ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: Object Oriented Programming with Java

COURSE CODE: 1BCS302

SEMESTER: III

COURSE COORDINATOR: Mrs. DIVYA N

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

• To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
• To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
• To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

M1: To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.

M2: To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Program Educational Objective (PEO's)

- Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.
- Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.
- Our students will be well-prepared to excel in collaborative, multi-disciplinary environments, showcasing their versatility and skills.

Program Specific Outcomes (PSO's)

- To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.
- Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

Introducing Classes: Class Fundamentals, Declaring Objects, Assigning Object Reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection.

Methods and Classes: Overloading Methods, Objects as Parameters, Argument Passing, Returning Objects, Recursion, Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes.

CLASS FUNDAMENTALS

In Java, a **class** is a fundamental building block of object-oriented programming (OOP). It serves as a blueprint for creating objects (instances) and encapsulates data for the object and methods to manipulate that data.

Defining class in java

A class in Java is defined using the class keyword, followed by the class name. It can contain:

- **Attributes (Fields)**: These are variables that hold the state or properties of the object.
- **Methods**: Functions that define the behaviours or actions that the object can perform.
- **Constructors**: Special methods used to initialize new objects of the class.

Syntax

```java
public class ClassName {
    // Attributes (fields)
    int attribute1;
    String attribute2;

    // Constructor
    public ClassName(int param1, String param2) {
        this.attribute1 = param1;
        this.attribute2 = param2;
    }

    // Method
    public void methodName() {
        // Method body
    }
}
```

Key Points:

1. **Encapsulation**: Classes encapsulate data (attributes) and methods, promoting modularity and code reusability.
2. **Access Modifiers**: Classes can use access modifiers (like public, private, protected) to control the visibility of their members.
3. **Inheritance**: A class can inherit fields and methods from another class, enabling code reuse and establishing relationships between classes.
4. **Polymorphism**: Classes support polymorphism, allowing methods to be overridden and providing dynamic method dispatch.

Example Program

```java
public class ATME {
    // Attributes
    private String collegeName;
    private int establishedYear;

    // Constructor
    public ATME(String collegeName, int establishedYear) {
        this.collegeName = collegeName;
        this.establishedYear = establishedYear;
    }

    // Method to display college information
    public void displayInfo() {
        System.out.println("College Name: " + collegeName + ", Established Year: " + establishedYear);
    }

    // Main method to test the class
    public static void main(String[] args) {
        // Creating an instance of ATME
        ATME college = new ATME("Academy for Technical & Management Excellence College of Engineering", 2010);

        // Displaying college information
        college.displayInfo();
    }
}
```

Output:

College Name: Academy for Technical & Management Excellence College of Engineering, Established Year: 2004

DECLARING OBJECTS

In object-oriented programming (OOP), an **object** is an instance of a class. It is a fundamental building block that encapsulates data (attributes) and behaviors (methods) associated with that data.

An object can be defined as follows:

- **Instance of a Class:** An object is created from a class blueprint and represents a specific entity or concept defined by that class.
- **Encapsulation of State and Behavior:** Each object contains data in the form of fields (also known as attributes or properties) and methods (functions) that operate on that data, allowing for modular and organized code.
- **Identity:** Each object has a unique identity, which distinguishes it from other objects, even if they have the same state.

Key Characteristics of an Object:

1. **State**: The current values of the object's attributes. For example, in a Car object, the state could include the car's model, color, and year.
2. **Behavior**: The methods that define what the object can do or how it can interact with other objects. For instance, a Car object may have methods like drive(), stop(), or honk().
3. **Identity**: Each object has a distinct reference in memory, meaning it can be identified separately from other objects.

Example: Using the previous ATME class as an example, here's how we might create an object:

```
// Creating an instance of ATME
ATME college = new ATME("Academy for Technical & Management Excellence College of Engineering", 2010);
// Displaying college information
college.displayInfo();
```

ASSIGNING OBJECT REFERENCE VARIABLES

- **Null Reference:** If we declare an object reference variable but do not assign it an object, it will hold a null value until assigned.

