Week 2 – Tutorial 1

Guidelines:

1. Use Lab Practical 1 and 2 as your guideline to create the report that can be used for future reference.
   Save your Tutorial 1 in words and Save as PDF file format.
2. Submission:
   a) GitHub:
      a. File types: JAVA code, Report
      b. Naming:
         i. Package: Chapter1
         ii. Class: read your slides / questions for the names
         iii. Report: Student Name [Student ID] _Tutorial 1.pdf
         iv. Presentation Slide: Student Name [Student ID] _Tutorial 1.pptx (if necessary)
   b) Teams:
      a. File types: Report
      b. File name: Student Name [Student ID] _Tutorial 1.pdf
3. For answers, please try to use lecture notes, google, w3schools, Tutorial Point as reference. This is to train research skills, problem-solving skills, conceptual skills and technical skills.
   AI tools can be used for brainstorming and explanation **BUT NOT AS ANSWERS PROVIDER**.
4. This tutorial consists of 2 parts which both need to be answered:
   a. Part A
   b. Part B

PART A:

1. Provide the answers for Class Activity from Chapter 1's page 9.
   a. Make sure proper references are used when providing the answer. Attach the references that have been used
   b. AI tools can be used to boost understanding but not direct copy-paste the answers as the answer somehow might not be relevant.

Difference between:

Table 1: low-level language and high-level language and middle-level language

Language	Low-level	Mid-level	High-level
Abstraction level	Very low (close to hardware)	Moderate (mix of hardware access and abstraction)	High (focus on logic, far from hardware)
portability	Very poor (specific to hardware/CPU)	Moderate (portable but may require adaptation)	High (runs on multiple platforms with little change)
Character	High performance and hardware control, but hard to write, debug, and maintain.	Balance control and abstraction for system programming like OS development.	Easier to use and maintain, but with less hardware control and possibly lower performance.
Examples	Machine language (binary code)	C	C++, Java, Python, JavaScript

- **Low-level languages** prioritize direct hardware control and efficiency but sacrifice ease of use and portability.
- **High-level languages** prioritize ease of use, readability, and portability but may offer less direct control over hardware.
- **Middle-level languages** attempt to strike a balance between these two extremes, providing a degree of hardware control while retaining some high-level features.

Table 2: machine language and assembly language

Languages	Machine language	Assembly language
Representation	Machine language is the lowest-level programming, made of binary (0s and 1s) instructions directly executed by the CPU.	Assembly language is a human-readable form of machine code, using mnemonics and symbolic addresses for instructions and memory.
Readability	Machine language is hard for humans to read, write, and debug because it is purely numeric.	Assembly language is low-level but more readable than machine code, using mnemonics (e.g., ADD A, B) instead of binary.

- **Machine Language:** Pure binary, directly executed by CPU, very hard for humans.
- **Assembly Language**: Symbolic form of machine code uses mnemonics, more readable.

Table 3: Imperative and Declarative paradigm

Type of paradigm	**Imperative Paradigm**	**Declarative Paradigm**
Core Idea	Tell the computer how to do step by step	Tell the computer what result you want
Subtype	Procedural, Object-Oriented (OOP)	Functional, Logic
Example Language	C, Python, Java	HTML, SQL

- **Imperative**: Focus on how to do by tell the computer step-by-step
- **Declarative:** Focus on what result you want

Table 4: Subtypes of imperative paradigm and declarative paradigm

Subtype	**Procedural**	**Object-Oriented Programming (OOP)**	**Functional Programming**	**Logic Programming**
definition	Organizes code into functions for tasks.	Structures code around objects with data and methods.	Focuses on pure functions, immutability, no side effects.	Based on rules and facts; system infers solutions.
Core idea	Focus on functions and step-by-step execution.	Focus on objects and their interactions.	Pure functions, immutability, no side effects	Rules and facts, system infers solutions

- **Procedural**: Function-centered, step-by-step execution.
- **OOP**: Object-centered, combines data and behavior.
- **Functional:** Function-centered, pure & immutable, no side effects.
- **Logic**: Rule-centered, system deduces solutions.

