



KAMALDNP

kamaldnp.com • IT Training, Varanasi

PDF created by Raj Kamal using AI

Short Notes — Object Oriented Programming Using Java

BTEUP Diploma in Computer Science & Engineering • 5th Semester • Short Notes
Session 2026-27

Short Revision Notes — prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus. This is NOT a BTEUP board paper or publication.

These **short revision notes** cover all five units of Object Oriented Programming Using Java from the official BTEUP 2026-27 syllabus, in syllabus order. Each point gives the definition, key difference and small example a short answer needs.

This subject is a **practicum** in the 2026-27 scheme, so these notes target its **theory (lecture) part, lab work and viva** together. Revise one unit, type its programs in the lab, then try the likely question at the end of the unit.

Unit 1 — Introduction and Features (2 Periods, 15%)

Topics: Fundamentals of object-oriented programming — procedure-oriented vs object-oriented programming (OOP); OOP concepts — classes, object, object reference, abstraction, encapsulation, inheritance, polymorphism; introduction of Eclipse (IDE) for developing programs in Java.

Procedure-oriented vs object-oriented programming

In **procedure-oriented programming** (POP), such as C, the program is a list of functions and the data is mostly global, so any function can change it. The focus is on **what is to be done** — the procedure — and large programs become hard to manage because data has little protection.

In **object-oriented programming** (OOP), the program is built from **objects** that combine data and the methods that work on that data. The focus is on the **data and the objects** themselves.

- POP follows a top-down approach; OOP follows a bottom-up approach.
- POP has no proper data hiding; OOP hides data through access control.
- OOP supports inheritance and polymorphism, so code can be reused and extended instead of rewritten.

Java follows the OOP route, which is why every Java program is written inside a class.

Classes, object and object reference

A **class** is a blueprint or template that describes the variables (data) and methods (behaviour) common to a group of similar items. A class itself occupies no memory for its data. An **object** is a real instance created from a class — memory is allotted when the object is created.

Example:

```
class Student { int roll; String name; }  
Student s1 = new Student();
```

Here Student is the class, s1 is the object, and **new** creates the object in memory. The variable s1 does not hold the object itself; it holds an **object reference** — the address-like link to the object. Through this reference we reach the object's variables and methods, for example s1.roll = 5; Two references can also point to the same object, so a change through one is visible through the other.

Abstraction and encapsulation

Abstraction means showing only the essential features and hiding the inner working details. When you call a method, you use what it does without worrying about how it does it inside. Abstract classes and interfaces are Java's tools for abstraction, covered in Unit 4.

Encapsulation means wrapping data and the methods that operate on it into one unit — the class — and protecting the data from direct outside access. The usual pattern is:

- Declare variables **private**, so no outside code can change them directly.
- Provide **public** methods to read or update them in a controlled way, after any checking that is needed.

Difference in one line: abstraction hides **complexity** (what is shown), encapsulation hides and protects the **data** itself (how it is guarded). Together they make programs safer and easier to change.

Inheritance and polymorphism

Inheritance is the mechanism by which one class acquires the variables and methods of another class. The existing class is the **superclass (parent)** and the new class is the

subclass (child), written with the keyword **extends**. The child reuses parent code and can add its own features — for example, a SavingsAccount class can inherit common balance and deposit behaviour from an Account class and add its own interest rule.

Polymorphism means **one name, many forms**. The same method name can behave differently in different situations: a draw() method draws a circle in one class and a square in another. Java shows polymorphism through method overloading (same class, different parameters) and method overriding (same method, redefined in the subclass). Both ideas are developed fully in Units 3 and 4, so learn their definitions and one example each here.

Introduction of Eclipse (IDE) for developing programs in Java

An **IDE (Integrated Development Environment)** is a single software package in which you write, compile, run and correct programs. **Eclipse** is a popular free IDE for Java. Instead of typing commands separately for editing, compiling and running, everything is done inside one window.