  ClassName object = null; // No object assigned

- **Copying Objects:** To create a new object that is a copy of an existing one, we typically need to implement a copy constructor or a cloning method.

```
public class Student {
  private String name; // Attributes
  private int age;// Attributes

  // Constructor
  public Student(String name, int age) {
    this.name = name;
    this.age = age;
  }
  // Method to display student information
  public void displayInfo() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
  // Main method to test object reference assignment
  public static void main(String[] args) {
    // Creating a Student object and assigning it to student1
    Student student1 = new Student("XYZ", 20);

    // Assigning student1 to student2 (same object reference)
    Student student2 = student1;

    // Displaying information using both references
    student1.displayInfo(); // XYZ, 20
    student2.displayInfo(); // XYZ, 20

    // Modifying age using student2 reference
    student2.age = 21;

    // Displaying information again to show changes
    student1.displayInfo(); // XYZ, 21
    student2.displayInfo(); // XYZ, 21
  }
}
```

Example Program which demonstrates assigning object reference variables

Methods:

In Java, methods are blocks of code that perform specific tasks. They allow we to encapsulate functionality, making wer code more organized and reusable. Here's an overview of methods in Java, including their types, syntax, and examples.

Key Components of Methods

1. **Method Declaration**: The way a method is defined.
2. **Method Signature**: The name of the method along with its parameters.
3. **Return Type**: Specifies the type of value the method will return. If it doesn't return anything, the return type is void.
4. **Method Body**: The code that defines what the method does.

```
returnType methodName(parameterType1 parameter1, parameterType2 parameter2) {
    // Method body
    // Code to perform the task
    return value; // Optional, depending on returnType
}
```

Types of Methods

There are two types of methods 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. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

  Example:

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

- **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:

```
public class Demo
{
int a=10,b=20;
// user defined function
void displayaddition()
{
    System.out.println("Sum of " +a+" & " +b +": "+(a+b));
}

public static void main(String[] args)
{

Demo ob=new Demo();
// calling function
ob.displayaddition();

}
}
```

Output:

```
C:\TestJava>javac Demo.java

C:\TestJava>java Demo
Sum of 10 & 20: 30
```

Static Method

A method that has **static** keyword is known as **static** method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the main() method.

Methods and its types

- **No-Argument Method:** A method that does not take any parameters.
  Example Snippet:

```
void displayaddition()
{
    System.out.println("Sum of " +a+" & " +b +": "+(a+b));
}
```

- **Parameterized Method:** A method that takes one or more parameters.
  Example Snippet:

```
void displayaddition(int a, int b)
{
    System.out.println("Sum of " +a+" & " +b +": "+(a+b));
}
```

- **Return Method:** A method that returns a value.

Example Snippet:

```
int multiply(int a, int b)
{
    return a * b;
}
```

Constructors: Constructors in Java are special methods used to initialize objects when they are created. They are called automatically when an object of a class is instantiated. Here's a detailed overview of constructors, their types, syntax, and examples.

Key Characteristics of Constructors

1. **Same Name as Class:** A constructor must have the same name as the class in which it resides.
2. **No Return Type:** Constructors do not have a return type, not even void.
3. **Called Automatically:** They are invoked automatically when an object is created using the new keyword.
4. **Overloading:** We can have multiple constructors in a class (constructor overloading) with different parameters.

Default Constructor: A constructor that does not take any parameters. If no constructor is defined in the class, Java provides a default constructor automatically.

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```
// Default constructor (no-arg constructor)
public MyPoint()
{
    this.x = 0;
    this.y = 0;
}
```

Parameterized Constructor: A constructor that takes parameters to initialize the object with specific values.

The parameterized constructor is used to provide different values to distinct objects. However, we can provide the same values also.

