# CHAPTER NO:02
# Classes, Objects and Methods    {12 MARKS}

2.1 Class Fundamentals, The General Form of a Class, A Simple Class

2.2 Declaring Objects, A Closer Look at new, Assigning Object Reference Variables

2.3 Introducing Methods, adding a Method to the Class, returning a Value, adding a Method that takes parameters

2.4 Constructors, Parameterized Constructors

2.5 'this' Keyword

2.6 static data, method, and blocks

2.7 String class and its methods

# 2.1 Class Fundamentals, The General Form of a Class, A Simple Class:

- In Java, **everything revolves around classes and objects**.
- A **class** defines the **blueprint or template** for creating objects.
- It contains:
- **Fields (variables)**
- **Methods (functions)**
- **Constructors**
- **Nested classes**
- and more.
- Objects are **instances** of classes.

# The General Form of a Class

```
class ClassName {
    // Fields (variables)
    type variableName;


    // Constructor(s)
    ClassName() {
        // Initialization code
    }



    // Methods
    returnType methodName(parameters) {
        // Method body
    }



    // Main method (if needed)
    public static void main(String[] args) {
        // Code to execute
    }

}
```

# A Simple Class Example

```java
class Rectangle {
    int width;
    int height;

    // Method to compute area
    int area() {
        return width * height;
    }
}

public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(); // Creating object
        rect.width = 10;
        rect.height = 5;

        int result = rect.area(); // Calling method
        System.out.println("Area is: " + result);
    }
}
```

**Output:**

```
Area is: 50
```

# 2.2 Declaring Objects, A Closer Look at new, Assigning Object Reference Variables:

- An **object** is an instance of a class.
- It holds actual data defined by the class fields.

**Declaring an Object Reference Variable**

- To declare an object reference variable (a variable that can point to an object), you specify the class name followed by the variable name:

```
ClassName objectRef;
```

- This only **declares** the reference; it does **not** create the object itself.

# A Closer Look at `new`

To actually create an object in memory, you use the `new` keyword, which:

- Allocates memory for the object

- Calls the class constructor to initialize the object

- Returns a reference (memory address) to the newly created object

Example:

```
objectRef = new ClassName();
```

## Assigning Object Reference Variables

```
ClassName objectRef = new ClassName();
```

Now, `objectRef` points to a newly created object.

```java
class Car {
    String color;

    void displayColor() {
        System.out.println("Car color is: " + color);
    }
}

public class Main {
    public static void main(String[] args) {
        // Declare an object reference (no object created yet)
        Car myCar;

        // Create a Car object and assign its reference to myCar
        myCar = new Car();

        // Assign value to object's field
        myCar.color = "Red";

        // Call method
        myCar.displayColor();
    }
}
```

Output:

Car color is: Red

# 2.3 Introducing Methods, adding a Method to the Class, returning a Value, adding a Method that takes parameters:

## Introducing Methods

- A **method** in Java is a block of code that performs a specific task.
- It's similar to a function in other programming languages.

## Syntax of a Method:

```
returnType methodName(parameterList) {
    // method body
}
```

- `returnType` — the type of value the method returns (e.g., `int`, `void`)

- `methodName` — name of the method

- `parameterList` — optional input parameters (like variables passed into the method)

# Adding a Method to the Class

```java
class Calculator {
    // Method that displays a message
    void showMessage() {
        System.out.println("Welcome to Calculator!");
    }
}
```

# Returning a Value from a Method

You can create a method that returns a value using a `return` statement.

```java
class Calculator {
    int add() {
        int a = 5;
        int b = 10;
        return a + b; // returns 15
    }
}
```

# Calling It:

```java
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(); // Calls add() and stores result
        System.out.println("Sum is: " + result);
    }
}
```

# Output:

```
Sum is: 15
```

# Adding a Method that Takes Parameters

Methods can take **arguments (parameters)** to operate on dynamic input values:

```java
class Calculator {
    int add(int x, int y) {
        return x + y;
    }
}
```

## Calling It:

```java
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(7, 3); // Pass values as arguments
        System.out.println("Sum is: " + result);
    }
}
```

## Output:

```
Sum is: 10
```

# 2.4 Constructors, Parameterized Constructors:

- In Java, **constructors** are special methods that are called when an object is created.

- They initialize the object with default or specific values.

- A **constructor** is a block of code that gets executed when an instance (object) of a class is created.

- It has the same name as the class and does not have a return type (not even void).

