Data Hiding

- The process of **restricting direct access** to internal data from outside the class.
- Enhances **security** by shielding sensitive information.
- Achieved using the `private` access modifier.

```
class DataHiding {
    private double balance; // cannot be accessed outside the class
}
```

- Access to `private` members is provided via **getter** and **setter** methods:
    - `getter()` → retrieves the value.
    - `setter()` → modifies the value.

Data Abstraction

- Involves hiding the internal **implementation details** while exposing only essential **functionality**.

```
interface Switch {
    void turnOn();
    void turnOff();
}

class Light implements Switch {
    public void turnOn() {
        System.out.println("on");
    }
    public void turnOff() {
        System.out.println("off");
    }

    public static void main(String[] args) {
        Light light = new Light();
        light.turnOn();
        light.turnOff();
```

```
    }
}
```

- Achieved using **interfaces** and **abstract classes**.
- Key advantages:
    - Improved **security**
    - Simplified **usage**

Encapsulation

- The concept of **bundling data** and the methods that operate on it into a single unit.
- All Java classes are examples of encapsulation.
- It combines **Data Hiding** and **Abstraction**.

```java
// Encapsulated Class
public class Student {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public static void main(String[] args) {
        Student student = new Student();
        student.setName("Sambit");
        System.out.println(student.getName());
    }
}
```

Polymorphism

- The ability to define methods with the **same name** but **different behavior**.

Types:

- **Compile-time Polymorphism** → Method **Overloading**
    - Includes **Method Hiding**
- **Run-time Polymorphism** → Method **Overriding**

Refer to `Polymorphism.md` for detailed coverage.

Inheritance in Java

Inheritance is the mechanism by which one class **acquires** the properties (i.e., variables and methods) of another class.

- **Properties** = Variables + Methods
- The class whose properties are inherited is called the **Parent / Super / Base** class.
- The class that inherits the properties is called the **Child / Sub / Derived** class.
- The keyword `extends` is used by the **child** class to specify its **parent** class.
- A **parent** class reference can hold an object of its **child** class, but the reverse is not allowed.

```
1  ParentClass parent = new ChildClass(); // Valid
2  ChildClass child = new ParentClass();  // Invalid
```

Factory Method

- A **static method** that returns an **instance of the class** when called via the class name.

```java
class FactoryExample {
    public static FactoryExample getInstance() {
        return new FactoryExample();
    }
}

// Usage:
FactoryExample obj = FactoryExample.getInstance();
```

Singleton Class

- A class that allows the creation of **only one instance**.
- Common in utility classes like `Runtime`, `ActionServlet`, `ServiceLocator`, `BusinessDelegate`.
- Enhances **memory utilization** and improves **performance**.

```java
class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

class TestSingleton {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2); // true
    }
}
```
