**1)What is software engineering?**

Software engineering is a discipline that applies engineering principles to the design, development, testing, deployment, and maintenance of software. It involves a systematic, disciplined, and quantifiable approach to the creation of software, ensuring that it meets user requirements, is reliable, maintainable, and scalable.

**2)why object oriented SE?**

Object-Oriented Software Engineering (OOSE) is a paradigm within software engineering that uses the concepts of "objects" and "classes" to model real-world entities and their interactions. It's based on the principles of object-oriented programming (OOP) and has become a popular approach for building software systems

**3)Difference between POP and OOP?**

**Procedural Programming (POP)**

Structure: POP organizes code into functions or procedures, focusing on a sequence of steps to perform tasks. Functions are the primary building blocks.

Data Handling: Data is often passed between functions, and there is less emphasis on encapsulation. Data can be accessed and manipulated freely, which can lead to a "global state" problem.

Abstraction: Abstraction is generally at the function level, and there is less emphasis on modeling complex systems as data structures or objects.

Encapsulation: Encapsulation is limited, as there are no inherent mechanisms to restrict direct data access.

Reusability: Reusability is achieved through function reuse and modularity, but it's generally less flexible compared to OOP.

Example Languages: C, Fortran, Pascal.

**Object-Oriented Programming (OOP)**

Structure: OOP organizes code into objects, which contain both data (attributes) and behavior (methods). Objects are instances of classes, which define the blueprint for the objects.

Data Handling: Data is encapsulated within objects, with access often controlled through methods or properties. This encapsulation leads to greater data integrity and security.

Abstraction: Abstraction is emphasized through classes and objects, allowing for more complex models that closely represent real-world entities.

Encapsulation: OOP has strong encapsulation, allowing objects to hide internal state and expose only what's necessary through defined interfaces.

Reusability: Reusability is achieved through inheritance and composition, allowing objects to inherit behavior and characteristics from other objects or combine existing objects to create new functionality.

Polymorphism: OOP supports polymorphism, allowing objects of different classes to be treated as instances of a common superclass, enabling flexible and extensible design.

Example Languages: Java, C++, Python, Ruby, C#.

**4)OOSE and lifecycle?**

Key aspects of software engineering include:

Requirements Gathering: Identifying and documenting the needs and constraints for a software project, often by collaborating with stakeholders.

Design: Creating a blueprint for the software, which includes the architecture, data structures, algorithms, and user interfaces.

Implementation: Writing the code based on the design, using various programming languages and tools.

Testing: Verifying that the software functions correctly and meets the specified requirements, which can include unit testing, integration testing, system testing, and user acceptance testing.

Deployment: Releasing the software to users or production environments, often involving additional steps like configuration and setup.

Maintenance and Support: Providing ongoing updates, fixing bugs, adding new features, and ensuring the software continues to meet users' needs.

**5)Legacy code**

Legacy code refers to software that is outdated, challenging to maintain, or built with obsolete technologies and practices. It often lacks sufficient documentation, relies on outdated frameworks or programming languages, and can contain complex dependencies, making it difficult to understand or change. The technical debt in legacy code can accumulate over time, leading to increased maintenance costs and security risks due to unpatched vulnerabilities. Because legacy code often supports critical business functions, replacing or significantly modifying it can be risky. To address these challenges, developers can employ various strategies: refactoring to improve code structure without altering behavior, rewriting or reengineering with modern technologies, and adding automated test coverage to ensure code stability. Gradual migration, where legacy components are incrementally replaced with newer ones, can help minimize risks. Additionally, encapsulating legacy code with wrappers can create a bridge to more modern systems, allowing for smoother integration. Clear documentation and knowledge sharing among team members are also essential to ensure the long-term maintainability and adaptability of legacy systems.