PART B:

A: MCQ (Choose the best answer)

No.	Questions	Answer
1	Which of the following is **NOT** a pillar of OOP? a) Abstraction b) Encapsulation c) Compilation d) Inheritance	C
2	In Java, a class is: a) An object b) A blueprint for creating objects c) A function d) A library	B
3	Which keyword is used to create an object in Java? a) class b) new c) extends d) public	A
4	The process of hiding data and providing controlled access is: a) Polymorphism b) Encapsulation c) Inheritance d) Abstraction	B
5	What is the default return type of a constructor in Java? a) void b) int c) No return type d) double	C
6	Which method is used to start program execution in Java? a) run() b) execute() c) main() d) start()	C
7	Which of these supports multiple inheritance in Java? a) Classes only b) Interfaces c) Abstract classes only d) None	B
8	Overloading in Java happens at: a) Compile time b) Runtime c) Link time d) Never	A

No.	Questions	Answer
9	Which OOP principle allows reusing code from another class? a) Inheritance b) Encapsulation c) Abstraction d) Polymorphism	A
10	The keyword *extends* is used for: a) Method overloading b) Inheritance c) Encapsulation d) Creating interfaces	B
11	Which pillar focuses on “what to do” and hides “how to do”? a) Encapsulation b) Abstraction c) Inheritance d) Polymorphism	B
12	An object in Java has: a) Identity, State, Behavior b) Class, Method, Variable c) Name, Type, Constructor d) None of the above	A
13	Which of the following can be overloaded? a) Constructors b) Methods c) Both a & b d) None	C
14	The keyword *super* is used to: a) Call parent class constructor b) Call child class method c) Creating an object d) None	A
15	Polymorphism means: a) One method, many forms b) One class, one object c) One attribute, one value d) None of the above	A

B: True/False

No	Questions	Answer
1	In Java, an abstract class can be instantiated	False
2	Encapsulation can be achieved using private variables and getters/setters	True
3	A constructor must have the same name as the class	True
4	Inheritance supports code reusability	True
5	The main method can be overloaded	True
6	Interfaces can have method implementations in Java	True
7	Method overriding happens at compile time	False
8	All classes in Java must have a constructor	True
9	Abstraction and encapsulation mean the same thing	False
10	Polymorphism is only possible through inheritance	False

C: Mix & Match

:

Column A	
1	Encapsulation
2	Abstraction
3	Inheritance
4	Polymorphism
5	Constructor
6	Overloading
7	Overriding
8	Class
9	Object
10	new keyword

Column B	
a	One method, many forms
b	Code reusability
c	Data hiding
d	Show essential details
e	Automatically called
f	Same method name, different parameters
g	Child redefines parent method
h	Blueprint for objects
i	Instance of a class
j	Creating an object

D: Structured Questions

1. Define OOP. List the four pillars of OOP.

   Object-oriented programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic, encapsulating data and operations within self-contained units.

Table 5 Four pillars of OOP

Pillars of OOP	Abstraction	Encapsulation	Inheritance	Polymorphism
Definition	Abstraction is the process of hiding complex internal workings, allowing users to see only the parts they need to interact with.	Encapsulation involves bundling data and the methods for processing that data together within a class, presenting it externally as a unified whole.	Inheritance enables a new class to automatically possess the data and functionality of another class, while also allowing modifications or enhancements to be made on that foundation.	Polymorphism means "the same command can produce different responses from different objects."
Purpose	To manage complexity by simplifying interactions with objects and making code easier to use and understand.	To restrict direct access to an object's internal state (data hiding), protecting it from accidental modification and promoting data integrity.	To promote code reuse and create a hierarchical relationship between classes, establishing an "is-a" relationship between them.	Providing flexibility so that the same method can have different implementations and be used interchangeably.

2. Explain the difference between a class and an object with examples.

Table 6 Difference between a class and an object

Aspect	Class	Object
Definition	A blueprint that defines attributes (data) and methods (behavior).	An instance of a class, created based on the class definition.
Nature	Logical/abstract entity.	Physical/real entity in memory.
Purpose	Defines what properties and behaviors objects will have.	Represents a specific entity with actual values.
Creation	Defined once in code.	Created many times from the class.

