**Week 11**

Q1. Write a report on the elements of object-oriented design.

Object-oriented design (OOD) is a software design approach that structures a program around objects rather than actions. Each object represents a component of the software system, encapsulating both data and behavior. OOD allows developers to model real-world entities, making the software system intuitive, reusable, and scalable. This report covers the essential elements of object-oriented design, including classes, objects, inheritance, encapsulation, abstraction, and polymorphism.

**1. Classes and Objects**

* **Classes**: A class is a blueprint that defines the attributes (data) and methods (functions) shared by all objects of that type. It represents a template for creating objects.
* **Objects**: An object is an instance of a class and is used to represent an entity with specific attributes and behaviors. Objects are the building blocks of an object-oriented program, enabling interaction and data manipulation.

**Example**: A Car class might have attributes like color, make, and model and behaviors such as startEngine() or stopEngine(). An instance of the Car class, like a Toyota or a Ford, is an object that embodies these properties and behaviors.

**2. Encapsulation**

* **Definition**: Encapsulation is the bundling of an object’s data (attributes) and behavior (methods) within a single unit, restricting access to certain elements to protect the integrity of the data.
* **Access Control**: Encapsulation is often implemented using access modifiers (private, protected, public) to control visibility, allowing only selected parts of a program to access specific data.
* **Benefits**: This helps safeguard data integrity and enhances modularity, allowing internal implementation to change without affecting external code.

**Example**: In a BankAccount class, the balance attribute can be private to prevent unauthorized modification. Access is managed through public methods like deposit() and withdraw().

**3. Abstraction**

* **Definition**: Abstraction is the process of simplifying complex systems by modeling classes that represent real-world entities without including unnecessary details. It focuses on the essential features relevant to the system.
* **Purpose**: By hiding unnecessary details and exposing only relevant attributes and methods, abstraction reduces complexity and increases efficiency in programming.
* **Implementation**: Abstract classes and interfaces in OOP help provide a high level of abstraction, defining a contract that subclasses must adhere to.

**Example**: A Shape interface might define methods like draw() and getArea(). Specific shapes like Circle or Rectangle implement this interface, each providing its details for the methods without exposing internal computations.

**4. Inheritance**

* **Definition**: Inheritance allows a new class (subclass) to inherit attributes and behaviors from an existing class (superclass), promoting code reuse and hierarchical relationships.
* **Types**: Inheritance can be single (one subclass inherits from one superclass) or multiple (a subclass inherits from multiple superclasses, depending on the language support).
* **Purpose**: It helps eliminate redundancy, as shared functionality can be defined in a base class and extended by subclasses with specific implementations.

**Example**: A Vehicle class with attributes like speed and fuel could be the base class. A Car class that inherits from Vehicle would have access to these attributes and could add specific behaviors like openTrunk().

**5. Polymorphism**

* **Definition**: Polymorphism allows the same function or method to operate in different ways depending on the context. It can be achieved through method overriding and method overloading.
* **Types**:
  + **Compile-time Polymorphism**: Achieved through method overloading, where methods have the same name but different parameters.
  + **Runtime Polymorphism**: Achieved through method overriding, where subclasses provide specific implementations of methods defined in a superclass.
* **Benefits**: Polymorphism enables flexibility and extensibility, allowing code to interact with objects of different classes in a unified way.

**Example**: A printDetails() method in a Vehicle class could be overridden by Car and Bike subclasses to provide specific information about each type of vehicle.

**6. Composition**

* **Definition**: Composition is the design principle that models a relationship where a class is composed of one or more objects from other classes, allowing complex objects to be built from simpler ones.
* **Purpose**: It enables code reuse without requiring inheritance and provides flexibility by allowing objects to hold references to other objects.
* **Difference from Inheritance**: While inheritance creates an “is-a” relationship, composition creates a “has-a” relationship.

**Example**: A Library class could have a List<Book> as an attribute, illustrating that a library has many books.

**7. Interfaces and Abstract Classes**

* **Interfaces**: Define a contract that implementing classes must adhere to, specifying what methods must be implemented without defining how they are implemented.
* **Abstract Classes**: Serve as partially implemented templates that other classes can extend. Abstract classes can contain both concrete methods and abstract methods.
* **Purpose**: Interfaces and abstract classes support polymorphism and abstraction by enforcing a consistent structure for classes that share similar behaviors.

**Example**: An Animal abstract class could have an abstract method makeSound(). Concrete classes like Dog and Cat must implement this method, allowing each to define their sound.

