

ATME COLLEGE OF ENGINEERING

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



A T M E
College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING-CYBER SECURITY



LECTURE NOTES

COURSE: Object Oriented Programming with Java

COURSE CODE: 1BCS302

SEMESTER: III

COURSE COORDINATOR: Dr.Shwetha G K

(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 nurture competent and ethical cyber security professionals capable of contributing to secure and resilient digital ecosystems.

Mission:

M1: To impart strong foundational knowledge in cyber security through effective teaching–learning practices and a continuously evolving curriculum that addresses emerging cyber threats and technologies.

M2: To promote research, innovation, and industry collaboration in emerging areas of cyber security to address real-world digital security challenges.

M3: To instill professional ethics, teamwork, leadership qualities, and a sense of social responsibility in addressing real-world cyber challenges.

M4: To foster a culture of lifelong learning among students by encouraging higher studies, professional development, and active engagement with the global cyber security community.

Program Educational Objectives:

PEO1 –Excel as cyber security professionals or entrepreneurs by applying strong fundamentals of cyber security engineering.

PEO2 – Pursue higher studies, research, and professional certifications while engaging in lifelong learning to keep pace with evolving cyber technologies and threats.

PEO3 – Exhibit leadership, teamwork, professional ethics, and social responsibility while addressing cyber security challenges in multidisciplinary environments.

Program Specific Outcomes:

PSO1 – Apply principles of computer science, programming, algorithms, operating systems, and computer networks to design and develop efficient and reliable computing solutions.

PSO2 – Identify, analyze, and mitigate cyber threats and vulnerabilities using techniques related to network security, ethical hacking, penetration testing, and digital forensics.

PSO3 – Apply cyber security policies, cyber laws, risk management practices, and incident response strategies to ensure the protection and resilience of digital systems.

Enumerations, Type Wrappers and Autoboxing: Enumerations (Enumeration Fundamentals, The values() and valueOf() Methods), Type Wrappers (Character, Boolean, The Numeric Type Wrappers), Autoboxing (Autoboxing and Methods, Autoboxing/Unboxing Occurs in Expressions, Autoboxing/Unboxing Boolean and Character Values).

Generics: A Simple Generics Example, A Generic Class with Two Type Parameters, The General Form of a Generic Class.

Enumerations in Java

An **enumeration** (enum) is a special data type that represents a group of constants. It is a type-safe alternative to int constants and makes the code more readable and maintainable.

Enumeration Fundamentals

- Defined using the enum keyword.
- Each enum constant is an object of the enum type.
- Enums are implicitly **final** and **static**.

Syntax:

```
enum EnumName {  
    CONSTANT1, CONSTANT2, CONSTANT3;  
}
```

Example:

```
enum Day  
{  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
}  
  
public class EnumExample {  
    public static void main(String[] args) {  
        Day today = Day.FRIDAY; System.out.println("Today is: " + today);  
        // Loop through all constants  
        for (Day day : Day.values()) { System.out.println(day);  
    }  
}
```

} }Output:

```
C:\TestJava>javac EnumExample.java  
C:\TestJava>java EnumExample  
Today is: FRIDAY  
MONDAY  
TUESDAY  
WEDNESDAY  
THURSDAY  
FRIDAY  
SATURDAY  
SUNDAY
```

values() and valueOf() Methods

values() Method

- Returns an array of all enum constants.
- Useful for iterating over constants.

valueOf(String name) Method

- Converts a string to the corresponding enum constant.
- Throws IllegalArgumentException if the string does not match an enum constant.

Example:

```
enum Color {  
    RED, GREEN, BLUE;  
}  
  
public class EnumExample {  
    public static void main(String[] args) {  
        // Using values()  
        for (Color color : Color.values()) {  
            System.out.println(color);  
        }  
  
        // Using valueOf()  
        Color favoriteColor = Color.valueOf("RED");  
        System.out.println("Favorite color: " + favoriteColor);  
    }  
}
```

Output:

```
C:\TestJava>javac EnumExample.java  
  
C:\TestJava>java EnumExample  
RED  
GREEN  
BLUE  
Favorite color: RED
```

Type Wrappers

Primitive types (like int, double, char, etc.) have corresponding wrapper classes in the java.lang package. These wrapper classes allow primitives to behave as objects.