**Types of Constructors:**

1. Default Constructor (No-Argument Constructor)

2. Parameterized Constructor

# 1. Default Constructor (No-Argument Constructor):

- A **default constructor** is a constructor that takes no parameters.

-  If no constructor is explicitly defined, Java provides a **default constructor** automatically.

- This constructor initializes the object with default values (such as null for objects, 0 for integers, false for Booleans, etc.).

```java
class Car {
    String brand;
    int year;

    // Default constructor
    Car() {
        brand = "Unknown";
        year = 0;
    }

    void display() {
        System.out.println("Brand: " + brand);
        System.out.println("Year: " + year);
    }
}
```

```java
    public static void main(String[] args) {
        Car myCar = new Car();   // Default constructor is called
        myCar.display();
    }
}
```

## Output

```
Brand: Unknown
Year: 0
```

The `Car()` constructor is a default constructor that initializes the `brand` to `"Unknown"` and the `year` to `0`.

# 2. Parameterized Constructor:

- A **parameterized constructor** is a constructor that takes parameters (arguments) to initialize the object with specific values at the time of object creation.

```java
class Car {
    String brand;
    int year;

    // Parameterized constructor
    Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    void display() {
        System.out.println("Brand: " + brand);
        System.out.println("Year: " + year);
    }
}
```

```java
public static void main(String[] args) {
    // Creating objects with different values using parameterized constructor
    Car car1 = new Car("Toyota", 2020);
    Car car2 = new Car("Honda", 2021);


    car1.display();
    car2.display();
}
}
```

**Output**

```
Brand: Toyota

Year: 2020


Brand: Honda

Year: 2021
```

The `Car(String brand, int year)` constructor takes two parameters and initializes the `brand` and `year` fields with the values passed during object creation.

# NOTE:

1. **Constructor Name**: The constructor name must be the same as the class name.

2. **No Return Type**: Constructors do not have a return type (not even `void`).

3. **Automatically Called**: Constructors are automatically invoked when an object is created using the `new` keyword.

4. **Can Have Parameters**: A constructor can have parameters (parameterized constructor) or not (default constructor).

5. **Constructor Overloading**: You can define multiple constructors with different parameter lists, allowing you to create objects in different ways.

# Constructor Overloading (Multiple Constructors):

- In Java, we can have **multiple constructors** with different parameter lists (constructor overloading).

- The constructor to be called is chosen based on the number and type of arguments passed.

```java
class Car {
    String brand;
    int year;

    // Constructor 1: Default constructor
    Car() {
        brand = "Unknown";
        year = 0;
    }

    // Constructor 2: Parameterized constructor with one parameter
    Car(String brand) {
        this.brand = brand;
        year = 2022;  // Default year if only brand is passed
    }
```

```java
// Constructor 3: Parameterized constructor with two parameters
Car(String brand, int year) {
    this.brand = brand;
    this.year = year;
}


void display() {
    System.out.println("Brand: " + brand);
    System.out.println("Year: " + year);
}
```

```java
public static void main(String[] args) {
    Car car1 = new Car();  // Calls default constructor
    Car car2 = new Car("Ford");  // Calls parameterized constructor with one parameter
    Car car3 = new Car("BMW", 2023);  // Calls parameterized constructor with two parameters


    car1.display();
    car2.display();
    car3.display();
  }
}
```

**Output:**

```
Brand: Unknown
Year: 0
Brand: Ford
Year: 2022
Brand: BMW
Year: 2023
```

# 2.5 'this' Keyword:

- In Java, the this keyword is used to refer to **the current instance of the class**. It is used within a class to reference the **current object** that the method or constructor is being called on.

- The **this** keyword is mainly used to:

1. **Refer to the instance variables (fields)** of the current class.

2. **Invoke the current class's constructor**.

3. **Pass the current object** as a parameter to other methods.

# 1. Referring to Instance Variables:

- The this keyword is used to differentiate between the **instance variable** and the **local variable** (parameter) when they have the same name.

```java
class Student {
    String name;   // Instance variable

    // Constructor with a parameter
    Student(String name) {
        this.name = name;  // 'this.name' refers to the instance variable, 'name' is the parameter
    }

    void display() {
        System.out.println("Name: " + this.name);  // Refers to the instance variable 'name'
    }
}
```

```java
    public static void main(String[] args) {
        Student student = new Student("John");
        student.display();  // Output: Name: John
    }
}
```

In the constructor, the parameter `name` and the instance variable `name` are both the same. The `this` keyword is used to refer to the **instance variable**, differentiating it from the parameter.