**8. Relationships**

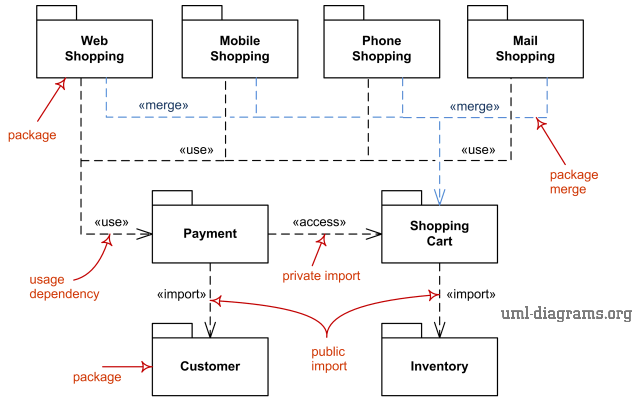
* **Association**: Represents a general relationship between two classes, where each class can function independently but work together.
* **Aggregation**: A form of association with a whole-part relationship where the parts can exist independently of the whole.
* **Composition**: A stronger form of aggregation, where the part cannot exist independently of the whole. When the whole is destroyed, so are the parts.

**Example**: In a university management system: - **Association**: A Student and a Course have an association, as students enroll in courses. - **Aggregation**: A Classroom has multiple Students, but students can exist without the classroom. - **Composition**: A House and its Rooms illustrate composition, as rooms cannot exist independently of the house.

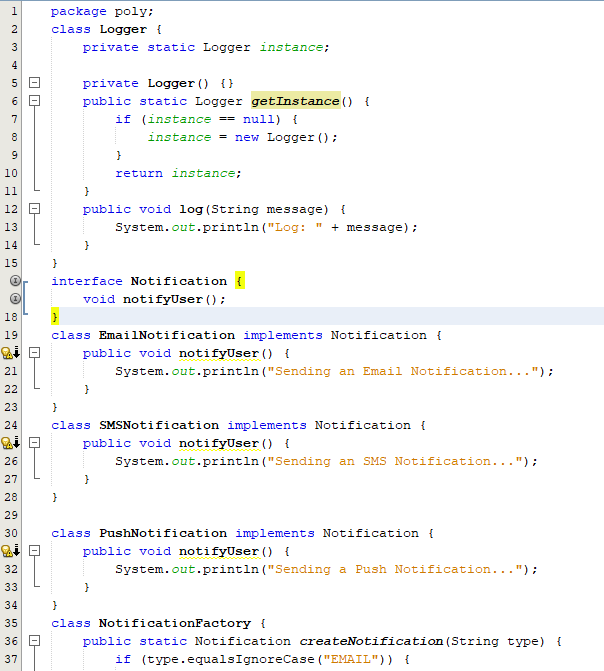
**Conclusion**

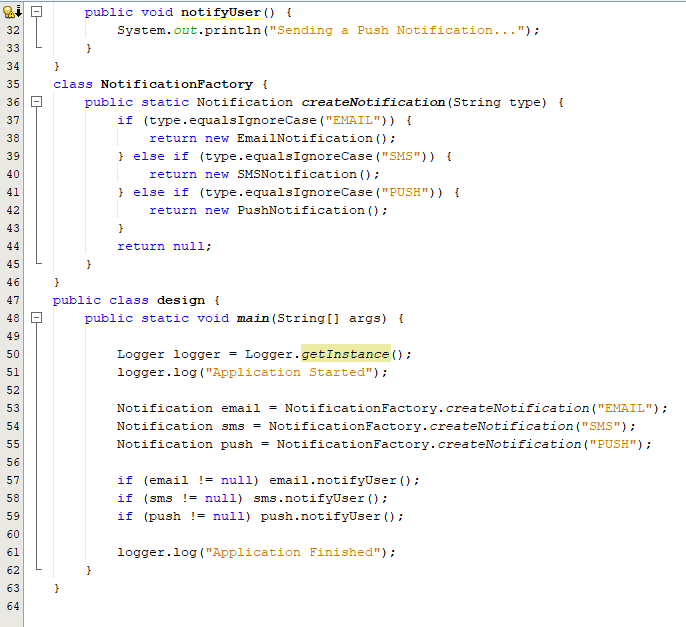
Object-oriented design (OOD) elements enable developers to create systems that are organized, flexible, and aligned with real-world structures. By utilizing OOD principles, such as encapsulation, inheritance, and polymorphism, developers can build systems that are modular and maintainable. Additionally, relationships like composition and association enhance the modularity and allow for high reusability. These elements form the foundation for creating scalable and robust software, making object-oriented design a cornerstone of modern software development.

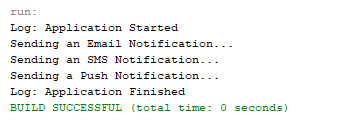
**Q.2).** Create a UML package diagram for a simple system.