Example:

Class: Car

- Attributes: colour, make, model, year
- Methods: start_engine(), accelerate(), brake()

Objects:

- myCar: An object of the Car class with colour = "red", Brand = "Toyota", model = "Camry", year = 2020.
- yourCar: Another object of the Car class with color = "blue", make = "Honda", model = "Civic", year = 2022.

3. Differentiate between abstraction and encapsulation.

Table 7 Differentiate between abstraction and encapsulation

Aspect	**Abstraction**	**Encapsulation**
Definition	Hiding complex implementation details and showing only essential features.	Bundling data (attributes) and methods (behavior) into a single unit (class).
Focus	Focuses on what a system does.	Focuses on how data and methods are combined and accessed.
Purpose	To reduce complexity and increase simplicity for the user.	To achieve data hiding and restrict direct access to data.
Implementation	Achieved using abstract classes and interfaces.	Achieved using classes, access modifiers (private, public, protected).
Example	A car driver uses steering and pedals without knowing the internal engine design.	The car’s engine details are hidden inside the car body, accessed only via defined methods.

4. Write a simple Building class with attributes and methods.

```java
package self_Project_ForHomeWork;
public class Motor {
    // Attributes (fields) of the Motor class
    String brand;   // Brand of the motor
    String plate;   // Plate number
    String owner;   // Owner's name

    // Constructor: initializes a new Motor object with brand, plate, and owner
    public Motor(String brand, String plate, String owner) {
        this.brand = brand;
        this.plate = plate;
        this.owner = owner;
    }
    // Method to display motor details
    public void detailInfo() {
        System.out.println("Motor brand: " + brand +
                           " plate is: " + plate +
                           " Owner is: " + owner);
    }
    // Main method: entry point of the program
    public static void main(String[] args) {
        // Create two Motor objects with different attributes
        Motor motor1 = new Motor("Suzuki", "PLL2224", "Te Chong Yi");
        Motor motor2 = new Motor("Yamaha", "ABC123", "Alex");

        // Call the detailInfo() method to display their details
        motor1.detailInfo();
        motor2.detailInfo();
    }
}
```

Figure 1 simple code for attributes and methods.

```
<terminated> Motor [Java Application] C:\Program Files\Java\jdk-24\bin\javaw.exe (20 Se
Motor brand: Suzuki plate is: PLL2224 Owner is: Te Chong Yi
Motor brand: Yamaha plate is: ABC123 Owner is: Alex
```

Figure 2 Output

5. What are constructors? Explain with an example.

   A constructor is a special method in object-oriented programming that is automatically invoked when creating an object of a class. It is used to initialize the object's state and set initial values for its properties.

   Key Characteristics:

- **Automatic Invocation:** A constructor is automatically executed when you create an instance (object) of a class.
- **Name and Return Type:** A constructor shares the same name as its class and does not have a return type, not even void.
- **Initialization:** Their primary purpose is to initialize the attributes (variables) of a new object, giving them their initial values.
- **Parameterization:** Constructors can take parameters, which allows you to pass different values during object creation to customize the object's initial state.
- **Validation:** They can also contain logic to validate input data, ensuring that the object is created with valid data.

Example: Figure 1 and Figure 2

6. Differentiate the mutators and accessors.

Table 8 Differentiate the mutators and accessors

Aspect	**Accessors (Getters)**	**Mutators (Setters)**
Purpose	Used to **access** (read) the value of a private attribute.	Used to **modify** (change) the value of a private attribute.
Return type	Usually returns a value (same type as attribute).	Always void (does not return a value).
Naming	Typically starts with get (e.g., getBrand()).	Typically starts with set (e.g., setBrand()).
Effect	Does not change the object's state.	Changes/updates the object's state.
Example	public String getBrand() { return brand; }	public void setBrand(String brand) { this.brand = brand; }