# 2. Calling a Constructor Using <u>this</u>:

- In Java, we can use the this() keyword to call another constructor of the same class from within a constructor.

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

    // Constructor with one parameter
    Student(String name) {
        this(name, 20);  // Calls the second constructor
    }

    // Constructor with two parameters
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

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

    public static void main(String[] args) {
        Student student = new Student("John");
        student.display();  // Output: Name: John, Age: 20
    }
}
```

The constructor `Student(String name)` calls the second constructor `Student(String name, int age)` using `this(name, 20)`. This helps avoid code duplication.

# or

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

    Student(String name) {
        this(name, 20); // default age is 20
    }

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

```java
public static void main(String[] args) {
    Student s1 = new Student("Akhilesh");
    Student s2 = new Student("Romil", 25);

    System.out.println(s1.name + " " + s1.age); // Akhilesh 20
    System.out.println(s2.name + " " + s2.age); // Romil 25
  }
}
```

# 3. Passing the Current Object:

- The **this** keyword can be used to pass the current object as a parameter to another method.

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

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

    // Method to display student info
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

```java
// Method to compare current student with another
void compare(Student other) {
    if (this.age == other.age) {
        System.out.println(this.name + " and " + other.name + " are of the same age.");
    } else {
        System.out.println(this.name + " and " + other.name + " have different ages.");
    }
}
```

```java
public static void main(String[] args) {
    Student student1 = new Student("John", 20);
    Student student2 = new Student("Alice", 20);

    student1.compare(student2);  // Output: John and Alice are of the same age.
    }
}
```

The `compare()` method compares the `age` of the current student ( `this` ) with another `Student` object passed as a parameter.

The `this` keyword refers to the current object ( `student1` ) inside the `compare()` method.

# NOTE:

1. **Referring to instance variables**: Use `this` to refer to instance variables when they have the same name as local variables or parameters.

2. **Calling another constructor**: Use `this()` to call another constructor within the same class.

3. **Passing the current object**: Use `this` to pass the current object to another method or constructor.

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

    // Constructor to initialize name and age
    Person(String name, int age) {
        this.name = name;  // Using 'this' to refer to instance variable
        this.age = age;    // Using 'this' to refer to instance variable
    }

    // Constructor that calls another constructor
    Person(String name) {
        this(name, 25);  // Using 'this' to call another constructor
    }
}
```

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

    public static void main(String[] args) {
        Person p1 = new Person("John", 30);
        Person p2 = new Person("Alice");

        p1.display();  // Output: Name: John, Age: 30
        p2.display();  // Output: Name: Alice, Age: 25
    }
}
```

The this keyword helps you refer to the current object and avoid ambiguity when variable names overlap, and also helps manage constructor chaining and object passing

# 2.6 static data, method, and blocks:

- In Java, the **static** keyword is used to create class-level members.
- This means that static variables, methods, and blocks belong to the class itself rather than to individual objects of the class.

# 1. Static Data (Variables)

- A **static variable** is shared by all instances (objects) of the class.

- When a variable is declared as static, there is only **one copy of that variable** for all objects of the class.

- The value of this variable is the same for all objects.

# Example:

```java
class Example {
    static int count = 0;  // Static variable

    Example() {
        count++;  // Incremented every time an
                        object is created
    }

    void display() {
        System.out.println("Count: " + count);
        // Prints the same count for all objects
    }
```

```java
    public static void main(String[] args)
    {
    Example obj1 = new Example();
    Example obj2 = new  Example();

    obj1.display();  //Output: Count: 2
    obj2.display();  // Output: Count: 2
    }
}
```

```java
class Example {
    static int count = 0;  // Static variable

    Example() {
        count++;  // Incremented every time an object is created
    }

    void display() {
        System.out.println("Count: " + count);  // Prints the same count for all objects
    }
}
```

```java
public static void main(String[] args) {
    Example obj1 = new Example();   // First object
    Example obj2 = new Example();   // Second object

    obj1.display();   // Output: Count: 2
    obj2.display();   // Output: Count: 2
}
}
```

- The count variable is static, meaning it is shared between both obj1 and obj2.
- Each time an object is created, count gets incremented.
- Therefore, both objects print the same count value

# 2. Static Method:

- A **static method** can be called without creating an object of the class.
- It can only access **static variables and other static methods**.
- It cannot access non-static variables or methods directly.

# Example:

