# THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

## SLIDES FOR THEORY LECTURES

KGiSL MicrCollege

Genius as a Service (GaaS)

# THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

## DAY-03

Welcome Section

## LECTURE

Watch before you Start

# Introduction to Object-Oriented Programming (OOP)

## Definition:

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects."
- These objects can contain data in the form of fields (attributes) and code in the form of methods (functions).
- OOP focuses on creating reusable, modular, and organized code by simulating real-world entities.

## Key Principles of OOP:

1. **Encapsulation:**
   - Wrapping data (attributes) and methods (behavior) inside a class to protect them from unauthorized access.
2. **Inheritance:**
   - Enabling a class to acquire the properties and behavior of another class.

# Introduction to Object-Oriented Programming (OOP)

1. **Polymorphism:**
   - Allowing methods to perform different tasks based on the context, achieved via method overloading or overriding.
2. **Abstraction:**
   - Hiding complex implementation details and showing only essential features of an object.

---

## Why Use OOP?

- Enhances code reusability and modularity.
- Makes software easier to maintain and extend.
- Simulates real-world entities for better problem modeling.

# Example

**Example: Real-Life Analogy of OOP**

- Class: A blueprint, e.g., a "Car."
- Object: An instance of the class, e.g., "Toyota Corolla."
- Attributes: Properties of the object, e.g., color, brand.
- Methods: Actions performed by the object, e.g., drive, stop.

---

## Classes and Objects

### Classes:

- A class is a blueprint for creating objects.
- It defines the structure (fields) and behavior (methods) that the objects of the class will have.

# Object-Oriented Programming (OOP) in Java

## Syntax:

```
class ClassName {
  // Fields (Attributes)
  dataType fieldName;

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

# Object-Oriented Programming (OOP) in Java

## Objects:

- An object is an instance of a class.
- Objects represent individual entities with their own unique attributes and behaviors.

---

## Syntax:

```
ClassName objectName = new ClassName();
```

# Example: Classes and Objects

**Code Example:**

```
// Define a class
class Car {
  // Fields (Attributes)
  String brand;
  String color;

  // Constructor
  Car(String brand, String color) {
    this.brand = brand;
    this.color = color;
  }
```

# Example: Classes and Objects

```
// Method (Behavior)
void drive() {
System.out.println(brand + " is driving!");
}
}

// Main class
public class Main {
  public static void main(String[] args) {
     // Create objects
     Car car1 = new Car("Toyota", "Red");
Car car2 = new Car("Honda", "Blue");
```

# Example: Classes and Objects

```
        // Access object properties and methods
        System.out.println(car1.brand + " is " + car1.color);
        car1.drive();

        System.out.println(car2.brand + " is " + car2.color);
        car2.drive();
    }
}
```

**Output:**

```
Toyota is Red
Toyota is driving!
Honda is Blue
Honda is driving!
```

# Combined Example: OOP with Multiple Objects

**Scenario: A program to manage multiple "Person" objects.**

**Code Example:**

```
// Define a Person class
class Person {
  String name;
  int age;

  // Constructor
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
```

# Combined Example: OOP with Multiple Objects

```
  // Method
  void introduce() {
    System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
  }
}

// Main class
public class Main {
  public static void main(String[] args) {
    // Create objects
    Person person1 = new Person("Alice", 25);
    Person person2 = new Person("Bob", 30);
```

# Combined Example: OOP with Multiple Objects

```
        // Call methods on objects
        person1.introduce();
        person2.introduce();
    }
}
```

**Output:**

Hi, I'm Alice and I'm 25 years old.

Hi, I'm Bob and I'm 30 years old.

# Object-Oriented Programming (OOP) in Java

## Benefits of Classes and Objects

- **Reusability:** A class can be reused to create multiple objects.
- **Data Security:** Encapsulation allows control over access to data.
- **Organization:** Code is modular, making it easier to maintain.

# Object-Oriented Programming (OOP) in Java

## Methods:

- A method is a block of code or collection of statements, or a set of code grouped together to perform a certain task or operation.
- It is used to achieve the reusability of code.
- We write a method once and use it many times.
- We do not require to write code again and again.
- It also provides the easy modification and readability of code, just by adding or removing a chunk of code.
- The method is executed only when we call or invoke it.

## Note:

- The most important method in java is **main()** method.

# Object-Oriented Programming (OOP) in Java

## Method Declaration:

- The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments.
- It has six components that are known as method header.

## Syntax:

```
public int sum (int a, int b)
{
    //method body
}
```

# Object-Oriented Programming (OOP) in Java

- public int sum(int a, int b)--->**Method Header**
- publi--->**Access specifier**
- int--->**Return Type**
- sum--->**Method Name**
- (int a, int b)--->**Parameter List**
- sum(int a, int b)--->**Method Signature**

## Types of Method:

2 types of methods in java

- Predefined method
- User-defined method

# Object-Oriented Programming (OOP) in Java

## Predefined method:

- In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods.
- It is also known as the standard library method or built-in method.

**Example:** length(), equals(), compareTo(), sqrt()

## Note:

- Each and every predefined method is defined inside a class.
- **print()** method is defined in the **java.io.PrintStream** class.

# Example:

## Example: predefined method

```
public class StringExample{
    public static void main(String[] args){
        String text = “Hello World”;
        System.out.println(“Text length: ”+text.length());
 }
}
```

**Output:**

Text length: 11

# Object-Oriented Programming (OOP) in Java

## User-defined method

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

### Example: User-defined method

```
public class Calculator{
 public static int add(int a, int b){
 return a+b;
 }
 public static void main(String[] args){
      int sum = add(10, 5);
      System.out.println("Sum: "+ sum);
 }
}
```

**Output:** Sum: 15

# Object-Oriented Programming (OOP) in Java

## How to call or invoke a user-defined method

- Once we have defined a method, it should be called.

```
java.util.Scanner;
public class Calculator{
    public static int add(int a, int b){
    return a+b;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter value1: ");
        int num1 = sc.nextInt();
        System.out.println("Enter value2: ");
        int num2 = sc.nextInt();
        add(num1,num3);
     }
}
```

# Object-Oriented Programming (OOP) in Java

## Static method:

- A method that has static keyword is known as static method.
- The main advantage of a static method is that we can call it without creating an object.

**Example:**

```
public class StaticMethodExample{
    static void display(){
     System.out.println("This is example of static method");
}
    public static void main(String[] args){
        display();
}
}
```

**Output:** This is example of static method

# Object-Oriented Programming (OOP) in Java

## Instance method:

- The method of the class is known as an **instance method.**
- It is a **non-static** method defined in the class.
- Before calling or invoking the instance method, it is necessary to create an object of its class.

# Object-Oriented Programming (OOP) in Java

**Example:**

```
public class Calculator{
        public static int add(int a, int b){
                s = a+b;
                return s;
         }
        public static void main(String[] args){
                Calculator cal = new Calculator();
                System.out.println("The sum is: "+ obj.add(10, 20));
         }
}
```

**Output:**

The sum is: 30

# Object-Oriented Programming (OOP) in Java

## 2 types of instance method:

- Accessor Method
- Mutator Method

## Accessor Method:

- The method that reads the instance variable is known as the accessor method.
- It is also known as getters.
- Method denotion: get()

**Example:**

```
public int getId()
{
    return Id;
}
```

# Object-Oriented Programming (OOP) in Java

## Mutator method:

- The method read the instance variable and also modify the values.
- It is also known as setters
- Method denotion: set()

**Example:**

```
Public void setRoll(int roll){
    this.roll = roll;
}
```

# Example: Accessor and mutator method

**Example:**

```
public class Register{
    private String name;
    private int regno;
    public String getName(){
     return name;
    }
    public String setName(String name){
        this.name = name;
    }
```

# Example: Accessor and mutator method

```
public int getRegno(){
    return regno;
}
public int setRegno(int regno){
    this.regno = regno;
}
}
```

# Object-Oriented Programming (OOP) in Java

## Default Method:

**Sample format**

```
public void display(){
    System.out.println(“This is default method”);
}
```

## Parameterized Method:

**Sample format**

```
public void display(int a, int b){
    System.out.println(a+b);
}
```

# Constructors

## Definition:

- A constructor is a special method in Java used to initialize objects.
- It is invoked automatically when an object is created.

---

## Key Features:

- Same name as the class.
- No return type (not even void).
- Used to assign initial values to object attributes.

# Constructors

## Types of Constructors:

- **Default Constructor:**
  - Automatically provided by Java if no other constructor is defined.
- **Parameterized Constructor:**
  - Allows passing arguments to initialize fields with specific values.

# Example for Default Constructor :

**Example:**

