
String csv = "Alice,Bob,Charlie";
String[] names = csv.split(",");
System.out.println(names[1]); // Output: Bob
```

✅ Keywords:

`String`, `length`, `charAt`, `substring`, `equals`, `replace`, `toUpperCase`, `SCP`, `Heap`, `split`

✅ Real-Time Example Code:

```
public class FormSanitizer {
    public static void main(String[] args) {
        String rawInput = "  Admin@Example.COM  ";
        String email = rawInput.trim().toLowerCase();

        if (email.contains("@") && email.endsWith(".com"))
{
            System.out.println("Valid Email: " + email);
        } else {
            System.out.println("Invalid Email");
        }
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. What does `charAt(3)` return in `"India"`?

   A) a

   B) i

   C) d

   D) I

   ✅ **Answer: A**

2. `"Java".equals("JAVA")` returns?

   A) true

   B) false

   ✅ **Answer: B**

3. `substring(1, 4)` of `"coding"`?

   A) "cod"

   B) "odi"

   C) "din"

   D) "ing"

   ✅ **Answer: B**

4. Which method splits a string?

   A) split()

   B) divide()

   C) break()

   D) separate()

✅ **Answer: A**

5. What will `replace("a", "o")` do in `"Data"`?

A) Doto

B) Dato

C) Doto

D) Dotoa

✅ **Answer: B**

6. What does `equalsIgnoreCase()` compare?

A) Length

B) Content (case-insensitive)

C) Hashcode

D) References

✅ **Answer: B**

7. `toUpperCase()` applied on `"java"`?

A) JAVA

B) java

✅ **Answer: A**

8. `isEmpty()` returns true if?

A) null string

B) only spaces

C) ""

D) "abc"

✅ **Answer: C**

9. What is the return type of `split()`?

A) List

B) Array

✅ **Answer: B**

10. Which method trims whitespace?

A) strip()

B) clean()

C) trim()

✅ **Answer: C**

11. What does `indexOf("a")` return for `"Java"`?

A) 0

B) 1

C) 3

✅ **Answer: 1**

12. `"Test".contains("t")` returns?

A) true

B) false

✅ **Answer: B**

13. What does `length()` return for `""`?

A) 0

✅ **Answer: A**

14. `"Java".equals("Java")` returns?

A) true

✅ **Answer: A**

15. What does `new String("Test")` store in?

A) Stack

B) Heap

✅ **Answer: B**

✅ 20 Interview Q&A:

1. **Q:** What does `length()` return?

   A: Number of characters in the string.

2. **Q:** How to get a single character?

   A: Use `charAt(index)`.

3. **Q:** What's the difference between `==` and `equals()`?

   A: `==` compares memory, `equals()` compares content.

4. **Q:** How do you extract a portion of string?

   A: Use `substring(start, end)`.

5. **Q:** What is SCP in Java?

   A: String Constant Pool – shared memory for literals.

6. **Q:** What is heap memory?

   A: Stores objects created via `new`.

7. **Q:** What's the output of `"abc".toUpperCase()`?

   A: "ABC"

8. **Q:** Is `replace()` destructive?

   A: No, it returns a new String.

9. **Q:** Can you compare strings ignoring case?

   A: Yes, using `equalsIgnoreCase()`.

10. **Q:** How to check if a string is empty?

    A: Use `isEmpty()`.

11. **Q:** Can `split()` handle CSVs?

    A: Yes, it splits by delimiter.

12. **Q:** What happens if `indexOf()` doesn't find a value?

    A: Returns -1.

13. **Q:** How to validate email format quickly?

    A: Use `contains("@")` and `endsWith(".com")`.

14. **Q:** How to clean extra spaces in form input?

    A: Use `trim()`.

15. **Q:** Are string objects immutable?

    A: Yes.

16. **Q:** Is `substring()` zero-based?

    A: Yes.

17. **Q:** How are two literals with same value handled?

    A: Reused from SCP.

18. **Q:** Which method splits string into words?

    A: `split(" ")`.

19. **Q:** How to check if string contains digits?

    A: Use `matches("[0-9]+")`.

20. **Q:** How to find first index of a letter?

    A: Use `indexOf()`.

✅ Topic Outcome:

Learners will:

- Use every major `String` method with confidence.
- Understand the memory difference between SCP and heap.
- Apply string logic in validation, parsing, formatting.
- Improve performance and code readability using correct methods.

✅ Summary:

- Java's `String` class methods are powerful tools for data formatting, user input processing, and content analysis.
- By mastering each method and its memory behavior (SCP vs Heap), developers can write cleaner, safer, and more effective Java applications.

🔹 Topic 18: StringBuffer, StringBuilder, and StringTokenizer in Java

---

✅ Definition:

Java provides three important classes for working with strings that require **modification**:

- **`StringBuffer`**: A mutable, thread-safe sequence of characters.
- **`StringBuilder`**: A mutable, non-thread-safe version of `StringBuffer`—faster in single-threaded contexts.
- **`StringTokenizer`**: A legacy class used for splitting a string into tokens based on delimiters.

These classes help improve performance when working with large or frequently modified strings.

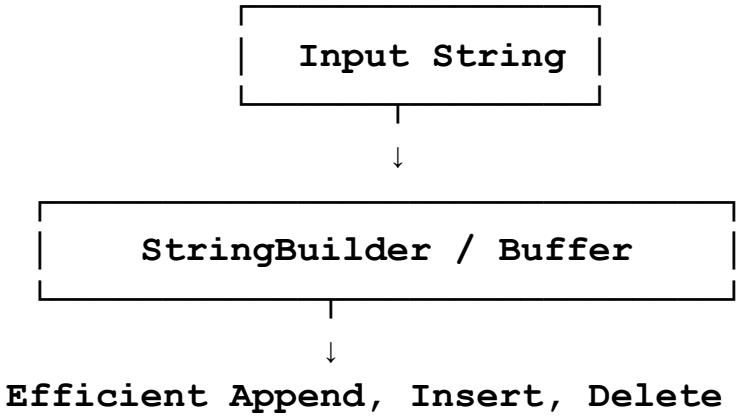
✅ Use Case:

Use Case	Class Used
Build dynamic strings in loops	`StringBuilder`
Append log messages with thread safety	`StringBuffer`
Tokenize CSV/space-delimited text	`StringTokenizer`

✅ Real-Time Usage:

Application	Real-Time Use of Buffer/Builder/Tokenizer
Logging Systems	Use `StringBuffer` to safely append logs
Chat Applications	Use `StringBuilder` to build messages
CSV Parsers	Use `StringTokenizer` to break lines
HTML Code Builders	Append HTML tags using `StringBuilder`

✅ Architecture Diagram: Mutable String Handling



↓

Final Output String

✅ Syntax & Examples

🔹 1. `StringBuffer` (Thread-safe)

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

🔹 2. `StringBuilder` (Faster, Non-thread-safe)

```
StringBuilder sb = new StringBuilder("Java");
sb.insert(4, "Script");
System.out.println(sb); // JavaScript
```

🔹 3. `StringTokenizer` (Legacy Token Splitter)

```
import java.util.StringTokenizer;

StringTokenizer st = new StringTokenizer("Zeno,
Blade,Victor", ",");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

✅ Comparison Table:

Feature	String	StringBuffer	StringBuilder
Mutable	❌ No	✅ Yes	✅ Yes
Thread-safe	✅ (Immutable)	✅ Yes	❌ No
Performance	Low (for loops)	Medium	High
Use Case	Fixed strings	Multi-threaded ops	Fast string building

---

✅ Keywords:

`StringBuffer`, `StringBuilder`, `StringTokenizer`, `mutable`, `append`, `insert`, `delete`, `token`, `thread-safe`

✅ Real-Time Example Code:

```
public class LogGenerator {
    public static void main(String[] args) {
        StringBuffer log = new StringBuffer();
        for (int i = 1; i <= 3; i++) {
            log.append("Log Entry
").append(i).append("\n");
        }
        System.out.println(log);
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. Which class is thread-safe?

   A) StringBuilder

   B) StringBuffer

   C) StringTokenizer

   D) String

   ✅ **Answer: B**

2. Which class is faster in single-threaded context?

   A) StringBuffer

   B) String

   C) StringBuilder

   ✅ **Answer: C**

3. `StringTokenizer` splits strings based on:

   A) index

   B) delimiter

   ✅ **Answer: B**

4. What is the output of: `new StringBuilder("Hi").append("All")`?

   A) HiAll

   ✅ **Answer: A**

5. Which class is legacy?

A) String

B) StringTokenizer

✅ **Answer: B**

6. Which method adds text to the end of a builder?

A) add()

B) insert()

C) append()

✅ **Answer: C**

7. What is the result of inserting at index 3 in `"Good"`?

A) GGood

B) GooD

C) GooXYZd

✅ **Answer: C**

8. StringBuffer is used when:

A) Memory is less

B) Concurrency is required

✅ **Answer: B**

9. Which method deletes characters in builder?

A) remove()

B) delete(start, end)

✅ **Answer: B**

10. `StringBuilder` is part of which package?

