These fundamental OOP concepts enable developers to create modular, organized, maintainable, reusable and extensible software systems:

- **Classes**: Classes are blueprints for creating objects. They define the structure and behavior of objects by specifying attributes (data) and methods (functions) that instances of the class will possess.

- **Objects**: Objects are instances of classes. They represent real-world entities and contain data (attributes) and methods (functions) that operate on that data.

- **Encapsulation**: Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit, called a class. It restricts direct access to an object's internal state and enforces access through defined methods, thus hiding implementation details.

- **Abstraction**: Abstraction is the process of simplifying complex reality by modeling classes based on their essential properties and behaviors while ignoring non-essential details. It allows developers to focus on what an object does rather than how it does it.

- **Inheritance**: Inheritance allows a new class (subclass/derived class) to inherit attributes and methods from an existing class (superclass/base class). It promotes code reuse and establishes a hierarchical relationship between classes.

- **Polymorphism**:
    - **Compile-time Polymorphism**: Also known as method overloading, it involves defining multiple methods with the same name in a class, but with different parameters or types. The correct method is determined at compile time based on the method signature.
    - **Runtime Polymorphism**: Also known as method overriding, it occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method to execute is determined at runtime based on the object's actual class.

Abstract Classes and Interfaces

Abstract Classes:

- They serve as a blueprint or template for other classes, known as its subclasses or derived classes.
- They cannot be instantiated directly; they exist solely to be inherited from by other classes.
- They can include both method declarations (signatures) and method implementations. This flexibility allows you to have some default behaviour.
- Subclasses that inherit from an abstract class are required to implement the abstract methods declared in the base class.

Interfaces:

- They focus on defining a contract specifying a set of method signatures that a class must adhere to.
- They do not contain any method implementations; they solely declare the methods that implementing classes must provide.
- The primary purpose of interfaces is to establish a common set of behaviors that multiple, often unrelated, classes can adhere to.
- This promotes a high degree of flexibility and polymorphism in the codebase, as classes from different hierarchies can implement the same interface.

SOLID Principles

- **Single Responsibility Principle (SRP)**:
    - A class should have only one reason to change, meaning it should have a single responsibility or job.
    - It is only a guideline for designing.
    - Single Responsibility can be subjective and dependent on the specific context of application.
    - Example - Order Validation can be either a single class's responsibility or there can be multiple classes to validate user inputs, generating error messages and checking permissions.

- **Open-Closed Principle (OCP)**:
    - Software entities (classes, modules, functions) should be open for extension but closed for modification, allowing new functionality to be added without altering existing code.
    - Once code is written and tested, it shouldn't have to be modified to add new features.
    - Code should be extensible.
    - Example - Logger has Info and Error Levels, we should be able to add Debug Level seamlessly.

- **Liskov Substitution Principle (LSP)**:
    - Subtypes (derived classes) should be substitutable for their base types (parent classes) without affecting the correctness of the program.
    - Example - We should be able to replace employee with manager or intern seamlessly.

- **Interface Segregation Principle (ISP)**:
    - Clients should not be forced to depend on interfaces they don't use.
    - We should aim for smaller, more specific interfaces.
    - Example - In Swiggy, a customer shouldn't be forced to implement KYC like Delivery Partner and Restaurant Owner.

- **Dependency Inversion Principle (DIP)**:
    - High-level modules should not depend on low-level modules.
    - Both should depend on abstractions.
    - Abstractions should not depend on details; details should depend on abstractions.
    - High-level order processing modules should depend on an abstract "PaymentGateway" interface, not on specific payment service implementations like PayPal or CreditCardProcessor.