7. Show method overloading with a Java example that is not given in the lecture slides.

```
Motor.java    *ex_OverLoading.java
package self_Project_ForHomeWork;
public class ex_OverLoading {
    String name;
    int age;
    // Constructor with parameters
    public ex_OverLoading(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Overloaded method 1: no parameters
    void sayHello() {
        System.out.println("Hello guys!");
    }

    // Overloaded method 2: one String parameter
    void sayHello(String name) {
        System.out.println("Hello guys, my name is " + name + "!");
    }
    // Overloaded method 3: one int parameter
    void sayHello(int age) {
        System.out.println("Hello guys, my name is " + name + ", I'm " + age + " years old!");
    }
    public static void main(String[] args) {
        // Create an object of ex_OverLoading using the constructor
        ex_OverLoading hello = new ex_OverLoading("Chong Yi", 20);
        System.out.println("--- Method Overloading Demo ---");
        // Call the overloaded methods
        hello.sayHello();              // Calls method with no parameters
        hello.sayHello("Chong Yi");    // Calls method with String parameter
        hello.sayHello(20);            // Calls method with int parameter
    }
```

Figure 3 code for showing overloading

```
Problems  @ Javadoc  Declaration  Console ×  Instal
<terminated> ex_OverLoading [Java Application] C:\Program Files\Java\
--- Method Overloading Demo ---
Hello guys!
Hello guys, my name is Chong Yi!
Hello guys, my name is Chong Yi, I'm 20 years old!
```

Figure 4 output

8. Show method overriding with a Java example that is not given in the lecture slides.

Motor.java | ex_OverLoading.java | *ex_Overriding.java

```java
package self_Project_ForHomeWork;
// Parent class
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound");
    }
}
// Child class (overrides the parent method)
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks: Woof Woof!");
    }
}
// Another Child class (also overrides the parent method)
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("The cat meows: Meow Meow!");
    }
}
```

```java
public class ex_Overriding {
    public static void main(String[] args) {
        // Create objects using parent reference
        Animal a1 = new Animal(); // Parent class object
        Animal a2 = new Dog();    // Dog object, but referenced as Animal
        Animal a3 = new Cat();    // Cat object, but referenced as Animal
        System.out.println("--- Method Overriding Demo ---");
        // Calls methods (runtime polymorphism decides which one to run)
        a1.makeSound();   // Calls Animal's method
        a2.makeSound();   // Calls Dog's overridden method
        a3.makeSound();   // Calls Cat's overridden method
    }
}
```

Figure 5 code for showing overriding

Problems | @ Javadoc | Declaration | Console

```
<terminated> ex_Overriding [Java Application] C:\Program
--- Method Overriding Demo ---
The animal makes a sound
The dog barks: Woof Woof!
The cat meows: Meow Meow!
```

Figure 6 output

9. Write a program that demonstrates inheritance using Animal, Reptiles, Mammals and Insects.

```java
package self_Project_ForHomeWork;

// Parent class
class AnimalBase {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Child class: Reptile
class Reptile extends AnimalBase {
    void eat() {
        System.out.println("Reptile is eating food");
    }
}

// Child class: Mammal
class Mammal extends AnimalBase {
    void sleep() {
        System.out.println("Mammal is sleeping");
    }
}

// Child class: Insect
class Insect extends AnimalBase {
    void fly() {
        System.out.println("Insect is flying");
    }
}

```

```java
public class ExInheritance {
    public static void main(String[] args) {
        // Create objects of parent and child classes
        AnimalBase a1 = new AnimalBase();
        Reptile a2 = new Reptile();
        Mammal a3 = new Mammal();
        Insect a4 = new Insect();

        System.out.println("--- Inheritance Demo ---");

        // Parent class method
        a1.makeSound();

        // Reptile inherits makeSound() from AnimalBase
        a2.makeSound();
        a2.eat();  // Reptile's own method

        // Mammal inherits makeSound() from AnimalBase
        a3.makeSound();
        a3.sleep();  // Mammal's own method

        // Insect inherits makeSound() from AnimalBase
        a4.makeSound();
        a4.fly();  // Insect's own method
    }
}
```