```
class Car {
  String brand;

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

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

# Example for Default Constructor :

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

**Output:**

Brand: Unknown

# Example for Parameterized Constructor :

**Example:**

```
class Car {
  String brand;

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

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

# Example for Default Constructor :

```
public class Main {
  public static void main(String[] args) {
    Car car = new Car("Toyota"); // Parameterized constructor is called
    car.display();
  }
}
```

**Output:**

Brand: Toyota

# Example for Multiple Constructors (Constructor Overloading):

**Example:**

```
class Car {
  String brand;
  String color;

  // Default constructor
  Car() {
     brand = "Unknown";
     color = "White";
  }
```

# Example for Multiple Constructors (Constructor Overloading):

```
    // Parameterized constructor
    Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Color: " + color);
    }
}
```

# Example for Multiple Constructors (Constructor Overloading):

```
public class Main {
  public static void main(String[] args) {
    Car car1 = new Car(); // Calls default constructor
    Car car2 = new Car("Toyota", "Red"); // Calls parameterized constructor

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

**Output:**

Brand: Unknown, Color: White

Brand: Toyota, Color: Red

# Access Specifiers or Modifiers

## Definition:

- Access modifiers in Java define the scope or visibility of classes, methods, and variables.
- They control where these elements can be accessed from.

## Types:

- Access modifiers
- Non-Access modifiers

# Access Specifiers or Modifiers

## Types of Access Modifiers

### 1. Private:

- Accessible only within the same class.
- Often used for encapsulating data.

### 2. Default (Package-Private):

- Accessible only within the same class.
- Often used for encapsulating data.

# Access Specifiers or Modifiers

## 3. Protected:

- Accessible within the same package and by subclasses in other packages.

## 4. Public:

- Accessible from anywhere.

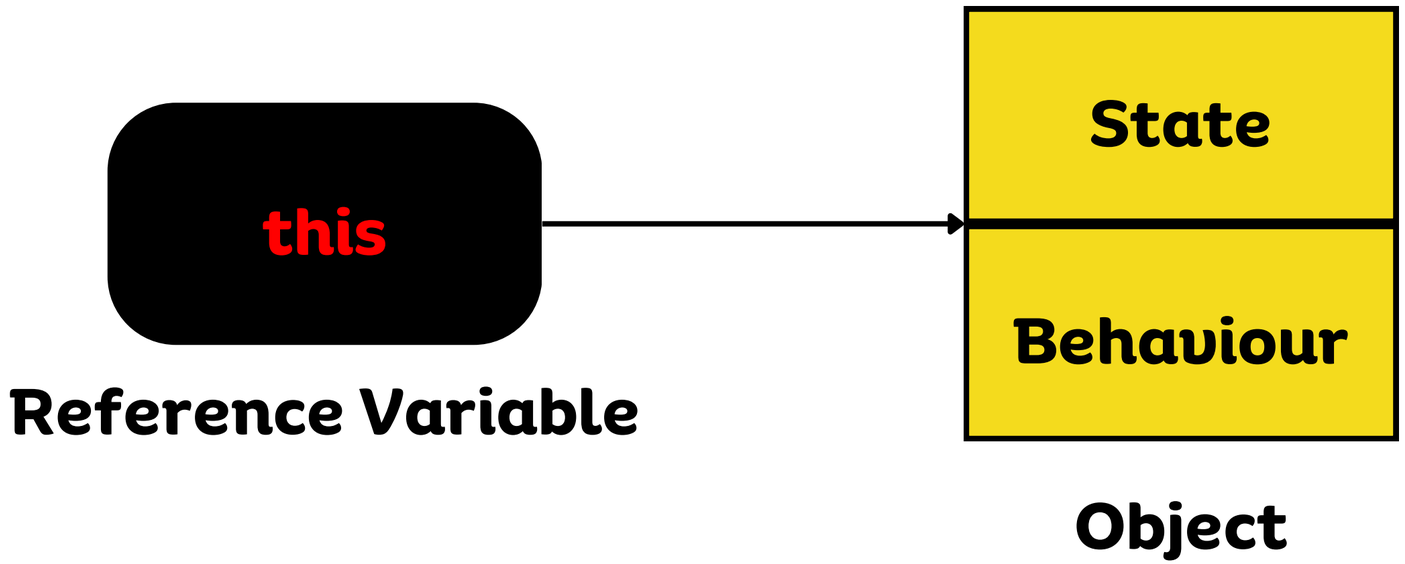
# Access Specifiers or Modifiers

| Modifier | Same Class | Same Package | Subclass | Outside Package |
|---|---|---|---|---|
| private | ✓ | ✗ | ✗ | ✗ |
| default | ✓ | ✓ | ✗ | ✗ |
| protected | ✓ | ✓ | ✓ | ✗ |
| public | ✓ | ✓ | ✓ | ✓ |

# this-keyword

## Definition:

- In Java, this is a **reference variable** that refers to the current object.