```
class Example {
    static int count = 0;

    static void increment() {
        count++;  // Static method can access static variables
    }

    public static void main(String[] args) {
        Example.increment();  // Calling static method without object
        System.out.println("Count: " + count);  // Output: Count: 1
    }
}
```

```java
class Example {
    static int count = 0;

    static void increment() {
        count++;  // Static method can access static variables
    }

    public static void main(String[] args) {
        Example.increment();  // Calling static method without object
        System.out.println("Count: " + count);  // Output: Count: 1
    }
}
```

- The increment() method is static, so it can be called directly using the class name, without creating an object.

- This method increments the static variable count.

**NOTE:**

**1.  Static Data (Variables)**:

- Shared across all instances of the class.

- All objects refer to the same memory location for this variable.

**2.  Static Method**:

- Can be called without an object.

- It can only access static variables and methods.

# 3. Static block:

- A **static block** (also called a static initialization block) is used to **initialize static variables** or run code once when the class is loaded, **before** any object is created or any static method is called.

## Syntax:

```
class ClassName {
    static {
        // Code that runs once when the class is loaded
    }
}
```

```java
class Demo {
    static {
        System.out.println("Static block executed.");
    }

    public static void main(String[] args) {
        System.out.println("Main method executed.");
    }
}
```

## Output:

```
Static block executed.
Main method executed.
```

The static block runs **before** the `main()` method, because it's part of class loading.

# 2.7 String class and its methods:

- In Java, the **String class** is part of the java.lang package and is used to represent a sequence of characters.

- Strings are **immutable** in Java, meaning once a string is created, it cannot be modified. If you modify a string, a new string is created instead.

- **Key Features of the String Class:**

1. **Immutable:** Once a string is created, it cannot be changed.

2. **String Pool:** Java uses a string pool to store string literals for memory optimization.

# Common Methods in the String Class:

1. `length()`

- **Purpose**: Returns the number of characters in the string.

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

The string `"Hello"` has 5 characters, so `length()` returns 5.

## 2. `charAt(int index)`

- **Purpose**: Returns the character at the specified index.

```
String str = "Hello";
System.out.println(str.charAt(1));   // Output: e
```

`charAt(1)` returns the character at index `1` (second character, which is 'e').

3. `substring(int startIndex)` / `substring(int startIndex, int endIndex)`

- **Purpose**: Returns a new string that is a substring of the original string.

```
String str = "Hello, World!";
System.out.println(str.substring(7));      // Output: World!
System.out.println(str.substring(0, 5));   // Output: Hello
```

- `substring(7)` extracts from index 7 to the end.

- `substring(0, 5)` extracts from index 0 to 4 (5 is excluded).

## 4. `toLowerCase()` / `toUpperCase()`

- **Purpose**: Converts all characters in the string to lowercase or uppercase.

```java
String str = "Hello";
System.out.println(str.toLowerCase());  // Output: hello
System.out.println(str.toUpperCase());  // Output: HELLO
```

- `toLowerCase()` converts all characters to lowercase.

- `toUpperCase()` converts all characters to uppercase.

## 5. `equals(String anotherString)`

- **Purpose**: Compares two strings for **exact equality** (case-sensitive).

```java
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equals(str2));  // Output: false
```

This method returns `true` because it ignores case differences.

## 6. `equalsIgnoreCase(String anotherString)`

- **Purpose**: Compares two strings for equality, ignoring case.

```
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equalsIgnoreCase(str2));   // Output: true
```

This method returns `true` because it ignores case differences.

## 7. `trim()`

- **Purpose**: Removes any leading and trailing whitespace from the string.

```java
String str = "   Hello World!   ";
System.out.println(str.trim());  // Output: "Hello World!"
```

The `trim()` method removes the spaces before and after the text.

## 8. `replace(char oldChar, char newChar)`

- **Purpose**: Replaces all occurrences of a character with another character.

```java
String str = "Hello World!";
System.out.println(str.replace('o', 'a'));  // Output: Hella Warld!
```

It replaces all occurrences of `'o'` with `'a'` .

## 9. `contains(CharSequence sequence)`

- **Purpose**: Checks if the string contains a specific sequence of characters.

```java
String str = "Hello World!";
System.out.println(str.contains("World"));  // Output: true
System.out.println(str.contains("world"));  // Output: false
```

`contains()` returns `true` if the specified sequence of characters is found in the string

## 10. `indexOf(String str)`

- **Purpose**: Returns the index of the first occurrence of the specified substring.