```
// Overloaded constructor that takes x and y coordinates
public MyPoint(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Refer to Lab Program3 & Program4

this Keyword: In Java, the ***this*** keyword refers to the current instance of a class. It is used to differentiate between instance variables (class-level attributes) and parameters or variables with the same name inside methods or constructors. The ***this*** keyword has several important uses in Java, particularly in object-oriented programming.

Key Uses of the this Keyword

1. **Referring to Instance Variables:** this is used to resolve ambiguity when instance variables and method/constructor parameters have the same name.
2. **Calling Other Constructors:** We can use this() to call one constructor from another constructor in the same class (constructor chaining).
3. **Passing the Current Object:** this can be used to pass the current object as a parameter to another method.
4. **Returning the Current Object:** this can be used to return the current object from a method.

Using this to Refer to Instance Variables: When instance variables are shadowed by method parameters, this helps in referring to the instance variables.

```
public class Student {
    // Instance variables
    private String name;
    private int age;

    // Constructor with parameters having the same name as instance variables
    public Student(String name, int age) {
        // Use 'this' to refer to instance variables
        this.name = name;
        this.age = age;
    }

    // Method to display student information
    public void displayInfo() {
        System.out.println("Name: " + this.name + ", Age: " + this.age);
    }

    public static void main(String[] args) {
        // Create a Student object
        Student student = new Student("John", 22);

        // Display student information
        student.displayInfo();
    }
}
```

Output:

Name: John, Age: 22

Using this() to Call Another Constructor (Constructor Chaining): We can use this() to call one constructor from another constructor in the same class. This is known as constructor chaining.

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

    // Constructor 1
    public Student() {
        // Call another constructor using this()
        this("Unknown", 18);  // Calls the constructor below
    }

    // Constructor 2
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

    public static void main(String[] args) {
        // Create Student objects using different constructors
        Student student1 = new Student();         // Calls the default constructor
        Student student2 = new Student("Alice", 20);  // Calls the parameterized constructor

        // Display information
        student1.displayInfo(); // Output: Name: Unknown, Age: 18
        student2.displayInfo(); // Output: Name: Alice, Age: 20
    }
}
```

Output:

Name: Unknown, Age: 18

Name: Alice, Age: 20

Using this to Pass the Current Object as a Parameter: We can pass the current object as an argument to another method using this.

```
public class Student {
    private String name;

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

    // Method that takes a Student object as a parameter
    public void greet(Student student) {
        System.out.println("Hello, " + student.name);
    }

    // Method to call greet method using 'this'
    public void callGreet() {
        greet(this);  // Passes the current object
    }

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

Using this to Return the Current Object: We can use this to return the current object from a method, which allows for method chaining.

```
public class Student {
    private String name;

    public Student setName(String name) {
        this.name = name;
        return this;  // Returns the current object
    }

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

    public static void main(String[] args) {
        // Method chaining using 'this'
        Student student = new Student().setName("John");
        student.displayInfo();  // Output: Name: John
    }
}
```

Using this is crucial for writing clean, concise, and flexible object-oriented code in Java.

Garbage collection (GC) in Java is the process by which the Java Virtual Machine (JVM) automatically manages memory by reclaiming the memory that is no longer in use or that is unreachable by any part of the program. This helps prevent memory leaks and ensures that the application runs efficiently without manually deallocating memory (as in languages like C or C++).

Key Points About Garbage Collection

1. **Automatic Memory Management**: In Java, memory management is automatic, meaning programmers don't need to manually allocate or free memory. The JVM takes care of releasing memory when objects are no longer in use.
2. **Heap Memory**: Java objects are stored in a region of memory called the **heap**. The garbage collector reclaims memory from the heap that is no longer referenced by the program.
3. **Unreachable Objects**: An object becomes eligible for garbage collection when it is unreachable from any live thread or by any other active object in the application.