# this-keyword

## Usage of java this keyword:

### 6 Usage of java this keyword

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly).
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as an argument in the constructor call.
- this can be used to return the current class instance from the method.

# Example for Private modifier:

## 1. Private Modifier:

**Example:**

```
class BankAccount {
  private double balance; // Private field

  // Public method to access private field
  public double getBalance() {
     return balance;
  }
```

# Example for Private modifier:

```
    public void setBalance(double balance) {
        if (balance > 0) {
            this.balance = balance;
        } else {
            System.out.println("Invalid balance!");
        }
    }
}
```

# Example for Private modifier:

```
public class Main {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    account.setBalance(1000); // Allowed through setter
    System.out.println("Balance: " + account.getBalance()); // Allowed through getter
  }
}
```

# Example for Default modifier:

## 2. Default Modifier:

**Example:**

```
class Person {
  String name; // Default access

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

# Example for Default modifier:

```
public class Main {
  public static void main(String[] args) {
    Person person = new Person();
    person.name = "Alice"; // Accessible within the same package
    person.display();
  }
}
```

# Example for Protected modifier:

## 3. Protected Modifier:

**Example:**

```
class Animal {
  protected String species;

  protected void display() {
     System.out.println("Species: " + species);
  }
}
```

# Example for Protected modifier:

```
public class Main extends Animal {
  public static void main(String[] args) {
    Main dog = new Main();
    dog.species = "Dog"; // Accessible due to inheritance
    dog.display();
  }
}
```

# Example for Public modifier:

## 4. Public Modifier:

**Example:**

```
class HelloWorld {
  public void greet() {
     System.out.println("Hello, World!");
  }
}
```

# Example for Public modifier:

```
public class Main {
  public static void main(String[] args) {
    HelloWorld hello = new HelloWorld();
    hello.greet(); // Accessible from anywhere
  }
}
```

**Output:**

Hello, World!

# Non-Access Modifiers

## Types

- static
- final
- abstract
- synchronized
- volatile
- transient
- native
- strictfp

# Encapsulation

## Definition:

- Encapsulation is the process of bundling data (fields) and methods into a single unit (class) and restricting access to some components to protect object integrity.

---

## Key Features of Encapsulation:

- **Private Fields:** Data is kept private to prevent unauthorized access.
- **Public Methods:** Provide controlled access to private fields.
- **Getter and Setter Methods:** Used to retrieve and update private fields.

# Getters and Setters

## Definition:

- **Getters and Setters** are methods in Java used to retrieve (get) and modify (set) the values of private fields.
- These methods provide controlled access to the data, ensuring encapsulation.

---

## Why Use Getters and Setters?

- Protect sensitive data by making fields private.
- Enforce rules or validations when getting or setting a value.
- Decouple internal representation from external access.

# Example for Without Getters and Setters (Encapsulation Violated):

**Example:**

```
class Student {
  String name; // Public field

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

# Example for Without Getters and Setters (Encapsulation Violated):

```
public class Main {
  public static void main(String[] args) {
    Student student = new Student();
    student.name = "Alice"; // Direct access to the field
    student.display();
  }
}
```

**Output:**

```
Name: Alice
```

- **Issue:** No control over the value being assigned. For example, someone could assign student.name = null or an invalid value.