Figure 7 code for showing inheritance

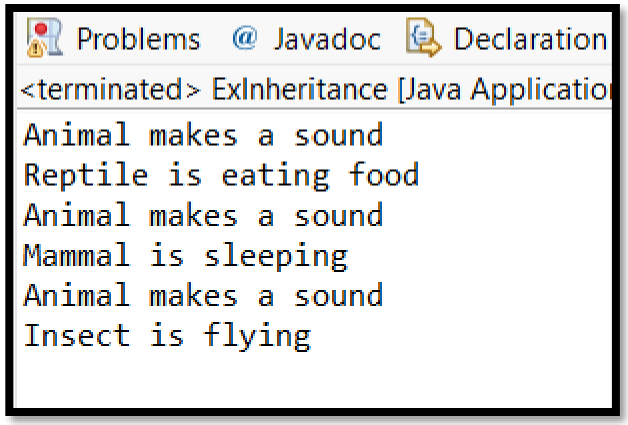

Figure 8 Output

10. Explain polymorphism with an overriding method example.

    Polymorphism is one of the four pillars of object-oriented programming. It allows objects of different classes to be treated as instances of a common parent class and enables the same method to perform different behaviors depending on the object.

    Example: Figure 5 and Figure 6

11. State 3 benefits and disadvantages of OOP beside from what have been learned in the lecture.

Table 9 advantages and disadvantages of OOP

Advantages	**Disadvantages**
Reusability Modularity Scalability	Steeper learning curve Increased complexity Testing and debugging challenges

12. Write and compare the characteristics of objects.

Table 10 compare characteristics of objects

Characteristic	Description	Example (Car object)
Identity	Uniqueness of the object (its reference in memory).	car1 and car2 are two different objects, even if both are Toyota.
State	The data/attributes that define the object.	color = red, speed = 120.
Behavior	The actions the object can perform.	start(), accelerate(), brake().

E: Research Questions

1. **Research the real-world use of abstraction in software development. Provide examples.**

   Abstraction is used in software development to manage complexity by hiding intricate internal details and presenting a simplified, user-friendly interface

   Real-world example:

   - **Smartphone Apps:** You tap an icon to make a call without knowing the complex network and hardware behind it.
   - **TV Remote:** You press buttons to change channels or volume without understanding how the signals work internally.
   - **Car Controls:** You use the steering wheel, gas, and brake without needing to know the engine or brake system details.

2. How is encapsulation applied in mobile app security?

Encapsulation refers to packaging data and its associated operations into a single unit while restricting direct external access to internal details. In the field of mobile application security, encapsulation ensures data integrity by concealing sensitive information and enforcing controlled access.

Encapsulation applied in mobile app security:

- **Data Hiding**: The internal data of an object is hidden from the outside world, preventing direct access.
- **Data Integrity:** Only validated or safe values can be assigned to an object's attributes via setter methods.
- **Reusability:** Encapsulated code is more flexible and reusable for future modifications or requirements.
- **Security:** Sensitive data is protected as it cannot be accessed directly.

```java
// Java program demonstrating Encapsulation
class Programmer {

    private String name;

    // Getter and Setter for name

    // Getter method used to get the data
    public String getName() { return name; }

    // Setter method is used to set or modify the data
    public void setName(String name) { this.name = name; }
}

public class Geeks {

    public static void main(String[] args) {
        Programmer p = new Programmer();
        p.setName("Geek");
        System.out.println("Name=> " + p.getName());
    }
}
```

```
Name=> Geek
```

Figure 9 example of encapsulation applied

Explanation: In the above example, we use the encapsulation and use getter (**getName)** and setter (**setName)** method which are used to show and modify the private data. This encapsulation mechanism protects the internal state of the Programmer object and allows for better control and flexibility in how the name attribute is accessed and modified.