How Garbage Collection Works

1. **Reference Counting**: The JVM keeps track of the number of references to each object. When the reference count drops to zero (i.e., no part of the program refers to the object), the object becomes eligible for garbage collection.
2. **Mark-and-Sweep Algorithm**: The JVM uses a mark-and-sweep algorithm to identify and reclaim unreachable objects:
   - **Mark Phase**: The garbage collector scans through the heap and marks all objects that are reachable (i.e., objects still being used).
   - **Sweep Phase**: All unmarked objects (i.e., those that are unreachable) are cleared from memory.
3. **Generational Garbage Collection**: Modern garbage collectors in Java divide objects into generations:
   - **Weng Generation**: New objects are allocated here. When this generation fills up, a minor garbage collection is triggered.
   - **Old (Tenured) Generation**: Objects that survive multiple rounds of garbage collection in the weng generation are moved here. A full (major) garbage collection is performed when this generation fills up.
   - **Permanent Generation (Metaspace)**: Stores metadata about classes, methods, etc. (Note: In Java 8, the Permanent Generation was replaced by Metaspace.)

Eligibility for Garbage Collection

An object becomes eligible for garbage collection when:
1. No active part of the program can access it.
2. There are no references pointing to the object, or the references are set to null.

Key Methods Related to Garbage Collection

- **System.gc()**: Suggests the JVM to run the garbage collector, though it's not mandatory for the JVM to act immediately.
- **finalize()**: This method was invoked by the garbage collector before destroying the object. However, it is deprecated in Java 9+.

```
public class GarbageCollectionDemo {
   public static void main(String[] args) {
      // Creating an object of type Car
      Car car = new Car("Toyota", 2021);

      // The car object is now reachable and not eligible for GC
      car.displayInfo();

      // Setting car reference to null, making the object eligible for GC
      car = null;

      // Calling garbage collector explicitly (not necessary in most cases)
      System.gc(); // Suggests to JVM that it may perform garbage collection

      System.out.println("Garbage collection is called");
   }
}

class Car {
   private String model;
   private int year;

   public Car(String model, int year) {
      this.model = model;
      this.year = year;
   }

   public void displayInfo() {
      System.out.println("Model: " + model + ", Year: " + year);
   }

   // Finalize method is called just before object is garbage collected
   @Override
   protected void finalize() throws Throwable {
      System.out.println("Car object is being garbage collected");
   }
}
```

Example: Demonstrates Garbage Collection

Output:

Model: Toyota, Year: 2021

Garbage collection is called

Car object is being garbage collected

Overloading Methods:

Method Overloading is a feature in Java that allows a class to have more than one method with the same name, as long as their parameter lists (number or types of parameters) are different. Overloading makes programs readable and easy to understand by grouping similar operations under a single method name, differentiated by their parameters.

Key Points of Method Overloading:

1. **Same method name**: The overloaded methods must have the same name.
2. **Different parameter lists**: The number, type, or sequence of parameters must differ for each overloaded method.
3. **Return type does not matter**: Overloading cannot be based solely on the return type. The compiler differentiates methods by their parameter list, not the return type.

Example

```
public class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling overloaded methods
        System.out.println("Sum of two integers: " + calc.add(5, 10));     // Calls add(int, int)
        System.out.println("Sum of three integers: " + calc.add(5, 10, 15)); // Calls add(int, int, int)
        System.out.println("Sum of two doubles: " + calc.add(5.5, 10.3));   // Calls add(double, double)
    }
}
```

Output:

Sum of two integers: 15
Sum of three integers: 30
Sum of two doubles: 15.8

Rules for Method Overloading:

- **Parameter Type Change**: We can change the type of parameters to overload a method.
  public int sum(int a, int b) { }
  public double sum(double a, double b) { }
- **Number of Parameters Change**: We can change the number of parameters to overload a method.
  public int sum(int a, int b) { }
  public int sum(int a, int b, int c) { }
- **Order of Parameters**: We can change the order of parameters (as long as their types are different) to overload a method.
  public void display(int a, String b) { }
  public void display(String b, int a) { }

Advantages of Method Overloading:

- Increases readability: By using the same method name for similar operations, code becomes more intuitive.
- Code reusability: We don't have to create new method names for similar functionalities.
- Flexibility: It allows the method to handle different types or numbers of arguments.