# Example Using Getters and Setters (Encapsulation Implemented):

**Example:**

```
class Student {
  private String name; // Private field

  // Getter method
  public String getName() {
     return name;
  }
```

# Example Using Getters and Setters (Encapsulation Implemented):

```
    // Setter method
    public void setName(String name) {
        if (name != null && !name.isEmpty()) { // Validation
            this.name = name;
        } else {
            System.out.println("Invalid name!");
        }
    }
}
```

# Example Using Getters and Setters (Encapsulation Implemented):

```
public class Main {
  public static void main(String[] args) {
    Student student = new Student();

    // Set the name using the setter
    student.setName("Alice");

    // Get the name using the getter
    System.out.println("Student Name: " + student.getName());
  }
}
```

**Output:**

Student Name: Alice

# Advanced Example with Multiple Fields:

**Example:**

```
class Employee {
  private int id;
  private String name;
  private double salary;

  // Getter and Setter for ID
  public int getId() {
     return id;
  }
```

# Advanced Example with Multiple Fields:

```
  public void setId(int id) {
     if (id > 0) { // Validation
        this.id = id;
     } else {
        System.out.println("Invalid ID!");
     }
  }

// Getter and Setter for Name
  public String getName() {
     return name;
  }
```

# Advanced Example with Multiple Fields:

```
public void setName(String name) {
    if (name != null && !name.isEmpty()) {
        this.name = name;
    } else {
        System.out.println("Invalid Name!");
    }
}

// Getter and Setter for Salary
public double getSalary() {
    return salary;
}
```

# Advanced Example with Multiple Fields:

```
    public void setSalary(double salary) {
        if (salary > 0) {
            this.salary = salary;
        } else {
            System.out.println("Invalid Salary!");
        }
    }
}
```

# Advanced Example with Multiple Fields:

```
public class Main {
  public static void main(String[] args) {
    Employee emp = new Employee();
    emp.setId(101);
    emp.setName("Bob");
    emp.setSalary(5000);

    System.out.println("Employee Details:");
    System.out.println("ID: " + emp.getId());
    System.out.println("Name: " + emp.getName());
    System.out.println("Salary: $" + emp.getSalary());
  }
}
```

# Advanced Example with Multiple Fields:

**Output:**

```
Employee Details:
ID: 101
Name: Bob
Salary: $5000.0
```

# Example for Encapsulation:

**Example:**

```
class BankAccount {
  private double balance; // Private field

  // Constructor to initialize balance
  BankAccount(double initialBalance) {
     this.balance = initialBalance;
  }

  // Getter method
  public double getBalance() {
     return balance;
  }
```

# Example for Encapsulation:

```
    // Setter method
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount!");
        }
    }
}
```

# Example for Encapsulation:

```
public class Main {
  public static void main(String[] args) {
    BankAccount account = new BankAccount(1000); // Initial balance

    System.out.println("Initial Balance: " + account.getBalance());
    account.deposit(500); // Deposit money
    System.out.println("Updated Balance: " + account.getBalance());
  }
}
```

**Output:**

Initial Balance: 1000.0

Deposited: 500.0

Updated Balance: 1500.0

# Static and non-static members

## Definition:

- In Java, static and non-static members refer to how variables and methods are associated with the class itself (static) or with instances (non-static).

## Static Members:

- **Static variables** are shared across all instances of the class. They are also called class variables.
- **Static methods** can be called without creating an instance of the class. They can only access other static variables and static methods.

## Key points:

- Declared using the static keyword.
- Associated with the class, not instances.
- Useful for data or behavior that should be shared across all instances.

# Example for static member:

**Example:**

```
class MyClass {
  static int count = 0; // static variable

  static void increment() { // static method
     count++;
  }
}

MyClass.increment(); // Accessing static method without creating an object
```

# Static and non-static members

## Non-static Members:

- **Non-static variables** are instance variables, meaning each object of the class has its own copy of the variable.
- **Non-static methods** require an instance of the class to be called and can access both static and non-static variables.

## Key points:

- Do not use the static keyword.
- Each instance has its own copy of the non-static variable.
- Can access both static and non-static members of the class.

# Example for static member:

**Example:**

```
class MyClass {
  int value = 0; // non-static variable

  void setValue(int value) { // non-static method
     this.value = value;
  }
}

MyClass obj = new MyClass();
obj.setValue(10); // Accessing non-static method via an object instance
```

# Static and non-static members

## Differences:

- **Memory Location:** Static members are stored in the class's memory area, while non-static members are stored in each object's memory.
- **Access:** Static members can be accessed directly via the class name or through instances, while non-static members require an object instance to access.

KGiSL MicroCollege

Genius as a Service (GaaS)

# THE COMPLETE CORE JAVA COURSE

PRESENTED BY,

KGM TECHNICAL TEAM

## DAY-03 COMPLETED 🎉