A) java.util

B) java.lang

✅ **Answer: B**

11. Can `StringBuilder` be converted to `String`?

A) No

B) Yes, via `toString()`

✅ **Answer: B**

12. Is `StringTokenizer` recommended in Java 8+?

A) Yes

B) No, prefer `split()`

✅ **Answer: B**

13. What's the default delimiter of `StringTokenizer`?

A) Comma

B) Whitespace

✅ **Answer: B**

14. Which is preferred for multiple appends in loop?

A) String

B) StringBuffer

✅ **Answer: B**

15. Can you reverse a StringBuilder?

A) No

B) Yes, using `reverse()`

✅ **Answer: B**

✅ 20 Interview Questions & Answers:

1. **Q:** What is the difference between `String` and `StringBuilder`?

   A: String is immutable; StringBuilder is mutable and faster.

2. **Q:** When do you use `StringBuffer`?

   A: When thread safety is required.

3. **Q:** Why is `StringBuilder` faster than `StringBuffer`?

   A: Because it is not synchronized.

4. **Q:** How to convert `StringBuilder` to `String`?

   A: Use `.toString()`.

5. **Q:** What does `.insert()` do?

   A: Inserts characters at a specified position.

6. **Q:** Is `StringTokenizer` still used?

   A: It's legacy; prefer `split()` in modern Java.

7. **Q:** Which is preferred for log generation?

   A: `StringBuffer`.

8. **Q:** What is the use of `reverse()` method?

   A: Reverses characters in builder/buffer.

9. **Q:** Is `StringBuilder` part of Java 1.5?

   A: Yes, introduced in Java 5.

10. **Q:** Can multiple threads use `StringBuilder` safely?

    A: No, it’s not thread-safe.

11. **Q:** Difference between `append()` and `insert()`?

    A: `append()` adds to end; `insert()` adds at given index.

12. **Q:** Can you chain methods in `StringBuilder`?

    A: Yes, due to method chaining.

13. **Q:** Is memory reused in `StringBuilder`?

    A: Yes, it grows dynamically.

14. **Q:** What's the initial capacity of builder?

    A: 16 characters by default.

15. **Q:** Which one is synchronized: Buffer or Builder?

    A: Buffer.

16. **Q:** How to split string with comma?

    A: `String.split(",")` or `StringTokenizer`.

17. **Q:** Can `delete()` be chained with `append()`?

    A: Yes.

18. **Q:** What happens if insert index is invalid?

    A: Throws `StringIndexOutOfBoundsException`.

19. **Q:** Is `StringBuffer` slower than builder?

    A: Yes, due to synchronization.

20. **Q:** Why not use `String` for loops?

    A: Because strings are immutable, new objects are created every time.

✅ Topic Outcome:

After this topic, learners will:

- Differentiate between mutable and immutable string classes.
- Use `StringBuffer` and `StringBuilder` for performance-intensive tasks.
- Apply `StringTokenizer` for legacy parsing.
- Choose the right class based on thread-safety needs and efficiency.

✅ Summary:

- `StringBuffer`, `StringBuilder`, and `StringTokenizer` offer powerful alternatives to `String` when performance, mutability, and flexibility are needed.
- Mastery of these classes is essential for efficient Java string manipulation, especially in real-time, multi-threaded, or large-scale text-processing systems.

🔹 Topic 19: `static` and `non-static` Members in Java

✅ Definition:

In Java, class members (variables and methods) can be either **static** (class-level) or **non-static** (instance-level):

- A **static member** belongs to the class and is shared among all instances.

- A **non-static member** is specific to each object instance and accessed through that object.

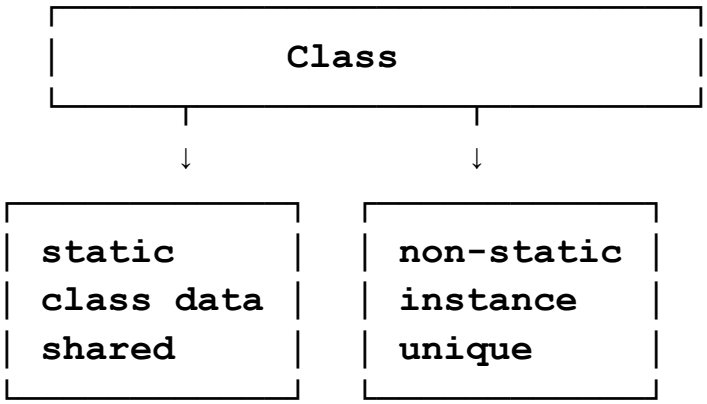
✅ Use Case:

Scenario	Member Type
Common property for all objects (e.g., `college`)	`static`
Unique property for each object (e.g., `studentName`)	`non-static`
Utility/helper methods (e.g., `Math.sqrt()`)	`static method`
Access data that changes per object	`non-static field`

✅ Real-Time Usage:

Application	Usage of `static` / `non-static`
School Management System	`schoolName` as static, `studentName` as non-static
Bank Application	`bankCode` as static, `accountBalance` as non-static
Utility Class	`calculateInterest()` as static
Counter Program	`count` as static to track total objects

✅ Architecture Diagram:



- Static members: loaded into memory once, shared across all instances.
- Non-static members: exist per instance and differ from object to object.

✅ Syntax & Examples:

🔹 Declaring `static` and `non-static` members:

```
public class Student {
    static String college = "Micro College";  // static
variable
    String name;                              // non-static
variable

    static void showCollege() {
        System.out.println(college);
    }

    void showName() {
        System.out.println(name);
    }
}
```

✅ Individual Example Explanations:

1. Static variable – shared across instances

```
Student.college = "KGCAS"; // All students now share
this college
```

2. Non-static variable – unique per object

```
Student s1 = new Student();
s1.name = "Raj";

Student s2 = new Student();
s2.name = "Kumar";

System.out.println(s1.name); // Raj
System.out.println(s2.name); // Kumar
```

3. Static method – called using class name

```
Student.showCollege(); // "KGCAS"
```

4. Non-static method – called via object

```
s1.showName(); // Raj
```

✅ Keywords:

`static`, `non-static`, `class`, `object`, `shared`, `instance`, `this`, `static block`

✅ Real-Time Example Code:

```
public class BankAccount {
    static String bankName = "Global Bank"; // static
    int accountNumber;                      // non-static

    BankAccount(int accNum) {
        this.accountNumber = accNum;
    }

    void displayDetails() {
        System.out.println("Bank: " + bankName);
        System.out.println("Account No: " + accountNumber);
    }

    static void displayBank() {
        System.out.println("Bank Name: " + bankName);
    }

    public static void main(String[] args) {
        BankAccount.displayBank();

        BankAccount a1 = new BankAccount(123);
        BankAccount a2 = new BankAccount(456);

        a1.displayDetails();
        a2.displayDetails();
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. Which keyword makes a method class-level?

   A) shared

   B) static

   C) public

   D) instance

   ✅ **Answer: B**

2. Which members require an object to access?

   A) static

   B) non-static

   ✅ **Answer: B**

3. What is the output of accessing static variable via object?

   A) Error

   B) Works but not recommended

   ✅ **Answer: B**

4. Can we use `this` keyword in static methods?

   A) Yes

   B) No

   ✅ **Answer: B**

5. What is shared among all objects?

   A) non-static variables

   B) static variables

✅ **Answer: B**

6. Can static methods access non-static data directly?

A) Yes

B) No

✅ **Answer: B**

7. What happens if you access non-static method via class?

A) Works

B) Compilation error

✅ **Answer: B**

8. Which variable is loaded during class loading?

A) non-static

B) static

✅ **Answer: B**

9. Is `main()` method static?

A) Yes

✅ **Answer: A**

10. Can constructors be static?

A) Yes

B) No

✅ **Answer: B**

11. Which block executes once when class is loaded?

A) static block

✅ **Answer: A**

12. What will `new Student().college` print if static?

A) Same value across all objects

✅ **Answer: A**

13. Static members belong to:

A) object

B) class

✅ **Answer: B**

14. Which can call both static and non-static members?

A) static method

B) non-static method

✅ **Answer: B**

15. What does JVM call first?

A) constructor

B) static main()

✅ **Answer: B**

✅ 20 Interview Questions & Answers:

1. **Q:** What is a static member in Java?

   A: A class-level variable/method shared across all instances.

2. **Q:** Can static methods access non-static fields?

   A: No, only after object instantiation.

3. **Q:** When should you use static methods?

   A: When functionality doesn't depend on instance data.

4. **Q:** Is `main()` method static? Why?

   A: Yes, so JVM can call it without creating an object.

5. **Q:** Can static methods be overloaded?

   A: Yes.

6. **Q:** Can we override static methods?

   A: No, they are hidden, not overridden.

7. **Q:** What is a static block?

   A: A block that runs once when the class is loaded.

8. **Q:** Can a static method use `this` keyword?

   A: No.

9. **Q:** What is a non-static member?

A: Instance-level property unique per object.

10. **Q:** How are static variables initialized?

A: During class loading.

11. **Q:** Can we have both static and non-static versions of a method?

A: Yes, method overloading supports it.

12. **Q:** Is memory reused for static variables?

A: Yes, only one copy is maintained.

13. **Q:** When is static block executed?

A: Before main, during class loading.

14. **Q:** Can static variables be private?

A: Yes.

15. **Q:** Are static variables part of the object?

A: No, they belong to the class.

16. **Q:** Can abstract methods be static?

A: No.

17. **Q:** Can we access static members from object?

    A: Yes, but not recommended.

18. **Q:** What if two objects modify static variable?

    A: The last change reflects across all.

19. **Q:** Is it possible to override static methods?

    A: No, only method hiding is allowed.

20. **Q:** What is the use of static imports?

    A: Access static members without class name prefix.

✅ Topic Outcome:

By the end of this topic, learners will:

- Differentiate between static and non-static variables/methods.
- Know when and how to use class-level versus instance-level logic.
- Write cleaner utility-based and object-based code using appropriate member types.

✅ Summary:

- `static` and `non-static` members help control **scope, memory, and access** in Java.
- Understanding their behavior is essential for building **efficient**, **thread-safe**, and **maintainable** object-oriented systems.

🔹 Topic 20: Access Modifiers and Non-Access Modifiers in Java

---

✅ Definition:

In Java, **modifiers** are keywords that set the **accessibility**, **behavior**, or **nature** of classes, methods, variables, and constructors. They are categorized into:

1. **Access Modifiers**: Control **visibility** of members (e.g., `public`, `private`, `protected`, default).

2. **Non-Access Modifiers**: Control **properties** like `final`, `static`, `abstract`, `synchronized`, `transient`, `volatile`, etc.

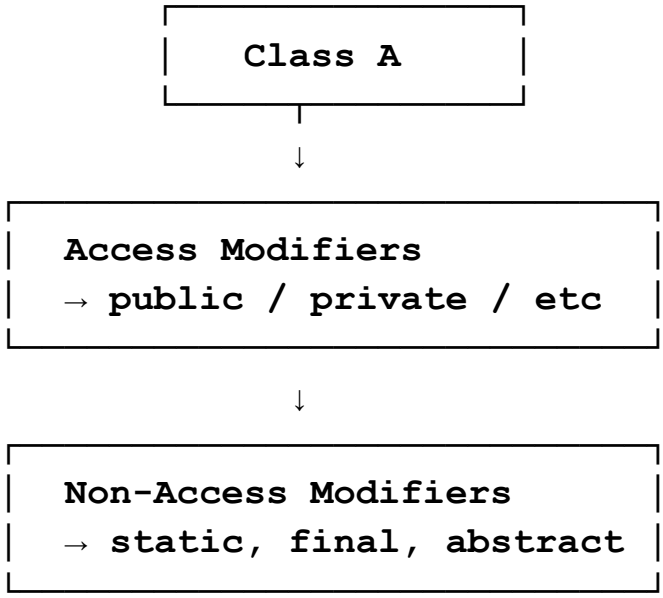
✅ Use Case:

Modifier Type	Purpose	Example
`public`	Access from anywhere	API methods
`private`	Access only within the class	Encapsulation of sensitive data
`protected`	Access within package and subclasses	Inheritance with controlled access
`static`	Belongs to class, not instance	Shared configurations
`final`	Prevents modification or inheritance	Constant fields, secure classes
`abstract`	Declares abstract class/method	Base class for inheritance only
`synchronized`	Ensures thread safety	Multithreaded banking systems

✅ Real-Time Usage:

Scenario	Modifier Used
Utility methods like `Math.pow()`	`public static`
Secure password variable	`private final`
Reusable shape base class	`abstract`
Safe withdrawal method	`synchronized`

✅ Architecture Diagram: Modifier Control in a Class



✅ Syntax & Examples

🔹 Access Modifiers

