DAY 03

(18 – 07 – 2025)

TOPICS COVERED

- ✓ Naming Conventions
- ✓ Class and Objects
- ✓ Access Specifiers
- ✓ Java Methods
- ✓ Constructors
- ✓ Default Constructor
- ✓ Parameterless Constructors
- ✓ Parameterized Constructors
- ✓ Copy Constructor
- ✓ this keyword in Java

Naming Conventions

Naming conventions in Java are **standard rules** that developers follow to make the code **readable, consistent, and professional**. Java is a case-sensitive language, so naming conventions help avoid confusion and errors.

Standard Java Naming Rules:

Element	Convention	Example
Class Name	PascalCase (Starts with Capital)	StudentDetails
Method Name	camelCase (starts lowercase)	calculateTotal()
Variable Name	camelCase	studentName
Constant Name	ALL_CAPS with underscores	MAX_VALUE
Package Name	Lowercase only, dot-separated	com.myapp.service

Control Statements in Java

Introduction to Control Statements :

In Java, **control statements** are used to control the flow of execution of the program based on certain conditions or loops.

They help us decide **what part of code should execute next**.

There are mainly **3 types of control statements** in Java:

1. **Decision-making statements**
2. **Looping statements**
3. **Jumping statements**

These control structures are very important to write **logical programs** like condition checking, repetitive tasks, skipping sections, and exiting loops.

Types of Control Statements

Category	Statements
Decision-Making	if, if-else, if-else-if, switch
Looping	for, while, do-while
Jumping	break, continue, return

Let’s now learn each of them with examples.

Decision-Making Statements

1. if Statement

-> The if statement executes a block of code **only if the condition is true**.

```
int age = 20;
if(age > 18) {
   System.out.println("Eligible to vote");
}
```

2. if-else Statement

-> Used when we want to execute **one block for true** and another **for false** condition.

```
int age = 15;
if(age >= 18) {
   System.out.println("Adult");
} else {
   System.out.println("Minor");
}
```

3. if-else-if Ladder

-> Used to check **multiple conditions** in a sequence.

```
int marks = 85;
if(marks >= 90) {
   System.out.println("Grade A");
} else if(marks >= 75) {
   System.out.println("Grade B");
} else {
   System.out.println("Grade C");
}
```

4. switch Statement

-> Used to replace multiple if-else-if when checking **equality**.

```
int day = 3;
switch(day) {
   case 1: System.out.println("Monday"); break;
   case 2: System.out.println("Tuesday"); break;
   case 3: System.out.println("Wednesday"); break;
   default: System.out.println("Invalid Day");
}
```

Looping Statements

Looping is used when we want to **execute a block of code multiple times**.

1. for Loop

-> Best used when **number of iterations is known**.

```
for(int i = 1; i <= 5; i++) {
   System.out.println("Hello");
}
```

2. while Loop

-> Used when the number of iterations is **not known in advance**.

```
int i = 1;
while(i <= 5) {
   System.out.println(i);
   i++;
}
```

3. do-while Loop

-> Same as while, but it **executes at least once**.

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while(i <= 5);
```

Note: do-while always runs at least once even if the condition is false.

Jumping Statements

-> Used to **jump or skip** part of the program's execution.

1. break Statement

-> Used to **exit from a loop or switch** early.

```
for(int i = 1; i <= 5; i++) {
   if(i == 3)
      break;
   System.out.println(i);
}
```

Output:

```
1
2
```

2. continue Statement

-> Used to **skip current iteration** and continue to the next one.

```
for(int i = 1; i <= 5; i++) {
   if(i == 3)
      continue;
   System.out.println(i);
}
```

Output:

```
1
2
4
5
```

3. return Statement

-> Used to **exit from the method** and optionally return a value.

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

Class and Object in Java

What is a Class?

- A **class** is a **blueprint** or **template** that defines the **structure and behavior** (data and methods) of an object.
- It is **not a real entity**; it is just a logical structure.

- A class contains **fields (variables)** and **methods (functions)**.

Syntax:

```
class ClassName {
   // fields (variables)
   // methods
}
```

Example:

```
class Student {
   int id;                  // Data members (variables)
   String name;
   void display() {     // Method
      System.out.println(id + " " + name);
   }
}
```

What is an Object?

- An **object** is an **instance of a class**.
- It is a **real entity** that occupies memory.

- We create an object using the new keyword.

Syntax:

```
ClassName obj = new ClassName();
```

Example :