Main parts and advantages:

- **Editor** — colour highlighting, automatic indentation and completion suggestions while typing code.
- **Project and package explorer** — organises your .java files, classes and packages in one tree.
- **One-click build and run** — Eclipse compiles automatically and shows errors in a problems list; double-clicking an error jumps to its line.
- **Debugger** — run the program step by step and watch variable values to find logical mistakes.

Typical lab flow: create a Java project, add a class with a main method, write the code, then choose Run. The output appears in the Console view at the bottom.

Likely question: Differentiate between procedure-oriented programming and object-oriented programming, and explain class, object, abstraction, encapsulation, inheritance and polymorphism with examples.

Unit 2 — Language Constructs (3 Periods, 15%)

Topics: Variables; data types — integer, floating point, character, boolean; operators; iteration and jump statements; conditional expressions; input using Scanner class and output statements; loops; switch-case; arrays; introduction to methods.

Variables and data types

A **variable** is a named memory location used to store a value of a fixed type, declared as: `int marks = 85;` Java is strongly typed — the type is checked at compile time, so a value of the wrong type is rejected.

Primitive data types in the syllabus groups:

- **Integer types** — byte (1 byte), short (2 bytes), int (4 bytes) and long (8 bytes), for whole numbers.
- **Floating point types** — float (4 bytes) for decimal values written like 99.5f, and double (8 bytes) for more precise decimals, which is the default for decimal literals.
- **Character type** — char (2 bytes), one character in single quotes, for example 'A'; it supports Unicode.
- **Boolean type** — boolean, holding only true or false, used for conditions.

Choose the smallest type that safely holds your values — int and double cover most diploma-level programs.

Operators and conditional expressions

Operators perform operations on values. The main groups are: arithmetic (+, -, *, /, % for remainder), relational (>, <, >=, <=, ==, !=) giving a boolean result, logical (&& AND, || OR, ! NOT) joining conditions, assignment (= and short forms such as +=), and increment/decrement (++ and --) which add or subtract 1.

A **conditional expression** uses the ternary operator **? :** to choose one of two values in a single line:

```
String result = (marks >= 40) ? "Pass" : "Fail";
```

The part before ? is the condition; if it is true the first value is taken, otherwise the second. It is a compact replacement for a simple if-else that assigns a value, but for long or multi-step actions a normal if-else is clearer and safer to write in exams.

Input using Scanner class and output statements

Keyboard input in console programs is commonly taken with the **Scanner** class from the java.util package.

Steps and skeleton:

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
double d = sc.nextDouble();
String word = sc.next();
```

- **nextInt(), nextDouble(), nextFloat()** read numbers; **next()** reads one word; **nextLine()** reads a full line including spaces.
- Create one Scanner object for System.in and reuse it throughout the program.

Output statements: System.out.print() writes without moving to a new line, while System.out.println() writes and then moves to the next line. Values and text are joined with +, for example System.out.println("Sum = " + sum);

Remember to import Scanner before the class; without the import line the compiler reports that the symbol Scanner cannot be found, even when the rest of the program is fully correct.

Loops, iteration and jump statements; switch-case

Iteration statements (loops) repeat a block: **for** when the count is known — `for (int i = 1; i <= 5; i++)`, **while** when repetition depends on a condition checked first, and **do-while** when the body must run at least once because the condition is checked at the end.

Jump statements change the normal flow: **break** leaves the loop or switch immediately, **continue** skips the rest of the current round and starts the next round, and **return** leaves the current method, optionally sending a value back.

switch-case selects one block by matching an exact value:

```
switch (day) { case 1: System.out.println("Mon"); break; default:
System.out.println("Other"); }
```

Without break the execution falls into the next case, so write break at the end of every case unless sharing code is intended.

Arrays

An **array** stores many values of the same type under one name, reached by an index starting at **0**.

Declaration and use:

```
int[] marks = new int[5];
marks[0] = 70; marks[1] = 85;
int[] nums = {10, 20, 30}; /* direct values */
```

- **new** creates the array; its size is fixed once created and cannot grow or shrink.
- **marks.length** gives the number of elements — note it is a field, written without brackets.
- A loop is the normal way to fill, print or total an array: `for (int i = 0; i < marks.length; i++)`.
- Using an index outside 0 to length-1 throws an `ArrayIndexOutOfBoundsException` at run time.

Arrays are objects in Java, so an array variable holds a reference to the array created with `new`.

Introduction to methods

A **method** is a named block of code that performs one task and can be called whenever that task is needed, instead of repeating the same lines. A method definition has a return type, a name, parameters in brackets, and a body in braces:

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

- **Return type** — the type of value sent back; use **void** when nothing is returned.

- **Parameters** — inputs the method needs; the values passed in a call are called arguments, for example add(4, 5).
- **return** — sends the result to the caller and ends the method.
- **main** is the special method where execution begins: public static void main(String[] args).

Methods divide a big program into small, testable pieces and are the base for overloading and overriding in Unit 4.

Likely question: Explain the primitive data types of Java and write a program that reads numbers using Scanner, stores them in an array, and prints their sum and average.

Unit 3 — Classes and Objects (3 Periods, 30%)

Topics: Class fundamentals; constructors; declaring objects (object and object reference); creating and accessing variables and methods; static and non-static variables/methods; defining packages; creating and accessing a package; importing packages; understanding CLASSPATH; auto-boxing; String and StringBuffer. Inheritance: definition; protected/private/public data; constructor chaining; order of invocation; types of inheritance (single, multilevel, hierarchical, hybrid); access control (private vs public vs protected vs default).

Class fundamentals, constructors and declaring objects

A **class** groups related variables (instance variables) and methods under one name. Its **fundamentals**: the class keyword, a name starting with a capital letter by convention, variables declared inside, and methods that work on them. Members are reached through an object with the dot operator.

A **constructor** is a special method with the **same name as the class and no return type**. It runs automatically when an object is created with new, mainly to give starting values:

```
Student(int r) { roll = r; }
```

If you write no constructor, Java supplies a default constructor with no arguments.

Constructors can be overloaded.

Declaring objects: Student s1; only declares a reference. Student s1 = new Student(5); declares the reference, creates the object, and runs the constructor. After that, variables and methods are created/accessed as s1.roll and s1.show().

Static and non-static variables and methods

Non-static (instance) members belong to each object separately. Every object gets its own copy of an instance variable — two Student objects have different roll values — and instance methods are called through an object: s1.show();

Static members belong to the class itself and are shared by all objects — there is only one copy. They are declared with the keyword **static** and are called through the class name:

- `static int count;` — one counter shared by every object, useful for counting how many objects were made.
- `ClassName.methodName();` — the way a static method is called, with no object needed.
- A static method can directly use only static members; to use instance members it needs an object.
- **main is static** so the JVM can call it before any object exists.

Packages, importing packages and CLASSPATH

A **package** is a folder-like group of related classes that avoids name clashes and organises large programs. **Defining and creating a package:** write `package mypack;` as the **first line** of the file; the class file is then stored in a folder with the same name as the package.

Accessing and importing: another program uses the class by writing `import mypack.MyClass;` or `import mypack.*;` to import all classes of that package, or by using the fully qualified name. Java's own `Scanner` comes the same way: `import java.util.Scanner;`

CLASSPATH is the list of locations (folders and jar files) where the Java compiler and JVM search for classes and packages. If a package folder is not on the CLASSPATH, the compiler reports that the package does not exist, even though the file is present on the disk. In Eclipse the project build path does this job automatically.

Auto-boxing; String and StringBuffer

Auto-boxing is the automatic conversion of a primitive value into its wrapper-class object: `Integer obj = 10;` converts `int` to `Integer` automatically. The reverse, object to primitive, is **unboxing**: `int n = obj;` Wrapper classes such as `Integer`, `Double` and `Character` are needed when objects are required, for example in collections.

String holds text and is **immutable** — once created, its characters cannot be changed; every change makes a new `String` object. `String name = "Student";` Common operations include `length()`, `charAt()`, `equals()` for comparing contents, and concatenation with `+`.

StringBuffer holds text that **can be changed (mutable)** in the same object, using methods such as `append()`, `insert()` and `reverse()`. Prefer `StringBuffer` when text is built up by many changes in a loop, because creating a new `String` again and again wastes memory and time.

Inheritance — definition, types and constructor chaining

Inheritance is creating a new class from an existing class so that the new class reuses its variables and methods: `class B extends A { }`. `A` is the superclass, `B` the subclass. Data may be kept protected, private or public in the parent — protected members are inherited and directly usable in subclasses, while private members are not directly accessible and are reached through inherited public or protected methods.

Types:

- **Single** — one subclass from one superclass.
- **Multilevel** — a chain, C from B and B from A.
- **Hierarchical** — many subclasses from one superclass.
- **Hybrid** — a mixture of the above; through classes Java supports it only in limited form because multiple class inheritance is not allowed.

Constructor chaining and order of invocation: when a subclass object is made, the superclass constructor runs **first**, then the subclass constructor. **super()** calls the parent constructor and must be the first line; **this()** calls another constructor of the same class.

Access control — private vs public vs protected vs default

Access control decides where a member can be used. Java has four levels, and choosing the right one is a very common exam table:

- **private** — usable only inside its own class. Same class: Yes. Same package: No. Subclass elsewhere: No. Everywhere: No.
- **default** (no keyword written) — usable inside the same package only. Same class: Yes. Same package: Yes. Subclass in another package: No. Everywhere: No.
- **protected** — usable in the same package, and in a subclass in another package. Same class: Yes. Same package: Yes. Subclass elsewhere: Yes. Unrelated class elsewhere: No.
- **public** — usable from everywhere: Yes in all four cases.

Good practice: keep variables private, provide public methods for controlled access, and use protected for members that subclasses genuinely need to inherit.

Likely question: Explain inheritance and its types with examples. Describe constructor chaining and the order of constructor invocation, and compare private, public, protected and default access.

Unit 4 — Abstract Class, Interface and Polymorphism (3 Periods, 20%)

Topics: Defining an interface; key points of abstract class and interface; difference between an abstract class and an interface; implementation of multiple inheritance through interface; method and constructor overloading; method overriding; up-casting and down-casting.

Defining an interface; key points of abstract class and interface

An **interface** is a pure specification of method names that a class promises to implement. It is defined with the keyword interface and used with implements:

```
interface Printable { void print(); }  
class Report implements Printable { public void print() { System.out.println("Report"); } }
```

Key points — interface: its methods are abstract by default and carry no body (in the basic syllabus sense); its variables are constants; a class can implement many interfaces; an object of an interface cannot be created.

Key points — abstract class: declared with the keyword **abstract**; it may contain both abstract methods (no body) and normal methods with a body, as well as variables and constructors; a subclass must implement its abstract methods or itself be declared abstract; its object also cannot be created directly — objects are made of concrete subclasses.

Difference between an abstract class and an interface

Both abstract classes and interfaces are used to achieve abstraction and neither can be instantiated directly, but they differ in important ways that are asked as a direct comparison:

- **Methods:** an abstract class may have abstract and concrete (bodied) methods; an interface mainly declares abstract methods.
- **Variables:** an abstract class can have ordinary variables of any access level; interface variables are constants — final and static by nature.
- **Constructors:** an abstract class can have constructors, called through subclass construction; an interface has no constructors.
- **Relation keyword:** a class extends an abstract class, but implements an interface.
- **How many:** a class can extend only **one** abstract class, but can implement **many** interfaces.
- **Use when:** choose an abstract class for closely related classes sharing common code; choose an interface for a capability that unrelated classes must all provide.

Implementation of multiple inheritance through interface

Multiple inheritance means one class taking features from two or more parents. Java does **not** allow it with classes, because if two parents had a method with the same name and body style, the child would not know which one to inherit — the well-known ambiguity (diamond) problem.

Interfaces solve this safely, because an interface gives only method declarations, not bodies. A class supplies its own body once, so there is no confusion:

```
interface A { void show(); }  
interface B { void display(); }  
class C implements A, B {  
    public void show() { System.out.println("A"); }  
    public void display() { System.out.println("B"); } }
```

The class C must implement **every** method of both interfaces, otherwise it must be declared abstract. In this way Java gets the benefit of multiple inheritance — one class honouring

several specifications — without the ambiguity of inheriting two different bodies for the same method.

Method and constructor overloading

Overloading means using the same name for more than one method (or constructor) in the **same class**, with **different parameter lists** — different number, types or order of parameters. The return type alone cannot be the difference.

Method example:

```
int add(int a, int b) { return a + b; }
double add(double a, double b) { return a + b; }
int add(int a, int b, int c) { return a + b + c; }
```

The compiler decides which version to call by matching the arguments — this is **compile-time (static) polymorphism**. **Constructor overloading** follows the same rule: a class may have a no-argument constructor, a constructor with one parameter, and so on, so objects can be created in different ways, for example `Student()` and `Student(int roll, String name)`. Overloading makes an API easy to use because one sensible name covers similar jobs.

Method overriding

Overriding means a subclass provides its **own version** of a method that it inherits from its superclass. The method name, return type and parameter list must be **exactly the same**; only the body changes.

```
class Animal { void sound() { System.out.println("Sound"); } }
class Dog extends Animal { void sound() { System.out.println("Bark"); } }
```

- The decision about which version runs is made at **run time** from the actual object — this is runtime (dynamic) polymorphism.
- The overriding method cannot be given a more restricted access than the parent method.
- Constructors and private or final methods are not overridden.
- Writing **@Override** above the method is good practice; the compiler then checks that a matching parent method really exists.

Difference from overloading in one line: overloading differs in parameters within one class, overriding keeps the signature same across parent and child.

Up-casting and down-casting

Casting between class types works along the inheritance line. **Up-casting** means treating a subclass object as its superclass type:

```
Animal a = new Dog(); /* up-casting, automatic and safe */
```

Through `a`, only members known to `Animal` can be called, but an overridden method still runs the `Dog` version at run time — this is exactly how runtime polymorphism is used with a common reference type.

Down-casting means converting a superclass reference back to the subclass type:

```
Dog d = (Dog) a; /* down-casting, explicit */
```

It must be written manually with a cast, and it is safe only when the object really is of that subclass. If the object is actually of another subclass, the program throws a `ClassCastException` at run time. The `instanceof` operator checks safely before casting: if (a instanceof Dog) { Dog d = (Dog) a; } Use up-casting freely for general code; use down-casting only when subclass-specific members are genuinely needed.

Likely question: Differentiate between an abstract class and an interface, and show how multiple inheritance is implemented in Java through interfaces with a program.

Unit 5 — Exception Handling and Multithreading (3 Periods, 20%)

Topics: Definition of exception handling; implementation of keywords `try`, `catch`, `finally`, `throw` and `throws`; built-in exceptions; difference between multi-threading and multi-tasking; thread life cycle; creating threads; thread priorities.

Definition of exception handling; try, catch and finally

An **exception** is an abnormal event that occurs during execution and disturbs the normal flow of a program, for example dividing by zero or using an index outside an array. **Exception handling** is the mechanism of catching such an event and dealing with it in a planned way, so the program ends gracefully or continues instead of crashing suddenly with a system message.

Basic skeleton:

```
try { int x = 10 / 0; }  
catch (ArithmeticException e) { System.out.println("Cannot divide by zero"); }  
finally { System.out.println("Always runs"); }
```

- **try** — keeps the risky code; if an exception occurs, the rest of the try block is skipped.
- **catch** — handles one type of exception; a try can have several catch blocks for different types.
- **finally** — runs whether an exception occurred or not; used for closing files and releasing resources.

throw and throws; built-in exceptions

throw is used inside a method to **raise an exception deliberately** when a condition fails: if (age < 18) throw new `ArithmeticException`("Not eligible"); Execution stops there and the matching catch is searched for.

throws is written in the **method declaration** to declare that the method may pass an exception to its caller instead of handling it itself: `void read() throws IOException { }` The caller must then handle it or declare it further.

In one line: throw actually throws one exception object; throws only declares a possibility.

Built-in exceptions to name in exams:

- **ArithmeticException** — division by zero.
- **NullPointerException** — using a reference that points to no object.
- **ArrayIndexOutOfBoundsException** — index outside the array range.
- **NumberFormatException** — converting invalid text to a number.

Difference between multi-threading and multi-tasking

Multi-tasking means running several separate **programs (processes)** at the same time, such as a browser, a music player and an editor open together. The operating system switches the CPU between them. Each process has its own separate memory space, so processes are heavy and do not directly share data.

Multi-threading means running several **threads — parts of one program** — at the same time, such as one thread downloading a file while another keeps the screen responsive. Threads of the same program share the same memory, so they are light, fast to create, and can share data easily, though shared data must be handled carefully.

- Level: multi-tasking works between processes; multi-threading works within one process.
- Memory: separate for processes; shared for threads.
- Cost: creating a process is heavy; creating a thread is light.
- Example: in a Java program, multithreading is created with Thread or Runnable, studied next.

Thread life cycle

A thread passes through several states during its life. Learn them in order:

- **New** — the Thread object is created but start() has not yet been called.
- **Runnable** — after start(), the thread is ready and waiting for the CPU scheduler to give it time; its run() method will execute when selected.
- **Running** — the scheduler has selected it and run() is actually executing.
- **Blocked / Waiting** — the thread is temporarily inactive, for example sleeping with sleep(), waiting for input, or waiting for another thread; it returns to Runnable when the reason ends.
- **Terminated (Dead)** — run() has finished or an unhandled exception stopped it. A dead thread cannot be started again.

Diagram in words: New → (start) Runnable ↔ Running → (sleep/wait) Blocked → back to Runnable → Terminated when run() ends. Calling run() directly does not create a new thread — always call start().

Creating threads and thread priorities

There are two standard ways of **creating a thread**:

- **Extending Thread class** — class MyThread extends Thread, override run() with the task, then MyThread t = new MyThread(); t.start();
- **Implementing Runnable interface** — class MyTask implements Runnable, write run(), then Thread t = new Thread(new MyTask()); t.start(); This way is preferred when the class already extends another class.

In both ways the task goes inside **run()**, and **start()** is called to begin the new thread — start() internally calls run() on the new thread.

Thread priorities tell the scheduler which thread should be preferred when several are runnable. Priorities are numbers from **1 to 10**: Thread.MIN_PRIORITY is 1, NORM_PRIORITY is 5 (the default) and MAX_PRIORITY is 10. They are set with setPriority() and read with getPriority(). A higher priority is a request to the scheduler, not a guarantee of order.

Likely question: Explain exception handling and the use of try, catch, finally, throw and throws. Also describe the thread life cycle with a diagram.

Quick Revision — One Line Each

- POP is function-centred with shared global data; OOP is object-centred with data protected inside classes.
- A class is the blueprint; an object is its instance; the reference variable holds the link to the object.
- Encapsulation wraps and protects data; abstraction hides complexity and shows only essentials.
- Array index starts at 0, size is fixed, and marks.length gives the element count.
- A constructor has the class name, no return type, and runs automatically with new.
- static members belong to the class and are shared; instance members belong to each object.
- package is written as the first line; import brings a package in; CLASSPATH tells Java where to search.
- String is immutable, StringBuffer is mutable — use StringBuffer for repeated changes.
- Inheritance types: single, multilevel, hierarchical, hybrid; parent constructor runs before the child constructor.
- private: own class only; default: own package; protected: package plus subclasses; public: everywhere.
- A class extends one class but can implement many interfaces — that is how Java gets multiple inheritance.
- Overloading differs in parameters in one class; overriding keeps the signature same in parent and child.
- try keeps risky code, catch handles it, finally always runs; throw raises, throws declares.

- Thread states: New, Runnable, Running, Blocked/Waiting, Terminated; priorities run from 1 to 10, default 5.

KAMALDNP • kamaldnp.com • support@kamaldnp.com • +91 8318013442 • Varanasi, UP

Prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus topics. Short explanations are the author's own plain-English summaries, not syllabus text. Please confirm with your own polytechnic for any local changes.