```
public class Example {
    public int a = 5;
    private int b = 10;
    protected int c = 20;
    int d = 25; // default (package-private)

    public void show() {
        System.out.println(a + b + c + d);
    }
}
```

🔹 Non-Access Modifiers

```
final class Vehicle { } // Cannot be inherited

abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Circle drawn");
    }
}

static class Helper {
    static void print() {
        System.out.println("Utility print");
    }
}
```

✅ Access Modifiers Table:

Modifier	Same Class	Same Package	Subclass	Other Packages
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
(default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

✅ Keywords:

`public`, `private`, `protected`, `default`, `static`, `final`, `abstract`, `synchronized`, `volatile`, `transient`, `native`, `strictfp`

✅ Real-Time Example Code:

```
public class User {
    private final String username; // cannot change once assigned

    public User(String name) {
        this.username = name;
    }

    public String getUsername() {
        return username;
    }

    public static void greet() {
        System.out.println("Hello, user!");
    }

    public synchronized void updatePassword() {
        // Thread-safe method
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. Which modifier allows access from anywhere?

   A) private

   B) protected

   C) public

   D) default

   ✅ **Answer: C**

2. Which modifier restricts access to the class only?

   A) protected

   B) public

   C) private

   D) static

   ✅ **Answer: C**

3. Which modifier prevents method override?

   A) static

   B) final

   C) abstract

   ✅ **Answer: B**

4. What is the default access modifier in Java?

   A) protected

   B) package-private

   C) public

✅ **Answer: B**

5. Which one ensures thread-safety?

   A) static

   B) volatile

   C) synchronized

   ✅ **Answer: C**

6. Can a final class be extended?

   A) Yes

   B) No

   ✅ **Answer: B**

7. Which modifier is used for abstract methods?

   A) final

   B) abstract

   C) native

   ✅ **Answer: B**

8. What does `transient` do?

   A) Makes variable accessible

   B) Prevents variable from being serialized

   ✅ **Answer: B**

9. Can static methods use `this` keyword?

   A) Yes

B) No

✅ **Answer: B**

10. What's the scope of protected members?

A) Class only

B) Same package & subclasses

✅ **Answer: B**

11. Which modifier indicates only one copy exists?

A) static

✅ **Answer: A**

12. Which modifier works for strict floating point math?

A) strictfp

✅ **Answer: A**

13. Can you declare a constructor as static?

A) Yes

B) No

✅ **Answer: B**

14. What's the behavior of `volatile`?

A) Makes variable shared and consistent across threads

✅ **Answer: A**

15. What does the `native` keyword mean?

A) Variable from native OS

B) Method implemented in C/C++

✅ **Answer: B**

✅ 20 Interview Questions & Answers:

1. **Q:** What are access modifiers in Java?

   A: Keywords that define the scope of class members.

2. **Q:** What is the default access level?

   A: Package-private (no keyword).

3. **Q:** Difference between `private` and `protected`?

   A: `private`: only within class; `protected`: class + package + subclass.

4. **Q:** Can we override `private` methods?

   A: No.

5. **Q:** Can abstract class have a constructor?

   A: Yes.

6. **Q:** Can `static` methods be overridden?

   A: No, they are hidden.

7. **Q:** Use of `final` keyword?

   A: To prevent method override, variable modification, or inheritance.

8. **Q:** What is `strictfp`?

   A: Ensures consistent floating-point calculations across platforms.

9. **Q:** What is `transient`?

   A: Prevents fields from being serialized.

10. **Q:** Use of `synchronized`?

    A: To ensure thread-safe access to methods.

11. **Q:** What is the purpose of `native`?

    A: Calls a platform-specific method in C/C++.

12. **Q:** Can an abstract method be static?

    A: No.

13. **Q:** What happens if final variable is not initialized?

    A: Compilation error.

14. **Q:** Can a method be both `abstract` and `final`?

    A: No, contradictory modifiers.

15. **Q:** Can an interface have access modifiers?

    A: Yes, but members are `public` by default.

16. **Q:** Can we have static block in class?

    A: Yes.

17. **Q:** Can abstract class have concrete methods?

    A: Yes.

18. **Q:** What does `volatile` ensure?

    A: Latest value of a variable is always visible across threads.

19. **Q:** Difference between `static` and `instance` variable?

    A: Static is class-wide; instance is object-specific.

20. **Q:** Why use access modifiers?

    A: To implement encapsulation and security.

✅ Topic Outcome:

By the end of this topic, learners will:

- Understand visibility and lifecycle of variables and methods.
- Apply the right modifier to enforce encapsulation, immutability, or thread safety.
- Use non-access modifiers to fine-tune class behavior and performance.

✅ Summary:

- Access and non-access modifiers empower developers to write **secure**, **efficient**, and **scalable** Java code.
- Mastery over modifiers ensures **clean object-oriented design**, **thread-safe execution**, and **flexible code reuse**.

🔹 Topic 21: Methods and Its Types in Java

✅ Definition:

- A **method** in Java is a block of code that performs a specific task.
- It is used to define the behavior of objects and can be called multiple times to avoid code duplication.
- Java supports different types of methods based on how they are defined and used, such as instance methods, static methods, constructors, and special methods like getters and setters.