Object as parameters: In Java, **objects can be passed as parameters** to methods, just like primitive data types (int, float, etc.). When we pass an object as a parameter, we are passing the reference to the object (not the actual object itself), meaning the method can modify the state of the object being passed.

- When an object is passed as a parameter, a copy of the reference is passed.
- The method can change the object's state (values of its attributes) within the method.
- Java uses pass-by-value for everything, but for objects, it's a reference to the object that is passed by value.

Example: Passing an Object as a Parameter

```
class Rectangle {
   int length;
   int width;

   // Constructor
   public Rectangle(int length, int width) {
      this.length = length;
      this.width = width;
   }

   // Method to calculate area
   public int calculateArea() {
      return length * width;
   }

   // Method that modifies the attributes of the Rectangle object
   public void modifyDimensions(Rectangle rect) {
      rect.length = 10;
```

```
        rect.width = 5;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a Rectangle object
        Rectangle myRectangle = new Rectangle(20, 15);

        // Display original dimensions and area
        System.out.println("Original Dimensions: " + myRectangle.length + "x" + myRectangle.width);
        System.out.println("Original Area: " + myRectangle.calculateArea());

        // Modify object dimensions by passing it to a method
        myRectangle.modifyDimensions(myRectangle);

        // Display modified dimensions and area
        System.out.println("Modified Dimensions: " + myRectangle.length + "x" + myRectangle.width);
        System.out.println("Modified Area: " + myRectangle.calculateArea());
    }
}
```

Output:

Original Dimensions: 20x15

Original Area: 300

Modified Dimensions: 10x5

Modified Area: 50

Uses of passing object as parameter

- **Reusability:** Methods can operate on any object of the class, increasing flexibility.
- **Modifying State:** We can change the state of objects by passing them to methods.
- **Efficiency:** Objects can be passed efficiently, as references are small and there is no need to create a copy of the entire object.

Argument Passing

In Java, **argument passing** refers to the process of providing input values (arguments) to methods when they are called. Java uses **pass-by-value** for all method calls, whether the arguments are primitive data types or objects.

Two Types of Argument Passing in Java:

1. **Passing Primitive Data Types**: For primitive types (like int, char, float), the actual value is passed to the method, and any changes made inside the method do not affect the original value outside the method.
2. **Passing Objects (Reference Data Types)**: When an object is passed to a method, the reference (or memory address) of the object is passed. Although the reference itself is passed by value, the method can modify the actual object's state (fields or properties) because both the caller and the method have a reference to the same object.

Primitive Data Type Argument Passing (Pass-by-Value):

When passing a primitive data type (like int, float, etc.), the value is copied into the method parameter. Any changes made to the parameter inside the method do not affect the original variable outside the method.

```
public class Main {
    public static void main(String[] args) {
        int num = 10;
        modifyPrimitive(num);
        System.out.println("Value of num after method call: " + num); // Value remains 10
    }

    // Method that tries to modify the primitive argument
    public static void modifyPrimitive(int n) {
        n = n + 5;
        System.out.println("Value of n inside method: " + n); // Value inside the method is 15
    }
}
```

Output:

Value of n inside method: 15

Value of num after method call: 10

Object (Reference Data Type) Argument Passing (Pass-by-Reference-Value):

When passing an object to a method, a copy of the **reference** to the object is passed. This means that the method can modify the object's fields, and the changes will be reflected outside the method.

```
class Box {
   int length;

   public Box(int length) {
      this.length = length;
   }

   // Method to modify object
   public static void modifyBox(Box box) {
      box.length = box.length + 5;
      System.out.println("Length inside method: " + box.length); // Length modified to 15
   }
}

public class Main {
   public static void main(String[] args) {
      Box myBox = new Box(10);
      modifyBox(myBox);
      System.out.println("Length after method call: " + myBox.length); // Length is 15
   }
}
```

Output:

Length inside method: 15

Length after method call: 15

Returning Objects: In Java, methods can return objects, just like they can return primitive data types. Returning objects from methods allows we to create new instances or modify existing ones, which can then be used by the calling method.