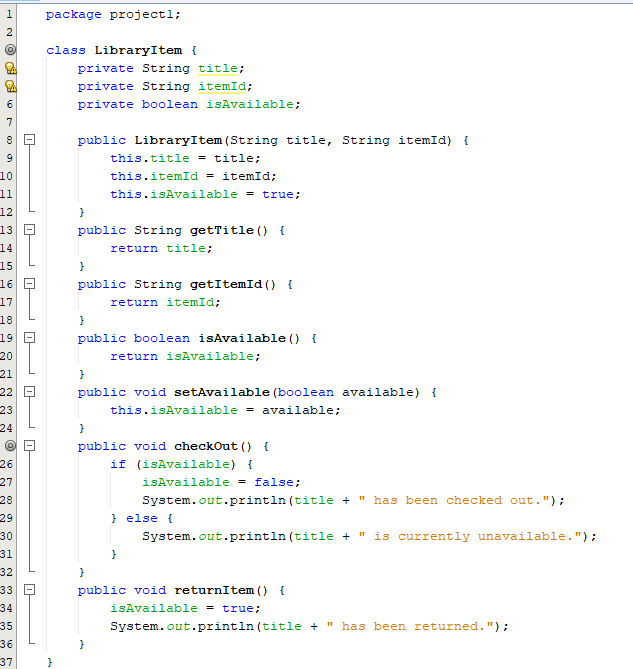
Q3. Implement a C++/Java program to demonstrate the use of design patterns.

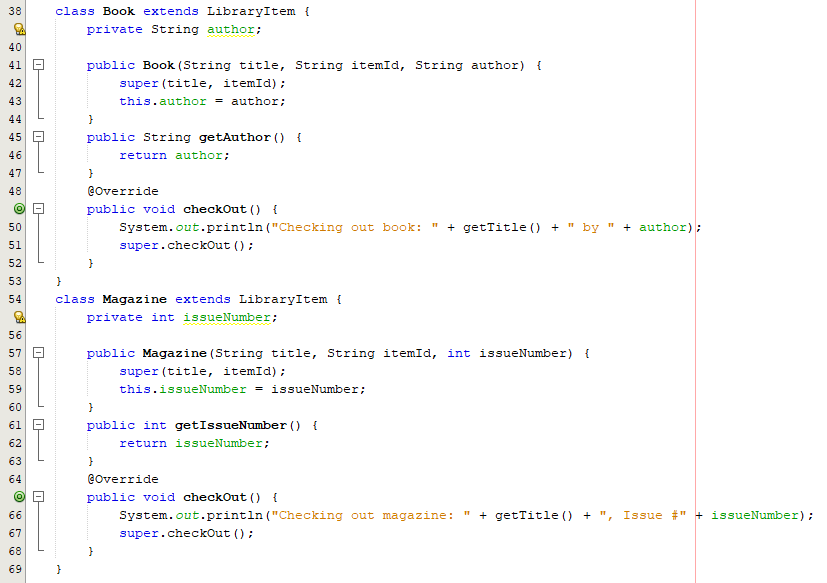
****

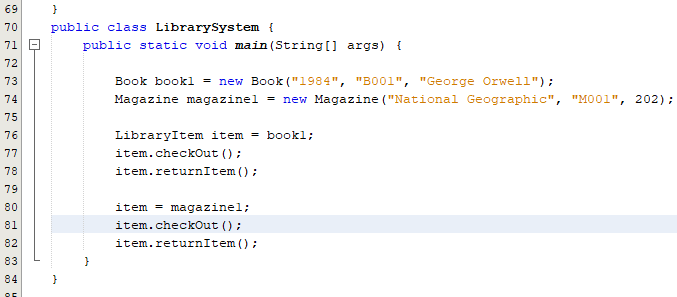
****

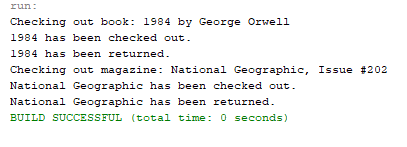
****

Q4. Write a C++/Java program to simulate object-oriented design.

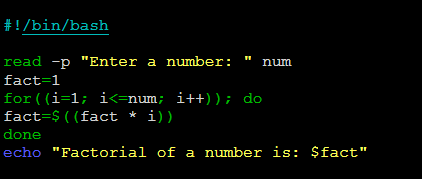
****

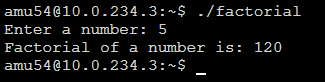
****

****

****

Q5. Write a program to find the factorial value of any number entered through the keyboard.





Q6. Two numbers are entered through the keyboard. Write a program to find the value of one number raised to the power of another.