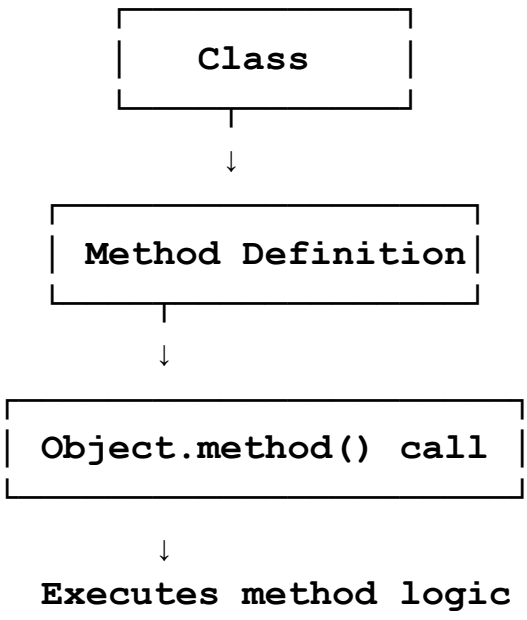
✅ Use Case:

Purpose	Method Type Used
Common logic across objects	`static` method
Custom behavior per object	Instance method
Creating objects	Constructor
Reading/updating private variables	Getter/Setter methods

✅ Real-Time Usage:

Application	Method Use Case Example
ATM System	`withdrawAmount()` instance method
Utility Class	`convertToUpperCase()` static method
Banking App	`calculateInterest()` instance method
Web Form Handling	`setUserName()` setter method

✅ Architecture Diagram: Method Invocation Flow



✅ Syntax & Examples

🔹 Method Syntax

```
returnType methodName(parameters) {
    // method body
    return value;
}
```

✅ Types of Methods with Examples

1. Instance Method

```
public class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

2. Static Method

```
public class MathUtils {
    static int square(int num) {
        return num * num;
    }
}
```

3. Parameterless Method

```
void greet() {
    System.out.println("Hello!");
}
```

4. Parameterized Method

```
void greetUser(String name) {
    System.out.println("Hello, " + name);
}
```

5. Method with Return Type

```
double getAverage(int a, int b) {
    return (a + b) / 2.0;
}
```

6. Getter & Setter

```
private String name;

public String getName() {
    return name;
}

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

7. Constructor (Special Method)

```
public class Student {
    String name;

    Student(String n) {
        name = n;
```

```
    }
}
```

✅ Real-Time Example Code

```
public class Bank {
    private double balance;

    public Bank(double amount) {
        balance = amount;
    }

    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
    }

    public double getBalance() {
        return balance;
    }

    public static void printBankName() {
        System.out.println("Bank: Java National Bank");
    }

    public static void main(String[] args) {
        Bank b = new Bank(1000);
        b.deposit(500);
        System.out.println("Balance: " + b.getBalance());
        printBankName();
    }
}
```

✅ Keywords:

`method`, `return`, `parameters`, `arguments`, `static`, `void`, `this`, `getter`, `setter`, `constructor`, `overloading`

✅ 15 MCQ Questions (with Options & Answers):

1. What does a method return if declared `void`?

   A) `null`

   B) `0`

   C) Nothing

   ✅ **Answer: C**

2. Which method belongs to the class and not object?

   A) Instance method

   B) Static method

   ✅ **Answer: B**

3. Which method automatically runs during object creation?

   A) constructor

   ✅ **Answer: A**

4. Can we overload methods in Java?

   A) Yes

✅ **Answer: A**

5. What is used to access instance variables?

A) this

✅ **Answer: A**

6. Which of these defines a method with no return type?

A) void method()

✅ **Answer: A**

7. Can a method return multiple values?

A) No

B) Yes, using arrays or objects

✅ **Answer: B**

8. What happens if return type is missing?

A) Error

✅ **Answer: A**

9. Which is a special method in a class?

A) main

B) constructor

✅ **Answer: B**

10. Static methods are called using:

A) object

B) class name

✅ **Answer: B**

11. What is `this` used for?

A) Refers to current object

✅ **Answer: A**

12. Are method parameters mandatory?

A) No

✅ **Answer: A**

13. Can you return an object from a method?

A) Yes

✅ **Answer: A**

14. What is the default return type in Java?

A) void

B) There is none

✅ **Answer: B**

15. Can static methods access instance variables?

A) Yes

B) No, unless via object

✅ **Answer: B**

✅ 20 Interview Questions & Answers:

1. **Q:** What is a method in Java?
   A: A block of code that performs a task.

2. **Q:** Difference between static and instance methods?
   A: Static methods belong to the class, instance methods to objects.

3. **Q:** What is method overloading?
   A: Defining multiple methods with same name but different parameters.

4. **Q:** Can we have method inside another method in Java?
   A: No, Java does not support nested methods.

5. **Q:** Why use getter/setter?
   A: To control access to private variables.

6. **Q:** Can methods return objects?
   A: Yes.

7. **Q:** What is the `this` keyword?
   A: Refers to the current object.

8. **Q:** What is a constructor?

   A: A special method that initializes an object.

9. **Q:** Can constructors be overloaded?

   A: Yes.

10. **Q:** What is the return type of constructor?

    A: None.

11. **Q:** What if method has no return value?

    A: Use `void`.

12. **Q:** Can we call non-static methods in static context?

    A: Only using an object.

13. **Q:** Can methods be overloaded by changing return type only?

    A: No.

14. **Q:** What is a recursive method?

    A: A method that calls itself.

15. **Q:** Can we declare a method as `final`?

    A: Yes, to prevent overriding.

16. **Q:** What is method signature?

    A: Method name + parameter list.

17. **Q:** Can we override static methods?

    A: No.

18. **Q:** What is method hiding?

    A: When static methods are redefined in subclass.

19. **Q:** Can we use return without a value in `void` method?

    A: Yes, just `return;`.

20. **Q:** What is the use of `main` method?

    A: JVM calls it to run the program.

✅ Topic Outcome:

By completing this topic, learners will:

- Understand how to define and use methods effectively.
- Differentiate between static and non-static methods.
- Apply method overloading, use constructors, and utilize getter/setters.

✅ Summary:

- Methods define how objects behave and interact. Java methods promote code reusability, flexibility, and encapsulation.
- A strong grasp of method types is foundational to writing clean, maintainable Java code.

🔹 Topic 22: Constructors and Its Types in Java

✅ Definition:

- A **constructor** in Java is a special method that is called **automatically when an object is created**.
- It is used to initialize the object. Unlike regular methods, constructors **do not have a return type**, not even `void`, and they **must have the same name as the class**.

✅ Use Case:

Use Case	Constructor Type Used
Creating an object with default values	Default Constructor

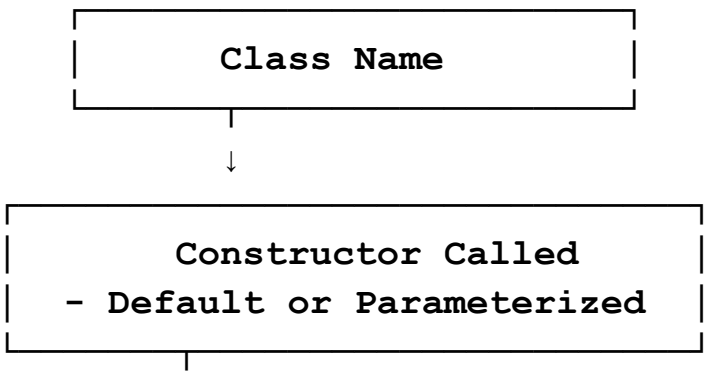
Creating objects with custom values	Parameterized Constructor
Sharing initialization across constructors	Constructor Overloading
Inheriting parent constructor properties	`super()` in Constructor
Reusing constructor in the same class	`this()` Constructor Chaining

---

✅ Real-Time Usage:

Application	Constructor Example
Student Registration	`Student(String name, int age)`
Bank Account Initialization	`BankAccount(String accNo, double balance)`
Game Character Setup	`Character(String name, int health, int mana)`

✅ Architecture Diagram: Constructor Invocation



↓

Object is Created in Heap Memory

↓

Instance Variables Initialized

✅ Syntax & Examples

🔹 1. Default Constructor (No parameters)

```
public class Car {
    Car() {
        System.out.println("Car object created");
    }
}
```

🔹 2. Parameterized Constructor

```
public class Car {
    String model;
    Car(String model) {
        this.model = model;
    }
}
```

🔹 3. Constructor Overloading

```
public class Book {
    String title;
    int pages;

    Book() {
        title = "Unknown";
```

```
        pages = 0;
    }

    Book(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }
}
```

- **4. Using `this()` Constructor Chaining**

```
public class Box {
    int length, width;

    Box() {
        this(10, 10); // calls the other constructor
    }

    Box(int l, int w) {
        length = l;
        width = w;
    }
}
```

- **5. Using `super()` Constructor Call**

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}

class Employee extends Person {
    Employee(String name) {
```

```
        super(name); // calls Person's constructor
    }
}
```

✅ Keywords:

`constructor`, `this()`, `super()`, `overloading`, `initialization`, `new`, `heap`, `object creation`, `parameterized`, `default`

✅ Real-Time Example Code:

```
public class Student {
    String name;
    int age;

    // Default constructor
    Student() {
        name = "Not Assigned";
        age = 0;
    }

    // Parameterized constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    void show() {
        System.out.println("Name: " + name + ", Age: " +
age);
    }
```

```
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Alice", 20);

        s1.show();
        s2.show();
    }
}
```

🔹 Copy Constructor in Java

✅ Definition:

- A **copy constructor** is a special constructor used to **create a new object by copying values from another object** of the same class.
- Java does not provide a default copy constructor, but we can define our own.

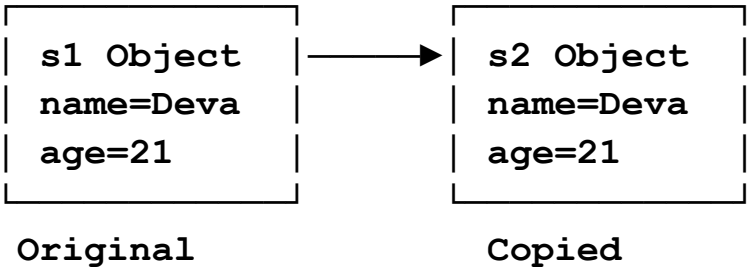
✅ Syntax:

```
ClassName(ClassName obj) {
    // copy field values from obj
}
```

✅ Example:

```
class Student {
    String name;
    int age;

    // Parameterized Constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy Constructor
    Student(Student s) {
        this.name = s.name;
        this.age = s.age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " +
age);
    }

    public static void main(String[] args) {
        Student s1 = new Student("Deva", 21);
        Student s2 = new Student(s1);  // Copy constructor

        s1.display();
        s2.display();
    }
}
```

✅ Diagram:



✅ 15 MCQ Questions (with Options & Answers):

1. What is the return type of a constructor?

   A) void

   B) int

   C) None

   ✅ **Answer: C**

2. Can constructors be overloaded?

   A) Yes

   ✅ **Answer: A**

3. What is the default constructor?

   A) A constructor with no parameters

   ✅ **Answer: A**

4. Which keyword is used to invoke a constructor from another constructor in the same class?

A) super

B) this

✅ **Answer: B**

5. Can we have multiple constructors in a class?

A) Yes

✅ **Answer: A**

6. Can a constructor be static?

A) Yes

B) No

✅ **Answer: B**

7. What happens if no constructor is defined?

A) Compilation error

B) Java provides default constructor

✅ **Answer: B**

8. Can constructor be private?

A) Yes

✅ **Answer: A**

9. What keyword is used to call superclass constructor?

A) base

B) super

✅ **Answer: B**

10. Can constructors be inherited?

A) Yes

B) No

✅ **Answer: B**

11. What is constructor chaining?

A) Linking multiple constructors using `this()`

✅ **Answer: A**

12. Can constructor return a value?

A) Yes

B) No

✅ **Answer: B**

13. When is a constructor called?

A) During object creation

✅ **Answer: A**

14. What is used to create an object in Java?

A) init

B) new

✅ **Answer: B**

15. What happens if `super()` is not written in subclass constructor?

A) Automatically called

✅ **Answer: A**

✅ 20 Interview Questions & Answers:

1. **Q:** What is a constructor?

   A: A special method used to initialize objects.

2. **Q:** Can constructor be private? Why?

   A: Yes, for singleton design patterns.

3. **Q:** Is constructor inherited?

   A: No, but subclass can call it using `super()`.

4. **Q:** What is constructor overloading?

   A: Defining multiple constructors with different parameters.

5. **Q:** What's the purpose of `this()`?

   A: To invoke another constructor in the same class.

6. **Q:** Can constructor call a method?

   A: Yes, from within its body.

7. **Q:** What's the difference between constructor and method?

   A: Constructor has no return type and same name as class.

8. **Q:** Can a class have both default and parameterized constructors?

   A: Yes.

9. **Q:** What is the role of `super()`?

   A: Calls the constructor of the superclass.

10. **Q:** Can constructor be final?

    A: No.

11. **Q:** What happens when no constructor is written?

    A: Java provides a default constructor.

12. **Q:** Can you create an object without using constructor?

    A: No, every object creation involves a constructor.

13. **Q:** Is it mandatory to define a constructor?

    A: No.

14. **Q:** Can abstract class have a constructor?

    A: Yes.

15. **Q:** Can we invoke constructor explicitly?

    A: Yes, using `new` keyword.

16. **Q:** Can `super()` and `this()` be used in the same constructor?

    A: No.

17. **Q:** Is constructor invoked during inheritance?

    A: Yes, base constructor is called first.

18. **Q:** Can constructor throw exceptions?

    A: Yes.

19. **Q:** Can constructor be overloaded with varying types?

    A: Yes.

20. **Q:** Can interface have constructor?

    A: No.

✅ Topic Outcome:

After studying this topic, learners will:

- Understand what constructors are and how they differ from methods.

- Apply default, parameterized, and overloaded constructors.
- Use `this()` and `super()` for constructor chaining and inheritance.

✅ Summary:

- Constructors are the backbone of **object initialization** in Java.
- They allow controlled and flexible creation of objects.
- Understanding constructor overloading and chaining empowers developers to write scalable, reusable, and maintainable object-oriented code.
- **Copy constructor** allows object cloning with controlled copy logic.

🔹 Topic 23: `this` Keyword and `super` Keyword in Java

✅ Definition:

In Java:

- The **`this`** keyword refers to the **current object** of the class.

- The `super` keyword refers to the **parent class** (superclass) and is used to access parent class members (methods, variables, constructors).

✅ Use Case:

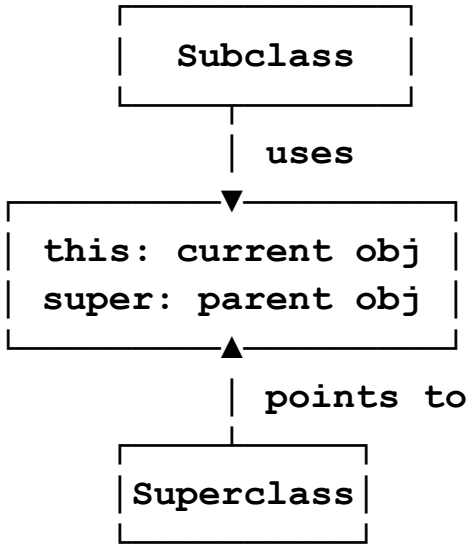
Use Case	Keyword Used
Refer current class instance variable	`this`
Call current class constructor	`this()`
Avoid confusion between instance and local variables	`this`
Access parent class variable hidden by subclass variable	`super`
Invoke superclass constructor	`super()`
Call overridden method from parent class	`super`

✅ Real-Time Usage:

Application	Keyword Example Use
Student Class with same variable names	`this.name = name;`
Calling parent class behavior	`super.displayDetails();`
Constructor chaining in base class	`this()` and `super()` in constructors

---

✅ Architecture Diagram: `this` vs `super` Relationships



✅ Syntax & Examples

🔹 Using `this` to refer current object:

```
public class Student {
    String name;

    Student(String name) {
        this.name = name;  // Resolves ambiguity
    }

    void display() {
        System.out.println("Name: " + this.name);
    }
}
```

- **Using `this()` to call another constructor:**

```
public class Box {
    int length, width;

    Box() {
        this(10, 20); // Calls the other constructor
    }

    Box(int l, int w) {
        this.length = l;
        this.width = w;
    }
}
```

- **Using `super` to access parent class methods/variables:**

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound();  // Calls parent method
        System.out.println("Dog barks");
    }
}
```

- **Using `super()` to invoke parent constructor:**

```
class Person {
    String name;
```

```
    Person(String name) {
        this.name = name;
    }
}

class Employee extends Person {
    Employee(String name) {
        super(name); // Calls constructor of Person
    }
}
```

✅ Keywords:

this, super, this(), super(), constructor, overridden, current object, parent class, constructor chaining