```
public class TestStudent {
   public static void main(String[] args) {
      Student s1 = new Student();  // Creating object
      s1.id = 101;
      s1.name = "Velayutham";
      s1.display();
   }
}
```

Output:

101 Velayutham

Access Specifiers in Java

Access Specifiers (also called **Access Modifiers**) in Java define **how accessible** a class, method, or variable is from different parts of the code. They help in **encapsulation** and maintaining **security** of code.

Types of Access Specifiers:

Specifier	Accessible Within	Keyword
private	Only within the same class	private
default	Same class and **same package only**	No keyword
protected	Same package + child classes (subclass)	protected
public	Accessible from **anywhere**	public

Example:

```
public class Sample {
   private int a = 10;       // Only inside this class
   protected int b = 20;     // Accessible in subclass
   public int c = 30;        // Accessible everywhere
}
```

- Use private for internal logic.
- Use protected when you want inheritance access.
- Use public for shared APIs or utilities.
- Leave it blank (default) if it's used only within the same package.

Java Method

What is a Method?

- A **method** in Java is a block of code that performs a specific task. It helps avoid code repetition and improves reusability.
- A method is executed only when it is called.

Syntax:

```
returnType methodName(parameter1, parameter2, ...) {
   // method body
}
```

Example:

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

Types of Methods

1. **Predefined Methods** – Already available in Java (e.g., `System.out.println()`)
2. **User-defined Methods** – Created by the programmer

Static Methods:

A **static method** belongs to the class rather than an object. It can be called **without creating an object**.

```
public class Example {
    static void show() {
        System.out.println("Static Method");
    }

    public static void main(String[] args) {
        show(); // no need to create object
    }
}
```

Output:

```
Static Method
```

Constructors in Java

- **What is a Constructor?**

A **constructor** is a special method that is called when an object is created. It has **no return type** and the **same name as the class**.

Syntax:

```
class ClassName {
    ClassName() {
        // constructor body
    }
}
```

Example:

```
class Student {
    Student() {
        System.out.println("Constructor called");
```

```
    }
}
```

Key Points:

- Automatically called when object is created
- Used to initialize objects
- Can be **default, parameterized, or copy constructor.**

Types of Constructors

1. Default Constructor:

-> A constructor with **no parameters**, provided by Java if none is written.

```
class A {
    A() {
        System.out.println("Default Constructor");
    }
}
```

2. Parameterless Constructor (User-defined) :

-> When you write your own constructor with **no parameters**.

```
class A {
    A() {
        System.out.println("User-defined No-Arg Constructor");
    }
}
```

3. Parameterized Constructor:

-> Takes arguments to initialize object values.

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

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

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

Copy Constructor and `this` Keyword

Copy Constructor

➔ A constructor that copies values from one object to another.

```
class Student {
   String name;

   Student(String n) {
      name = n;
   }

   Student(Student s) {
      name = s.name;
   }
}
```

this Keyword in Java :

➔ `this` refers to the **current object** of the class.

Uses of `this`:

1. Refer instance variables
2. Call current class constructor
3. Pass current object as parameter

Example:

```
class Student {
    String name;

    Student(String name) {
        this.name = name; // 'this' refers to current object
    }
}
```

Concept	Purpose
Method	Reusable block of code
Static Method	Called without object
Method Overloading	Same method name, different params
Constructor	Initializes object
Default Constructor	No arguments, auto-generated
Parameterized Constructor	Initializes with given values
Copy Constructor	Clones another object
this keyword	Refers to current object

Type Casting

Type Casting is the process of **converting one data type into another**. It is commonly used when we want to convert:

- **Primitive types** (e.g., int to float)
- **Object references** (upcasting/downcasting)

---

1. Primitive Type Casting

There are two types:

- **a) Implicit Casting (Widening)**

  Java automatically converts a smaller type to a larger type.

  int a = 10;

  double b = a; // int to double (automatic)

- **b) Explicit Casting (Narrowing)**

  Manually converting a larger type to a smaller type.

  double a = 10.5;

  int b = (int) a; // double to int (manual)

---

2. Reference Type Casting :

Used in inheritance between **parent and child classes**.

Upcasting (Implicit)

Converting a **subclass object to a superclass reference**.

➔ Done automatically
➔ Safe and common.

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

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

public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();  // Upcasting
        a.sound();  // Only parent methods accessible
    }
}
```

Downcasting (Explicit)

Converting a **superclass reference back to a subclass**.

➔ Must be done manually
➔ Can throw ClassCastException if not handled carefully

```
Animal a = new Dog();  // Upcasting
Dog d = (Dog) a;       // Downcasting
d.bark();              // Now subclass method accessible
```

Type	Conversion	Automatic	Safe?
Widening	int → long → float	Yes	Yes
Narrowing	double → int	No	Risky
Upcasting	Dog → Animal	Yes	Safe
Downcasting	Animal → Dog	No	Risky