- A method can return any type of object (including custom objects) as a result.
- When returning an object, we are actually returning a reference to that object.
- The object returned can be a new object created within the method or an existing object that is passed and modified within the method.

Example: Returning a Newly Created Object

```
class Rectangle {
   int length;
   int width;

   // Constructor
   public Rectangle(int length, int width) {
      this.length = length;
      this.width = width;
   }

   // Method to return a new Rectangle object
   public static Rectangle createRectangle(int length, int width) {
      return new Rectangle(length, width); // Returning a new object
   }

   // Method to display rectangle dimensions
   public void displayDimensions() {
      System.out.println("Length: " + length + ", Width: " + width);
   }
}

public class Main {
   public static void main(String[] args) {
      // Call the method to get a new Rectangle object
      Rectangle myRectangle = Rectangle.createRectangle(15, 10);

      // Display the dimensions of the returned object
      myRectangle.displayDimensions();  // Output: Length: 15, Width: 10
   }
}
```

Output: Length: 15, Width: 10

Benefits of Returning Objects:

1. **Encapsulation**: We can control how objects are created or modified inside the method.
2. **Reuse**: Returned objects can be reused in the calling method.
3. **Flexibility**: We can return different types of objects based on conditions.

Recursion:

Recursion in Java is a technique where a method calls itself to solve smaller instances of the same problem. It is commonly used to solve problems that can be broken down into simpler, smaller sub-problems of the same type.

Key Points:

1. A **recursive method** must have a **base case** that stops the recursion, preventing infinite recursion.
2. Each recursive call should work towards reaching the base case.
3. Problems like calculating factorials, Fibonacci numbers, and solving the Towers of Hanoi are classic examples of recursion.

Structure of a Recursive Method:

- **Base Case**: The condition where the recursion ends.
- **Recursive Case**: The part where the method calls itself with a smaller or simpler argument.

Example: Finding factorial of a number

```
public class RecursionExample {

    // Recursive method to calculate factorial of a number
    public static int factorial(int n) {
        // Base case: factorial of 0 or 1 is 1
        if (n == 0 || n == 1) {
            return 1;
        }
        // Recursive case: n * factorial of (n-1)
        else {
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is: " + result);  // Output: 120
    }
}
```

Example: Fibonacci Series

```
public class FibonacciExample {

    // Recursive method to calculate Fibonacci numbers
    public static int fibonacci(int n) {
        // Base case: return n if n is 0 or 1
        if (n == 0 || n == 1) {
            return n;
        }
        // Recursive case: Fibonacci of (n-1) + Fibonacci of (n-2)
        else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }

    public static void main(String[] args) {
        int number = 6;
        int result = fibonacci(number);
        System.out.println("Fibonacci of " + number + " is: " + result);  // Output: 8
    }
}
```

Recursion vs. Iteration:

- **Recursion** is often more intuitive for problems that naturally fit the recursive model (like tree traversal, factorial, etc.).
- However, recursion can lead to performance issues like **stack overflow** if the recursion depth is too large. In such cases, **iteration** (using loops) can be more efficient.

Benefits of Recursion:

1. Simplifies problems that have a recursive nature.
2. Helps break down complex problems into smaller, more manageable sub-problems.

Drawbacks of Recursion:

1. **Overhead**: Each recursive call adds a new frame to the call stack, which can lead to memory overhead.
2. **Stack Overflow**: If the recursion depth is too high and there's no proper base case, it can cause a stack overflow error.
3. **Performance**: In some cases (like the Fibonacci example), recursion can lead to redundant calculations, reducing efficiency.

Access Control in Java refers to the mechanism of restricting access to the members (fields, methods, constructors) of a class. It is a part of Java's **encapsulation** principle, which ensures that sensitive data is hidden from external access and only allowed through controlled access points (like methods).