---

✅ Real-Time Example Code:

```
class Vehicle {
    int speed = 100;

    void display() {
        System.out.println("Speed: " + speed);
    }
}

class Car extends Vehicle {
    int speed = 180;

    void display() {
```

```
        System.out.println("Car speed: " + speed);
// Child variable
        System.out.println("Vehicle speed: " +
super.speed); // Parent variable
    }
}

public class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.display();
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. What does `this` refer to in Java?

   A) Superclass object

   B) Current class object

   ✅ **Answer: B**

2. What is the use of `super`?

   A) Refers to static methods

   B) Refers to parent class

   ✅ **Answer: B**

3. Which keyword resolves variable name ambiguity?

   A) new

   B) this

✅ **Answer: B**

4. Which calls the parent class constructor?

   A) this()

   B) super()

   ✅ **Answer: B**

5. Can `super()` be used in static methods?

   A) Yes

   B) No

   ✅ **Answer: B**

6. What happens if `super()` is not called in child constructor?

   A) Automatically inserted

   ✅ **Answer: A**

7. Can `this()` and `super()` be used together in constructor?

   A) Yes

   B) No

   ✅ **Answer: B**

8. Which keyword is used to call another constructor of the same class?

   A) this()

   ✅ **Answer: A**

9. Can `this` be used in static methods?

A) Yes

B) No

✅ **Answer: B**

10. Which statement is true about `super`?

A) Used to access child class methods

B) Used to access parent class methods

✅ **Answer: B**

11. `super()` must be the ____ statement in constructor.

A) last

B) first

✅ **Answer: B**

12. What will `this.variable = variable;` do?

A) Assign local value to instance variable

✅ **Answer: A**

13. Can `super` be used to call overridden methods?

A) Yes

✅ **Answer: A**

14. Can a constructor use both `this()` and `super()`?

A) Yes

B) No

✅ **Answer: B**

15. What happens if we remove both `this` and `super`?

A) Compiler adds default constructor call

✅ **Answer: A**

✅ 20 Interview Questions & Answers:

1. **Q:** What is `this` keyword?

   A: Refers to the current object of the class.

2. **Q:** Why use `this` keyword?

   A: To resolve ambiguity between class fields and parameters.

3. **Q:** What is `super` keyword?

   A: Refers to the parent class object.

4. **Q:** Can `this` be used to call another constructor?

   A: Yes, using `this()`.

5. **Q:** When is `super()` called?

   A: First line of subclass constructor.

6. **Q:** Can we override methods in superclass?

   A: Yes, and `super` can call the original method.

7. **Q:** Can `super` be used in methods?

   A: Yes, to access parent class variables and methods.

8. **Q:** Is it necessary to call `super()` in constructor?

   A: Not mandatory, added automatically if not provided.

9. **Q:** Can you use `this` in static context?

   A: No.

10. **Q:** Can `this()` be recursive?

    A: No, it causes stack overflow.

11. **Q:** Can `super()` be overloaded?

    A: No, but it calls appropriate parent constructor based on arguments.

12. **Q:** When do you use `this` in method chaining?

    A: When returning current object reference.

13. **Q:** What is constructor chaining?

    A: Calling one constructor from another using `this()`.

14. **Q:** Can `super` be used to access static members?

    A: No, use class name.

15. **Q:** Which comes first in constructor: `this()` or `super()`?

    A: Either, but not both and must be first statement.

16. **Q:** What if both subclass and superclass have a variable `x`?

    A: Use `super.x` to access parent's.

17. **Q:** Can abstract class use `super()`?

    A: Yes, for its constructor.

18. **Q:** Does `this` refer to current method?

    A: No, it refers to current object.

19. **Q:** Can interface use `super`?

    A: No, interfaces don't have constructors.

20. **Q:** Where is `this` commonly used?

    A: In setters, constructor chaining, method chaining.

✅ Topic Outcome:

After this topic, learners will:

- Understand how to use `this` and `super` for referencing.
- Chain constructors using `this()` and `super()`.
- Resolve ambiguity and call superclass methods cleanly.

✅ Summary:

- `this` and `super` are foundational keywords for object behavior and inheritance in Java.
- They improve code clarity, resolve ambiguity, and promote proper object-oriented practices, especially in polymorphism and constructor chaining.

🔹 Topic 24: OOPs Concepts in Java (Class, Object, Inheritance, Polymorphism, Encapsulation, Abstraction)

✅ Definition:

- **OOP (Object-Oriented Programming)** is a programming paradigm that uses **objects** and **classes** to design and build applications.

Java is a fully object-oriented language that implements the 4 core OOP principles:

1. **Encapsulation** – Binding data and methods into a single unit (class).
2. **Inheritance** – Reusing properties and methods of an existing class.
3. **Polymorphism** – Same interface but different behavior.
4. **Abstraction** – Hiding internal details and showing only functionality.

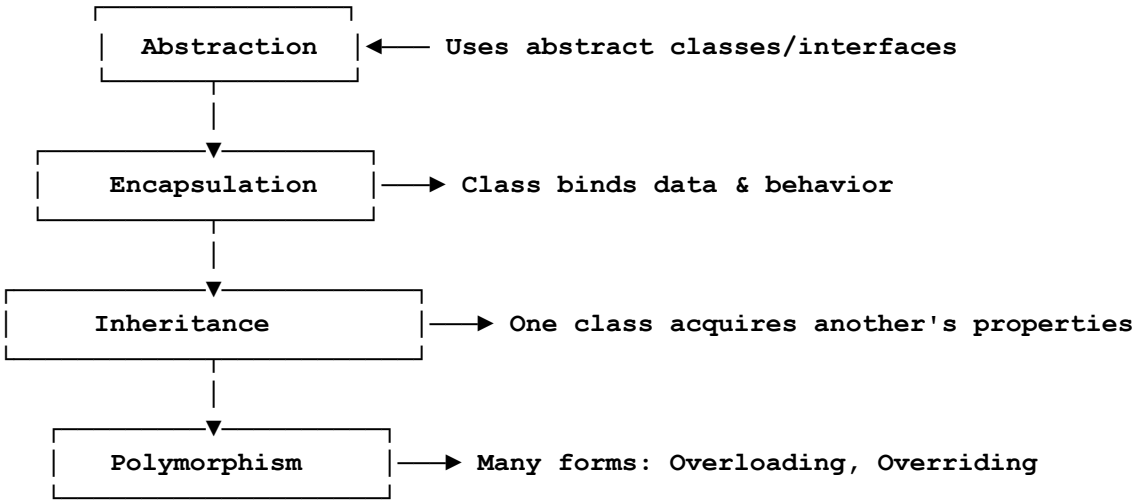
✅ Use Case:

Principle	Purpose
Encapsulation	Protect data with access control
Inheritance	Achieve code reusability
Polymorphism	Method overloading and overriding flexibility
Abstraction	Expose only relevant information

✅ Real-Time Usage:

Application Module	OOP Principle Example
Banking System	Account class (Encapsulation)
Vehicle Management	Car extends Vehicle (Inheritance)
Payment Gateway	Multiple payment methods (Polymorphism)
API SDK	Abstract interfaces for clients (Abstraction)

✅ Architecture Diagram: OOPs Model in Java



✅ Syntax & Subtopics with Examples

🔹 1. Class & Object

```
public class Person {
    String name;
    int age;

    void display() {
        System.out.println(name + " is " + age + " years
old.");
    }
}

public class Test {
    public static void main(String[] args) {
        Person p = new Person();  // Object creation
        p.name = "Alice";
        p.age = 25;
```

```
        p.display();
    }
}
```

🔹 2. Encapsulation

```
public class Employee {
    private String empName;

    public void setName(String name) {
        empName = name;
    }

    public String getName() {
        return empName;
    }
}
```

Note: Fields are `private`, access is via `public` methods (getters/setters).

🔹 3. Inheritance

```
class Animal {
    void eat() {
        System.out.println("Animal eats food");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

🔹 4. Polymorphism

- **Compile-Time (Method Overloading)**

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

- **Runtime (Method Overriding)**

```
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child method");
    }
}
```

🔹 5. Abstraction

- **Abstract Class**

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

- **Interface**

```
interface Drawable {
    void draw();
}

class Rectangle implements Drawable {
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}
```

✅ Keywords:

class, object, extends, implements, abstract, interface, override, private, public, protected, this, super, encapsulation, polymorphism, inheritance, abstraction

✅ Real-Time Example Code (All Concepts Together):

```
abstract class Vehicle {
    abstract void start();
}
```

```
class Car extends Vehicle {
    private String model;

    Car(String model) {
        this.model = model;
    }

    public void start() {
        System.out.println(model + " starts with key.");
    }

    public void displayModel() {
        System.out.println("Model: " + model);
    }
}
```

```
public class OOPExample {
    public static void main(String[] args) {
        Vehicle v = new Car("Toyota");
        v.start();  // Polymorphism
        ((Car)v).displayModel();  // Downcasting
    }
}
```

✅ 15 MCQ Questions (with Options & Answers):

1. What is OOP?

   A) Object Oriented Process

   B) Object Oriented Programming

   ✅ **Answer: B**

2. Which is not a core concept of OOP?

   A) Compilation

   B) Encapsulation

   ✅ **Answer: A**

3. What does inheritance support?

   A) Code reusability

   ✅ **Answer: A**

4. Which keyword is used to inherit a class?

   A) extends

   ✅ **Answer: A**

5. Which of these supports abstraction?

   A) Interface

   ✅ **Answer: A**

6. How is encapsulation achieved?

   A) Using private fields and public methods

   ✅ **Answer: A**

7. What is polymorphism?

A) One task with many forms

✅ **Answer: A**

8. What is method overloading?

A) Same name, different parameters

✅ **Answer: A**

9. What is overriding?

A) Subclass redefining parent method

✅ **Answer: A**

10. Which class cannot be instantiated?

A) Abstract class

✅ **Answer: A**

11. Can an interface contain variables?

A) Yes, they are public static final

✅ **Answer: A**

12. Can an abstract class have a constructor?

A) Yes

✅ **Answer: A**

13. Does Java support multiple inheritance?

A) No (classes), Yes (interfaces)

✅ **Answer: A**

14. What is used to hide implementation details?

A) Abstraction

✅ **Answer: A**

15. What is the smallest unit of OOP in Java?

A) Object

✅ **Answer: A**

✅ 20 Interview Questions & Answers:

1. **Q:** What are the main pillars of OOP?

   A: Encapsulation, Inheritance, Polymorphism, Abstraction

2. **Q:** Difference between class and object?

   A: Class is a blueprint; object is its instance.

3. **Q:** What is encapsulation?

   A: Binding data with methods and controlling access.

4. **Q:** How is abstraction achieved in Java?

   A: Using abstract classes and interfaces.

5. **Q:** What is the difference between overloading and overriding?

   A: Overloading = compile-time; Overriding = runtime.

6. **Q:** Can we override static methods?

   A: No, they are hidden, not overridden.

7. **Q:** Can abstract class have a body?

   A: Yes, it can have both abstract and concrete methods.

8. **Q:** Why Java does not support multiple inheritance with classes?

   A: To avoid ambiguity (Diamond problem).

9. **Q:** What is polymorphism?

   A: One interface, many forms.

10. **Q:** Is Java 100% OOP?

    A: No, because it has primitive types.

11. **Q:** What is interface?

    A: A contract containing abstract methods and constants.

12. **Q:** How is abstraction different from encapsulation?

    A: Abstraction hides complexity; encapsulation hides data.