3. Discuss a case study were inheritance improved software reusability.

Inheritance is a feature of object-oriented programming that allows subclasses to reuse the fields and methods of their parent classes. This approach prevents code duplication, enhances maintainability, and enables developers to extend existing functionality rather than rewrite code.

```java
package self_Project_ForHomeWork;

class User{
    String username;

    void login() {
        System.out.println(username + " logged in.");
    }

    void logout() {
        System.out.println(username + " logged out.");
    }
}

class Customer extends User {
    void placeOrder() {
        System.out.println(username + " placed an order.");
    }
}

// Derived class: Admin
class Admin extends User {
    void manageProducts() {
        System.out.println(username + " is managing products.");
    }
}
```

```java
//Derived class: Vendor
class Vendor extends User {
 void uploadProduct() {
     System.out.println(username + " uploaded a new product.");
 }
}
public class CaseStudy {
    public static void main(String[] args) {
            Customer c = new Customer();
            c.username = "Alice";
            c.login();
            c.placeOrder();
            c.logout();

            Admin a = new Admin();
            a.username = "Bob";
            a.login();
            a.manageProducts();
            a.logout();
    }
}
```

Figure 10 case study code

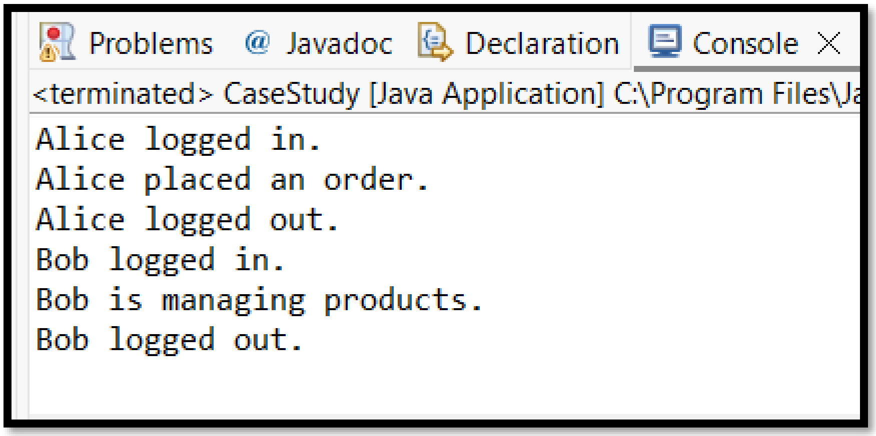

Figure 11 Output

4. Compare Java's approach to polymorphism with C++ and Python using Comparison Table.

Table 11 compare Java, C++ and Python

Feature / Aspect	**Java**	**C++**	**Python**
Type Checking	Strong, static type checking	Strong, static type checking	Dynamic, duck typing
Method Overloading	Supported (compile-time polymorphism)	Supported (function overloading)	Not supported directly
Method Overriding	Supported (runtime polymorphism)	Supported using virtual keyword	Supported (runtime polymorphism by default)
Operator Overloading	Not supported	Supported (custom operator overloading)	Supported (__add__, __str__, etc.)
Runtime Polymorphism	Achieved via method overriding and interfaces	Achieved via virtual functions	Achieved via dynamic typing and method overriding
Compile-time Polymorphism	Achieved via method overloading	Achieved via function overloading and templates	Not applicable
Inheritance Requirement	Must inherit from superclass or implement interface	Must inherit from base class	Inheritance optional (duck typing works)

5. **Explain the importance of OOP in large-scale software development projects.**

Object-oriented programming (OOP) is important because it simplifies complex software by translating real-world concepts into modular, reusable "objects," thereby enabling the construction of more flexible, maintainable, scalable, and secure applications.

Key Points:

Modularity

- OOP divides software into **classes and objects**, each representing a real-world entity or component.
- This makes the code easier to understand, test, and debug.

Reusability

- Through **inheritance** and **composition**, existing classes and objects can be reused in new parts of the system without rewriting code.
- Reduces development time and prevents duplication.

Maintainability

- Encapsulation ensures that internal implementation details are hidden.
- Changes in one class do not affect other parts of the system if interfaces remain consistent.

Scalability

- OOP allows developers to add new features or extend functionality by creating new classes or modifying existing ones without rewriting the entire codebase.

Polymorphism & Flexibility

- Methods can behave differently depending on the object, making the system adaptable to change and easier to extend.

Example in Large-Scale Systems

- In an **e-commerce platform**, classes like User, Product, Order, and Payment can be created and reused across different modules (web, mobile, admin).
- Updating the Payment class (e.g., adding a new payment method) does not require changing the rest of the system, thanks to encapsulation and modular design.