```
String str = "Hello, World!";
System.out.println(str.indexOf("World"));  // Output: 7
System.out.println(str.indexOf("Java"));    // Output: -1
```

`indexOf()` returns the index where the substring first occurs. If not found, it returns `-1`.

## 11. `split(String regex)`

- **Purpose**: Splits the string into an array of substrings based on the provided delimiter.

```java
String str = "apple,banana,orange";
String[] fruits = str.split(",");
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

**Output**

```
apple
banana
orange
```

`split(",")` divides the string into an array of strings based on commas.

## 12. `startsWith(String prefix)`

- **Purpose**: Checks if the string starts with the specified prefix.

```java
String str = "Hello World!";
System.out.println(str.startsWith("Hello"));  // Output: true
System.out.println(str.startsWith("World"));  // Output: false
```

Returns `true` if the string starts with the given prefix.

## 13. `endsWith(String suffix)`

- **Purpose**: Checks if the string ends with the specified suffix.

```java
String str = "Hello World!";
System.out.println(str.endsWith("World!"));  // Output: true
System.out.println(str.endsWith("Hello"));   // Output: false
```

Returns `true` if the string ends with the given suffix.

# Summary of Common String Methods:

1. `length()` - Returns the length of the string.

2. `charAt(int index)` - Returns the character at the specified index.

3. `substring(int startIndex)` / `substring(int startIndex, int endIndex)` - Extracts a portion of the string.

4. `toLowerCase()` / `toUpperCase()` - Converts to lowercase or uppercase.

5. `equals(String)` - Checks if two strings are exactly the same.

6. `equalsIgnoreCase(String)` - Checks if two strings are equal, ignoring case.

7. `trim()` - Removes leading and trailing whitespace.

8. `replace(char oldChar, char newChar)` - Replaces characters in the string.

9. `contains(CharSequence sequence)` - Checks if the string contains the specified sequence.

10. `indexOf(String)` - Returns the index of the first occurrence of a substring.

11. `split(String regex)` - Splits the string into an array based on the delimiter.

12. `startsWith(String prefix)` - Checks if the string starts with the specified prefix.

13. `endsWith(String suffix)` - Checks if the string ends with the specified suffix.

```java
public class StringExample {
    public static void main(String[] args) {
        // Creating a string
        String str = "  Hello, World! Java is fun!  ";

        // 1. length()
        System.out.println("Length of string: " + str.length());

        // 2. charAt(index)
        System.out.println("Character at index 5: " + str.charAt(5));

        // 3. substring(startIndex, endIndex)
        System.out.println("Substring from index 7 to 12: " + str.substring(7, 12));
```

```java
// 4. toLowerCase() and toUpperCase()
System.out.println("Lowercase: " + str.toLowerCase());
System.out.println("Uppercase: " + str.toUpperCase());

// 5. equals() and equalsIgnoreCase()
String str2 = "hello, world! java is fun!";
System.out.println("Equals (case-sensitive): " + str.equals(str2));
System.out.println("Equals (ignore case): " + str.equalsIgnoreCase(str2));

// 6. trim()
System.out.println("Trimmed string: '" + str.trim() + "'");

// 7. replace(oldChar, newChar)
System.out.println("Replacing 'o' with 'O': " + str.replace('o', 'O'));

// 8. contains()
System.out.println("Does the string contain 'Java'? " + str.contains("Java"));
```

```java
        // 9. indexOf()
        System.out.println("Index of 'Java': " + str.indexOf("Java"));

        // 10. split() (Splitting string into words)
        String[] words = str.split(" ");
        System.out.println("Words in the string:");
        for (String word : words) {
            System.out.println(word);
        }


        // 11. startsWith() and endsWith()
        System.out.println("Starts with ' Hello': " + str.startsWith(" Hello"));
        System.out.println("Ends with 'fun! ': " + str.endsWith("fun! "));
    }
}
```

# OUTPUT:

```
Length of string: 30
Character at index 5:
Substring from index 7 to 12: World
Lowercase:    hello, world! java is fun!
Uppercase:    HELLO, WORLD! JAVA IS FUN!
Equals (case-sensitive): false
Equals (ignore case): true
Trimmed string: 'Hello, World! Java is fun!'
Replacing 'o' with 'O': HellO, WOrld! Java is fun!
Does the string contain 'Java'? true
```

```
Index of 'Java': 19
Words in the string:
Hello,
World!
Java
is
fun!
Does the string start with '  Hello': true
Does the string end with 'fun!   ': true
```