13. **Q:** Can you instantiate an abstract class?

    A: No.

14. **Q:** What is the use of `final` keyword in OOP?

    A: Prevent inheritance or method overriding.

15. **Q:** Can interface have constructor?

    A: No.

16. **Q:** Can an abstract class implement an interface?

    A: Yes.

17. **Q:** Can a method be both abstract and final?

    A: No.

18. **Q:** What is object slicing?

    A: Not supported in Java, happens in C++.

19. **Q:** Is polymorphism achieved via inheritance?

    A: Yes, through overriding.

20. **Q:** What happens when abstract method is not implemented?

    A: Compilation error.

✅ Topic Outcome:

After learning this topic, students will be able to:

- Design class-based architectures using Java OOPs principles.
- Write reusable, modular, and extendable code.
- Implement abstraction using interfaces and abstract classes.

✅ Summary:

- OOPs in Java allows developers to model real-world entities, enforce clean architecture, and write scalable code.
- Mastering OOP principles is essential to becoming a professional Java developer.

🔹 Types of Inheritance in Java

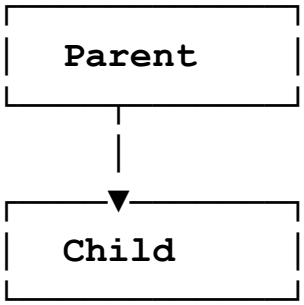
- Java supports the concept of inheritance to promote **code reusability**, **modularity**, and **logical hierarchy**.
- While Java does not support **multiple** and **hybrid** inheritance through **classes**, it does support them through **interfaces**.

✅ 1. Single-Level Inheritance

Definition:

A child class inherits directly from a single parent class.

Diagram:



Example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // inherited method
```

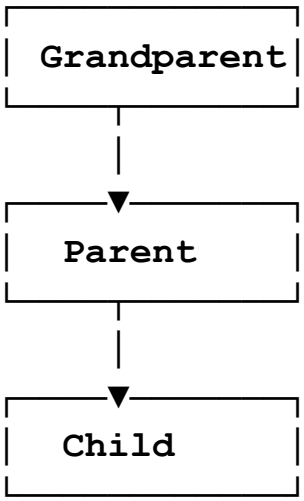
```
        d.bark();   // child method
    }
}
```

✅ 2. Multi-Level Inheritance

Definition:

A class inherits from a derived class, creating a multi-level hierarchy.

Diagram:



Example:

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

```
class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy weeps");
    }
}
```

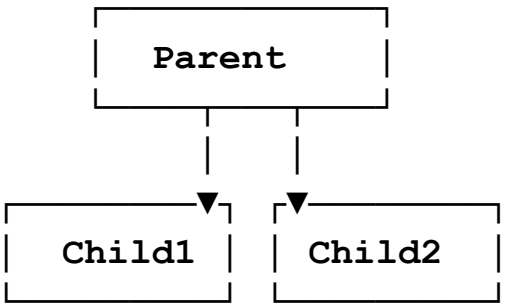
```
public class Test {
    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();
        p.bark();
        p.weep();
    }
}
```

✅ 3. Hierarchical Inheritance

Definition:

Multiple classes inherit from a single parent class.

Diagram:



Example:

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

```
class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
```

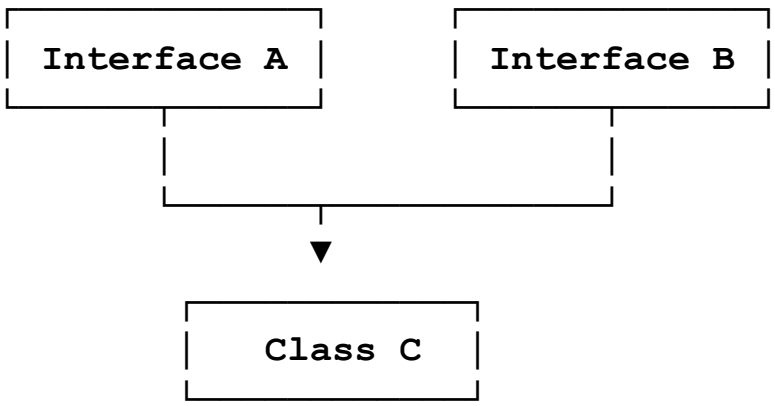
```
        d.bark();

        Cat c = new Cat();
        c.eat();
        c.meow();
    }
}
```

✅ 4. Multiple Inheritance (Using Interfaces in Java)

Definition:

- A class implements multiple interfaces.
- Java doesn't allow multiple inheritance with classes to avoid ambiguity but allows it via interfaces.

Diagram:



Example:

```
interface Printable {
    void print();
}
```

```
interface Showable {
    void show();
}
```

```
class Document implements Printable, Showable {
    public void print() {
        System.out.println("Printing...");
    }
    public void show() {
        System.out.println("Showing...");
    }
}
```

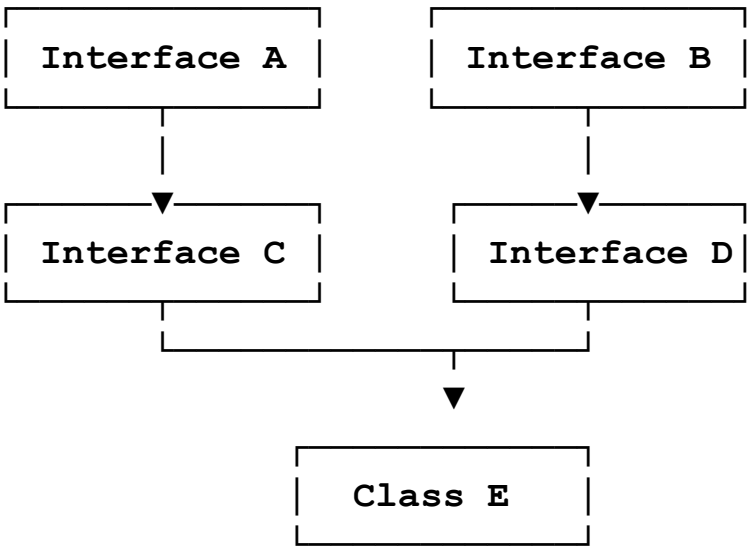
```
public class Test {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print();
        doc.show();
    }
}
```

✅ 5. Hybrid Inheritance (Achieved via Interfaces)

Definition:

Combination of more than one type of inheritance.

Diagram:



Example:

```
interface A {
    void methodA();
}
```

```
interface B {
    void methodB();
}
```

```
interface C extends A, B {
    void methodC();
}
```

```
class Hybrid implements C {
    public void methodA() {
        System.out.println("Method A");
    }
```

```
    public void methodB() {
        System.out.println("Method B");
    }

    public void methodC() {
        System.out.println("Method C");
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Hybrid h = new Hybrid();
        h.methodA();
        h.methodB();
        h.methodC();
    }
}
```

✅ Summary Table:

Inheritance Type	Supported in Java	Achieved Using	Code Reusability	Ambiguity Risk
Single-level	✅ Yes	Class	✅ High	❌ None
Multi-level	✅ Yes	Class	✅ High	❌ None
Hierarchical	✅ Yes	Class	✅ Medium	❌ None
Multiple	✅ Yes (via interfaces)	Interfaces	✅ High	❌ None

Hybrid	✅ Yes (via interfaces)	Interfaces	✅ High	❌ None

🔹 Types of Polymorphism in Java

✅ Definition:

- **Polymorphism** means "**many forms**".
- In Java, it allows one interface or method to behave differently based on how it is used.
- It enhances code reusability, flexibility, and readability.

✅ Types of Polymorphism in Java:

Type	Resolved At	Mechanism
Compile-time Polymorphism	Compile time	Method Overloading
Run-time Polymorphism	Runtime	Method Overriding

✅ 1. Compile-Time Polymorphism (Method Overloading)

Definition:

When multiple methods in the same class have the **same name but different parameters**, it is called **method overloading**.

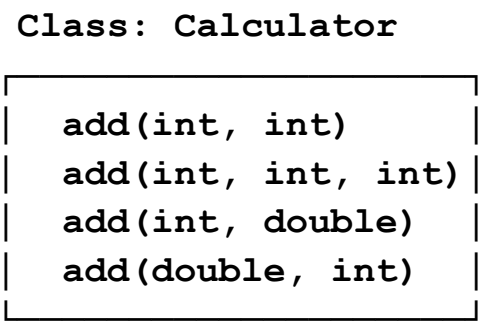
✅ a) By changing the number of arguments

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

✅ b) By changing the data types of parameters

```
class Calculator {
    double add(int a, double b) {
        return a + b;
    }

    double add(double a, int b) {
        return a + b;
    }
}
```

✅ Diagram:



✅ 2. Run-Time Polymorphism (Method Overriding)

Definition:

- Method overriding occurs when a **subclass provides a specific implementation** of a method already defined in its **parent class**.
- The call is resolved **at runtime** using **dynamic dispatch**.

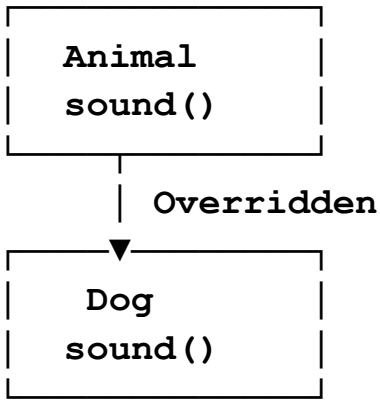
```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
```

```
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Usage:

```
Animal obj = new Dog();
obj.sound();  // Output: Dog barks (resolved at
runtime)
```

✅ Diagram for Runtime Polymorphism:



🔹 Implicit vs Explicit Method Overriding

Type	Description	Example Keyword
Implicit Overriding	Subclass overrides method without `@Override` annotation (not enforced)	None
Explicit Overriding	Subclass uses `@Override` annotation to **ensure correct overriding**	`@Override`

✅ Implicit Overriding Example:

```
class Parent {
    void show() {
        System.out.println("Parent class");
    }
}
```

```
class Child extends Parent {
    void show() {  // No @Override
        System.out.println("Child class");
    }
}
```

✅ Explicit Overriding Example:

```
class Parent {
    void display() {
        System.out.println("Parent");
    }
}
```

```
class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child");
    }
}
```

Note: Using `@Override` ensures compile-time checking and prevents mistakes (like typo or signature mismatch).