Common Wrapper Classes

Primitive Type	Wrapper Class
int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

Features:

1. Provide methods for manipulating and converting primitive values.
2. Useful in **collections** like ArrayList, which only store objects.

Common Type Wrappers

Character Wrapper

- Used for manipulating char values.

Example:

```
public class CommonTypeWrappersExample {  
    public static void main(String[] args) {  
        char c = 'A';  
        System.out.println(Character.isLetter(c)); // true  
        System.out.println(Character.isDigit(c)); // false  
    }  
}
```

Output:

```
C:\TestJava>javac CommonTypeWrappersExample.java  
  
C:\TestJava>java CommonTypeWrappersExample  
true  
false
```

Boolean Wrapper

- Used for boolean values.

Example:

```
public class CommonTypeWrappersExample {  
    public static void main(String[] args) {  
        Boolean b = Boolean.valueOf(true);  
        System.out.println(b); // true  
    }  
}
```

Output:

```
C:\TestJava>javac CommonTypeWrappersExample.java  
C:\TestJava>java CommonTypeWrappersExample  
true
```

Numeric Type Wrappers

- Includes Integer, Double, Float, etc.
- Provide methods for parsing and manipulating numbers.

Example:

```
public class CommonTypeWrappersExample {  
    public static void main(String[] args) {  
        String num = "123";  
        int number = Integer.parseInt(num); // Convert string to int  
        System.out.println(number * 2); // 246  
    }  
}
```

Output:

```
C:\TestJava>javac CommonTypeWrappersExample.java  
C:\TestJava>java CommonTypeWrappersExample  
246
```

Autoboxing and Unboxing**What is Autoboxing?**

Autoboxing refers to the automatic conversion of a primitive type into its corresponding wrapper class object.

- Automatic Conversion: The Java compiler automatically converts a primitive type to its corresponding wrapper class when necessary.
- Used in Collections: Since collections like ArrayList only accept objects (not primitives), autoboxing allows you to use primitive types directly in collections.

Unboxing is the reverse process of autoboxing, where an **object of a wrapper class** is automatically converted into its corresponding **primitive type**.

- **Automatic Conversion Back:** The Java compiler automatically converts a wrapper class object to a primitive type when necessary.
- **Used in Expressions:** Unboxing allows wrapper objects to be used in arithmetic and comparison expressions as primitives.

Example:

```
public class AutoboxingExample {  
    public static void main(String[] args) {  
        Integer num = 10; // Autoboxing  
        int result = num + 5; // Unboxing  
        System.out.println("Result: " + result);  
    }  
}
```

Output: Result: 15

Autoboxing in Methods

Autoboxing occurs when passing primitives to methods expecting wrapper objects.

Example:

```
import java.util.ArrayList;  
  
public class AutoBoxingExample {  
    public static void main(String[] args) {  
        ArrayList<Integer> list = new ArrayList<>();  
        list.add(10); // Autoboxing  
        System.out.println("List: " + list);  
    }  
}
```

Output:

```
C:\TestJava>javac AutoBoxingExample.java  
  
C:\TestJava>java AutoBoxingExample  
List: [10]
```

Autoboxing/Unboxing in Expressions

Java automatically handles conversions in arithmetic and logical expressions.