In Java, access control is achieved through **access modifiers**, which define the scope and visibility of classes, methods, and fields.

Access Modifiers:

1. **Private** (private):
   - The member is **accessible only within the class** where it is declared.
   - It provides the highest level of data protection.
2. **Default** (No Modifier):
   - The member is **accessible only within the same package**.
   - It is also called **package-private**.
3. **Protected** (protected):
   - The member is **accessible within the same package** and to **subclasses (even if in different packages)**.
   - It provides a wider access scope than private but still offers some protection.
4. **Public** (public):
   - The member is **accessible from anywhere** in the program, across all packages and classes.
   - It offers the least restriction.

Access Control Best Practices:

- Use private for sensitive data that should not be exposed outside the class.
- Use public methods as access points to manipulate or retrieve private data (getter and setter methods).
- Use protected when you want a member to be accessible to child classes.
- Use default when package-level access is sufficient and no cross-package access is required.

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

Example:

AccessControlExample.java

```
public class AccessControlExample {
    // Private member: accessible only within this class
    private String name;

    // Protected member: accessible within the package and in subclasses
    protected int age;

    // Public member: accessible from anywhere
    public String college;

    // Default access: accessible only within the same package
    String department;

    // Constructor to initialize data
    public AccessControlExample(String name, int age, String college, String department) {
        this.name = name;   // private member
        this.age = age;      // protected member
        this.college = college;  // public member
        this.department = department;  // default access
    }

    // Private method: only accessible within the class
    private void displayPrivate() {
        System.out.println("Name: " + name);
    }

    // Public method: accessible from anywhere
    public void displayInfo() {
        displayPrivate();  // calling private method within the same class
        System.out.println("Age: " + age + ", College: " + college + ", Department: " + department);
    }
}
```

Create a main class where AccessControlExample.java class members and member functions are accessed

AccessControlExample.java

```
public class TestAccess {
    public static void main(String[] args) {
        AccessControlExample student = new AccessControlExample
        ("John", 22, "XYZ University", "Computer Science");

        // Accessing public method
        student.displayInfo();  // Accessible

        // Accessing public field directly
        System.out.println("College: " + student.college);  // Accessible

        // Accessing protected field
        // System.out.println("Age: " + student.age);
        // ERROR: Cannot access directly outside package unless subclass

        // Accessing private field or method
        // student.displayPrivate();
        // ERROR: Cannot access private method outside the class
        // System.out.println("Name: " + student.name);
        // ERROR: Cannot access private field
    }
}
```

Understanding Static:

In Java, the static keyword is used for memory management and helps define class-level elements (fields, methods, or nested classes) that are shared among all instances of a class. When a member is marked static, it belongs to the class itself, not to any specific instance, and can be accessed without creating an object of the class.

Key Concepts of static:

1. **Static Fields (Variables)**:
    - A static variable is shared across all instances of the class.
    - It is allocated memory only once when the class is loaded.
    - All objects of the class share the same copy of the static variable.
2. **Static Methods**:
    - A static method belongs to the class, not to an object.
    - It can be called using the class name (e.g., ClassName.methodName()), without needing an object.
    - Static methods can only access static variables and other static methods directly, since they do not have access to instance-level data (non-static members).
3. **Static Block**:
    - A static block is used to initialize static data members and is executed only once, when the class is loaded.
4. **Static Nested Classes**:
    - A nested class can be made static if it does not need access to the outer class's instance members.