✅ Summary:

- **Compile-time polymorphism** uses method overloading.
- **Run-time polymorphism** uses method overriding.
- **Explicit overriding** is preferred using `@Override`.

🔹 Topic 25: Wrapper Classes and Their Methods in Java

✅ Definition:

- **Wrapper classes** in Java provide a way to use **primitive data types** (like `int`, `char`, `boolean`, etc.) as **objects**.
- Each primitive type has a corresponding wrapper class in the `java.lang` package.

Primitive Type	Wrapper Class
`byte`	`Byte`

`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

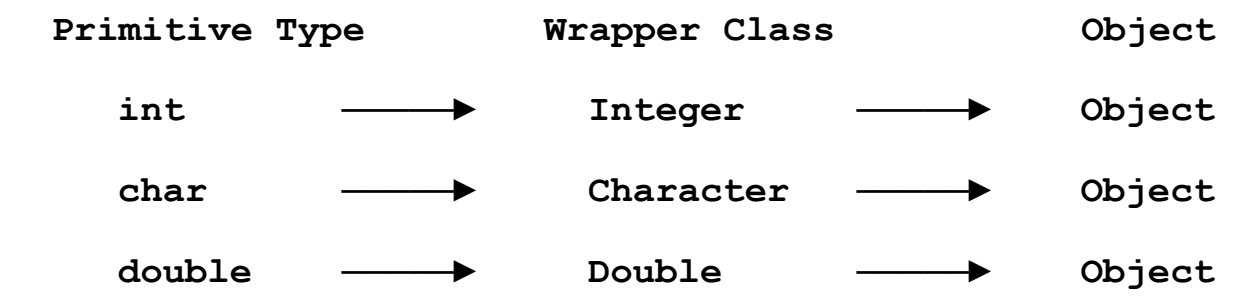
✅ Use Case:

- Used in **collections** (like `ArrayList`, `HashMap`) that work with objects, not primitives.
- Useful for **object-oriented features** like serialization, synchronization, and generics.
- Enables **conversion between strings and numeric values** (`Integer.parseInt()`, etc.).

✅ Real-Time Usage:

Application Area	Usage
Web/Database Applications	Fetch numeric input as strings → parse to numbers
Collections (e.g. List)	Store integers using `Integer`, not `int`
Parsing & Validation	`Double.parseDouble("45.67")`
Optional/Nullable Values	Use wrapper to allow nullability

✅ Architecture Diagram



Autoboxing and Unboxing helps in seamless conversion between these types.

✅ Syntax & Usage

🔹 Autoboxing:

```
int a = 10;

Integer obj = a; // Autoboxing
```

🔹 Unboxing:

```
Integer obj = 20;

int b = obj; // Unboxing
```

✅ Common Wrapper Methods

Wrapper	Method	Description
Integer	`parseInt(String)`	Converts string to int
Integer	`valueOf(String)`	Returns Integer object from string
Integer	`intValue()`	Returns int from Integer object

Double	`parseDouble(String)`	Converts string to double
Boolean	`parseBoolean(String)`	Converts string to boolean
Character	`isLetter(char)`	Checks if character is a letter
Character	`isDigit(char)`	Checks if character is a digit

✅ Example 1: Using Integer Wrapper

```
public class WrapperExample {
    public static void main(String[] args) {
        int x = 100;
        Integer obj = Integer.valueOf(x); // Boxing
        int y = obj.intValue();           // Unboxing
        System.out.println("Value: " + y);
    }
}
```

✅ Example 2: Autoboxing & Unboxing with Collections

```
import java.util.ArrayList;

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

```
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);     // Autoboxing
        int num = list.get(0); // Unboxing
        System.out.println(num);
    }
}
```

✅ Example 3: Using Wrapper Methods

```
public class ParseTest {
    public static void main(String[] args) {
        String s = "123";
        int num = Integer.parseInt(s);
        double d = Double.parseDouble("456.78");

        System.out.println(num + d); // 579.78
    }
}
```

✅ Real-Time Example Code:

```
public class LoginValidator {
    public static void main(String[] args) {
        String ageStr = "20";

        if (Integer.parseInt(ageStr) >= 18) {
            System.out.println("Valid age to register.");
        } else {
            System.out.println("Too young to register.");
        }
    }
}
```

✅ 15 MCQ Questions (with Answers):

1. Which wrapper class corresponds to `int`?

   A) Float

   B) Integer

   ✅ **Answer: B**

2. What is Autoboxing?

   A) Converting object to primitive

   B) Converting primitive to object

   ✅ **Answer: B**

3. What does `Integer.parseInt("123")` return?

   A) Integer object

   B) int

   ✅ **Answer: B**

4. Which method converts string to double?

   A) `Double.parse()`

   B) `Double.parseDouble()`

   ✅ **Answer: B**

5. Can wrapper classes be null?

   A) No

   B) Yes

✅ **Answer: B**

6. What is the output of `Boolean.parseBoolean("true")`?

   A) false

   B) true

   ✅ **Answer: B**

7. What will `Integer.valueOf("123")` return?

   A) int

   B) Integer object

   ✅ **Answer: B**

8. Which class is used to represent single character?

   A) String

   B) Character

   ✅ **Answer: B**

9. Can wrapper classes be used in collections?

   A) Yes

   ✅ **Answer: A**

10. Which method checks if a char is digit?

    A) `Character.isNumber()`

    B) `Character.isDigit()`

    ✅ **Answer: B**

11. Which method returns int from Integer object?

A) `getValue()`

B) `intValue()`

✅ **Answer: B**

12. Which keyword is used for autoboxing?

A) box

B) None

✅ **Answer: B**

13. Are wrapper classes immutable?

A) No

B) Yes

✅ **Answer: B**

14. Which of these is not a wrapper class?

A) String

✅ **Answer: A**

15. What is the parent of all wrapper classes?

A) Object

✅ **Answer: A**

✅ 20 Interview Q&A:

1. **Q:** What are wrapper classes in Java?
   A: Classes that convert primitive types into objects.

2. **Q:** Why do we need wrapper classes?
   A: To use primitives where objects are required (e.g. Collections).

3. **Q:** Name all wrapper classes.
   A: Byte, Short, Integer, Long, Float, Double, Character, Boolean.

4. **Q:** What is autoboxing?
   A: Automatic conversion from primitive to wrapper object.

5. **Q:** What is unboxing?
   A: Automatic conversion from wrapper to primitive.

6. **Q:** Are wrapper classes immutable?
   A: Yes.

7. **Q:** Can wrapper classes hold null values?
   A: Yes.

8. **Q:** Which method parses string to int?

   A: `Integer.parseInt(String)`

9. **Q:** What is the difference between `parseInt()` and `valueOf()`?

   A: `parseInt()` returns primitive; `valueOf()` returns object.

10. **Q:** Can wrapper objects be compared using `==`?

    A: Only if within the same object range; use `.equals()` instead.

11. **Q:** Can we extend wrapper classes?

    A: No, they are final.

12. **Q:** How does autoboxing work in ArrayList?

    A: Automatically converts `int` to `Integer` when added.

13. **Q:** What happens on `Integer i = null; int j = i;`?

    A: Throws `NullPointerException`.

14. **Q:** How to convert `Integer` to `String`?

    A: Using `String.valueOf(i)` or `i.toString()`.

15. **Q:** Can we override methods in wrapper classes?
    A: No, they are final classes.

16. **Q:** Are wrapper classes part of java.lang package?
    A: Yes.

17. **Q:** Is `Character` a subclass of `String`?
    A: No.

18. **Q:** What does `Double.isNaN()` do?
    A: Checks if a double is Not-a-Number.

19. **Q:** Can you modify the value of an Integer object?
    A: No, it's immutable.

20. **Q:** How to convert `String "123"` to `Integer` object?
    A: `Integer.valueOf("123")`

✅ Topic Outcome:

After this topic, learners will:

- Know how to convert primitives to objects and vice versa.
- Use wrapper methods for parsing and formatting.
- Apply wrapper classes in collections and generics effectively.

✅ Summary:

- Wrapper classes enable Java to treat primitive data types as full-fledged objects, allowing their use in collections, generics, and APIs.
- With features like autoboxing and useful static methods, wrapper classes are essential in modern Java development.