OOP is a programming paradigm that uses objects and classes to structure software. In OOP, an object is an instance of a class and encapsulates both data (attributes) and behavior (methods). The four core principles of OOP are:

1. **Encapsulation** – Bundling data and methods that operate on the data into a single unit or class.
2. **Abstraction** – Hiding the internal workings of an object and exposing only what’s necessary.
3. **Inheritance** – A mechanism where one class inherits properties and behaviors from another class.
4. **Polymorphism** – The ability for different classes to be treated as instances of the same class through inheritance, typically using method overriding or overloading.

Encapsulation:

Encapsulation is the concept of bundling data (variables) and methods (functions) that operate on the data into a single unit, typically a class. It restricts direct access to some of an object’s components, which can prevent unintended interference and misuse of the data. This is usually done by making some variables private and providing public getter and setter methods to access and update them.  
**Real-Time Example:** A **Mobile Phone**.

* The phone has a **power button** (a method) to turn on/off the phone. The user doesn’t need to know the inner details (like the internal circuitry, battery management, etc.). They can simply press the button to achieve the desired outcome.

Abstraction:

Abstraction is the principle of hiding the complexity of a system and exposing only the essential parts. In OOP, this means that objects are designed to provide a simplified interface, and the implementation details are hidden from the user. This can be achieved through abstract classes and interfaces in many programming languages.  
The **ATM machine** provides a simple interface to the user (insert card, input PIN, select transaction type, withdraw money), but it hides the complex operations behind the scenes,

Inheritance:

Inheritance allows a class to inherit properties and behaviors (methods) from another class. The class that is inherited from is called the **base class** or **parent class**, while the class that inherits is called the **derived class** or **child class**. Inheritance helps in reusability and creates a hierarchical relationship between classes.  
A **Car** and **Truck** are both types of **Vehicle**.

Polymorphism means "many shapes" and allows objects of different classes to be treated as objects of a common superclass. The two types of polymorphism are:

1. **Compile-time polymorphism (Method Overloading)** – Multiple methods with the same name but different parameters.
2. **Run-time polymorphism (Method Overriding)** – A subclass can provide a specific implementation of a method that is already defined in its superclass.

A **Payment system** that supports multiple methods of payment (Credit Card, Debit Card, PayPal, etc.) is example of Polymorphism.

**Association**

Association is a broad term that represents any relationship between two objects. It describes how objects are related but doesn't specify the nature of the relationship (i.e., whether the objects are dependent on each other or have a strong ownership). The relationship can be one-to-one, one-to-many, or many-to-many.  
  
A **Student** and a **Teacher** are associated because a student may have one or more teachers, and a teacher may teach multiple students. **The association here is simply a connection between the two, but neither object "owns" the other.**

**Aggregation**

* **Definition:**
  + **Aggregation** is a specialized form of association where one object **contains** or **is made up of** other objects, but the contained objects can exist independently of the parent object. Aggregation represents a "Has-A" relationship.
  + It shows that objects can have a relationship, but they don’t necessarily have a strong ownership.

**Real-World Example:**

* A **Library** has **Books**. The library contains books, but if the library is closed or destroyed, the books can still exist independently in other libraries or homes.

**Composition**

* **Definition:**
  + **Composition** is a stronger form of aggregation where one object **owns** another object. The lifetime of the contained object is dependent on the parent object. If the parent object is destroyed, the contained objects are also destroyed. It represents a "Contains-A" relationship with strong ownership.
* **Real-World Example:**
  + A **House** contains **Rooms**. If the house is destroyed, the rooms no longer exist (they can't exist independently without the house).