Example: Demonstrating Static keyword usage

```
public class ATMECollege {
   // Static field
   static String collegeName = "ATME College of Engineering";

   // Instance field (non-static)
   int studentID;

   // Constructor
   public ATMECollege(int studentID) {
      this.studentID = studentID;
   }

   // Static method
   static void changeCollegeName(String newCollegeName) {
      collegeName = newCollegeName;  // Accessing static field
   }

   // Instance method
   public void displayStudentInfo() {
      System.out.println("Student ID: " + studentID + ", College: " + collegeName);
   }
}

public class TestStatic {
   public static void main(String[] args) {
      // Accessing static field directly using class name
      System.out.println(ATMECollege.collegeName);  // Output: ATME College of Engineering

      // Creating objects of the class
      ATMECollege student1 = new ATMECollege(101);
      ATMECollege student2 = new ATMECollege(102);

      // Calling instance method
      student1.displayStudentInfo();  // Output: Student ID: 101, College: ATME College of
Engineering
      student2.displayStudentInfo();  // Output: Student ID: 102, College: ATME College of
Engineering

      // Calling static method to change the static field
      ATMECollege.changeCollegeName("XYZ Institute of Technology");

      // Displaying the updated static field for both objects
      student1.displayStudentInfo();  // Output: Student ID: 101, College: XYZ Institute of
Technology
      student2.displayStudentInfo();  // Output: Student ID: 102, College: XYZ Institute of
Technology
   }
}
```

Using Final Keyword:

In Java, the final keyword is used to define constants and prevent modifications. It can be applied to variables, methods, and classes, and has different implications depending on where it is used. Here's a breakdown of its uses:

Final Variables

- **Definition**: A variable declared with the final keyword cannot be reassigned after it has been initialized.
- **Use Case**: To create constants that should not change once assigned.

Example:

```
public class FinalVariableExample {
    final int MAX_VALUE = 100;  // Final variable

    public void changeValue() {
        // MAX_VALUE = 200;  // This will cause a compile-time error
        System.out.println("MAX_VALUE: " + MAX_VALUE);
    }
}
```

Final Methods

- **Definition**: A method declared as final cannot be overridden by subclasses.
- **Use Case**: To ensure that a particular method's behavior remains unchanged in subclasses

```
class Parent {
    final void display() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // Attempting to override the final method will cause a compile-time error
    // void display() {
    //     System.out.println("Trying to override.");
    // }
}
```

Final Classes

- **Definition**: A class declared as final cannot be subclassed.
- **Use Case**: To create immutable classes or to prevent inheritance for security or design reasons.

```
final class ImmutableClass {
    private final int value;

    public ImmutableClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

// Attempting to extend the final class will cause a compile-time error
// class ExtendedClass extends ImmutableClass {
// }
```

Introducing Nested and Inner Classes

In Java, nested classes and inner classes are used to logically group classes that are only used in one place, which increases encapsulation and readability. Here's an introduction to both concepts:

Nested Classes

A **nested class** is a class defined within another class. It can be static or non-static.

Static Nested Class

- **Definition**: A static nested class can be instantiated without an instance of the outer class. It can access static members of the outer class.
- **Use Case**: To logically group classes that are only relevant to the outer class.

Example:

```java
public class OuterClass {
    static String outerStaticField = "Outer Static Field";

    static class StaticNestedClass {
        void display() {
            System.out.println("Accessing: " + outerStaticField);
        }
    }
}

public class TestStaticNestedClass {
    public static void main(String[] args) {
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        nestedObject.display();  // Output: Accessing: Outer Static Field
    }
}
```

Inner Classes

An **inner class** is a non-static nested class. It can access both static and non-static members of the outer class.

Non-Static Inner Class

- **Definition**: An inner class requires an instance of the outer class to be instantiated. It can access all members (including private) of the outer class.
- **Use Case**: To associate a class logically with its enclosing class.

Advantages of Using Nested and Inner Classes

1. **Encapsulation**: Helps in logically grouping classes and controlling visibility.
2. **Readability**: Makes code easier to read and understand.
3. **Access to Outer Class Members**: Inner classes can access all members of the outer class, enhancing functionality.

Example:

```
public class OuterClass {
    private String outerField = "Outer Field";

    class InnerClass {
        void display() {
            System.out.println("Accessing: " + outerField);
        }
    }
}

public class TestInnerClass {
    public static void main(String[] args) {
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
        innerObject.display();  // Output: Accessing: Outer Field
    }
}
```