Example:

```
public class AutoboxingExample{  
    public static void main(String[] args) {  
        Integer a = 100;  
        Integer b = 200;  
        int sum = a + b; // Unboxing during addition  
        System.out.println("Sum: " + sum);  
    }  
}
```

Output:

```
C:\TestJava>javac AutoboxingExample.java  
  
C:\TestJava>java AutoboxingExample  
Sum: 300
```

Autoboxing/Unboxing Boolean and Character Values

Example:

```
public class AutoboxingExample{
    public static void main(String[] args) {
        Boolean flag = true; // Autoboxing
        if (flag) { // Unboxing in conditional
            System.out.println("Flag is true.");
        }
    }
}
```

Output:

```
C:\TestJava>java AutoboxingExample
Flag is true.
```

Summary of Key Features:

Feature	Description
Automatic Conversion	Autoboxing: Primitive → Wrapper, Unboxing: Wrapper → Primitive
Simplified Code	Reduces the need for explicit boxing/unboxing operations
Works in Expressions	Autoboxing/unboxing occurs automatically in arithmetic/logical operations
Method Calls	Enables primitive and wrapper class compatibility in method parameters
Collections Compatibility	Enables the use of primitives in collections (e.g., ArrayList<Integer>)
Supports All Wrapper Classes	All primitive types have corresponding wrapper classes supporting autoboxing/unboxing
Performance Considerations	Small performance overhead due to conversion, important for high-performance code

Generics in Java

Introduction

Generics is a feature introduced in **Java 5** that enables classes, interfaces, and methods to operate on **different data types** while providing **compile-time type safety**. By using generics, programmers can write reusable and flexible code that works with multiple types without sacrificing type checking.

Advantages of Generics

- Provides **compile-time type checking**.
- Eliminates the need for **explicit type casting**.
- Reduces the possibility of **ClassCastException**.
- Promotes **code reusability** and **type safety**.
- Improves readability and maintainability of programs.

A Simple Generics Example

A generic class is declared using a **type parameter**, commonly represented by T. The type parameter acts as a placeholder for the actual data type specified when an object is created.

Syntax

```
class Box<T> {  
    T value;  
  
    void setValue(T value) {  
        this.value = value;  
    }  
  
    T getValue() {  
        return value;  
    }  
}
```

Program

```
class Box<T> {  
    T value;  
  
    void setValue(T value) {  
        this.value = value;  
    }  
  
    T getValue() {  
        return value;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
  
        Box<Integer> intBox = new Box<Integer>();  
        intBox.setValue(10);  
        System.out.println(intBox.getValue());  
  
        Box<String> strBox = new Box<String>();  
        strBox.setValue("Hello");  
        System.out.println(strBox.getValue());  
    }  
}
```

Output

10

Hello

Explanation

- T is a **type parameter**.
- Box<Integer> stores only integer values.
- Box<String> stores only string values.
- The same class can be reused for different data types without creating separate classes.

A Generic Class with Two Type Parameters

A generic class can contain **multiple type parameters**. The type parameters are separated by commas within the angle brackets.

Syntax

```
class Pair<K, V> {  
    K key;  
    V value;  
}
```

Program

```
class Pair<K, V> {  
    K key;  
    V value;  
  
    Pair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    void display() {  
        System.out.println(key + " : " + value);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
  
        Pair<Integer, String> p1 =  
            new Pair<Integer, String>(1, "Apple");  
  
        Pair<String, Double> p2 =  
            new Pair<String, Double>("Price", 99.5);  
  
        p1.display();  
        p2.display();  
    }  
}
```

Output

1 : Apple

Price : 99.5

Explanation

- K represents the **key type**.
- V represents the **value type**.
- The class can store two different data types simultaneously.
- Common examples include collections such as Map<K, V>.

The General Form of a Generic Class

The general declaration of a generic class consists of a class name followed by one or more type parameters enclosed in angle brackets.

General Syntax

```
class ClassName<T> {
```

```
    T variable;
```

```
    ClassName(T value) {  
        variable = value;  
    }
```

```
    T getValue() {  
        return variable;  
    }  
}
```

Program

```
class GenericClass<T> {
```

```
    T data;
```

```
    GenericClass(T data) {  
        this.data = data;  
    }
```

```
    void show() {  
        System.out.println(data);  
    }  
}
```

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

```
        GenericClass<Integer> g1 =  
            new GenericClass<Integer>(100);
```

```
        GenericClass<String> g2 =  
            new GenericClass<String>("Java");
```

```
        g1.show();  
        g2.show();
```

```
    }  
}
```

Output

100

Java