OOPs Concept

MUKESH KUMAR

AGENDA

- Introduction to OOP
- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Advantages of OOP
- Summary and Q&A

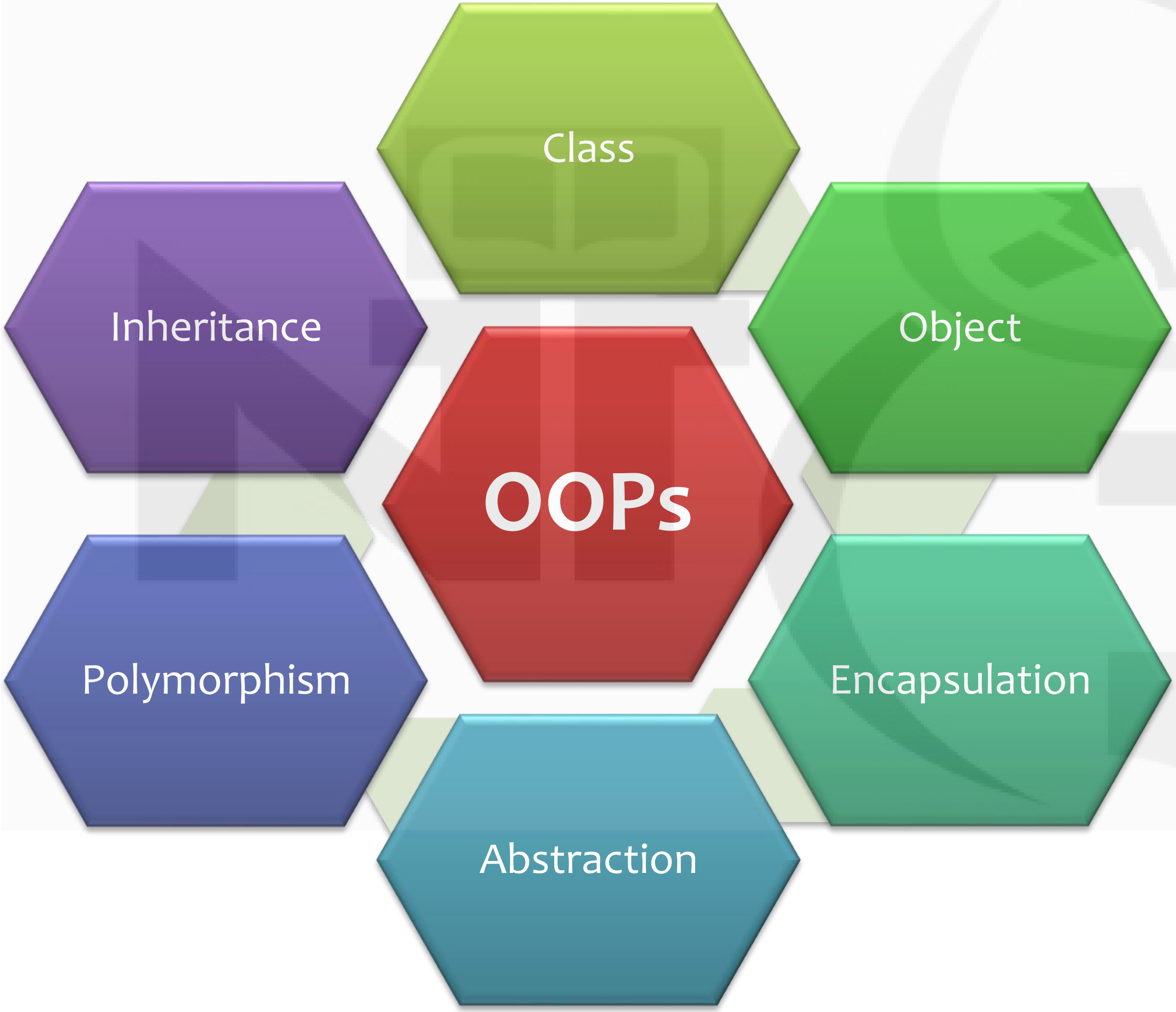
Introduction to OOP

- **Definition of Object-Oriented Programming (OOP)**: OOP is a programming paradigm that organizes code into objects containing attributes (data) and methods (functions).

- **Importance of OOP in Python**: Helps in code reusability, modularity, and easier debugging.

- **Real-world analogy**: A car can be considered an object with properties (color, model) and behaviors (start, stop).

OOPs Concepts



Core Concepts of OOP

- **Class:** Blueprint for objects.
- **Object:** Instance of a class.
- **Encapsulation:** Hiding data for security.
- **Inheritance:** Reusing existing code.
- **Polymorphism:** Multiple behaviors under one interface.
- **Abstraction:** Hiding complexity and exposing functionality.

OOPs

- Object-oriented programming (OOP) is a way of designing and organizing software using objects and classes.

ENCAPSULATION

Encapsulation

- It refers to the bundling of data (variables) and methods (functions) that operate on that data into a single unit, typically a class.

- Encapsulation restricts direct access to some of an object's components, which helps in **data hiding** and **protecting the integrity** of the data.

Key Points

Encapsulation serves **two main purposes:**

- ✓ **Bind the Data** – It groups data (variables) and methods (functions) into a single unit (class).

- ✓ **Hide the Data** – It restricts direct access to the data and allows controlled access through methods (getters & setters).

Data hiding

- **Access Modifiers –** Control the visibility of class members:
  - **public –** Accessible from anywhere.
  - **private –** Accessible only within the same class.
  - **protected –** Accessible within the same class and subclasses.

- **Getter and Setter Methods –** Provide controlled access to private variables.

Conventions

- **Public Members:** By default, all variables and methods in a Python class are public.
- **Protected Members:** They are denoted by a single underscore prefix (_).
- **Private Members:** Private members should not be accessed by anyone outside the class or any base classes. They are indicated by a double underscore prefix (__).

Using Access Modifier in Python

- Show class example

Key Takeaways

Public (self.variable) – Can be accessed from anywhere.

Protected (self._variable) – Can be accessed within the class and subclasses (by convention, not strictly enforced).

Private (self.__variable) – Can only be accessed inside the class; to access it, use a getter method.

Benefits of Encapsulation

- **Security:**
  - It protects data from outside interference and misuse, ensuring that it is only modified in controlled ways.
  - Only necessary information is exposed, and implementation details are concealed from the outside world, making the code more secure.
- **Modularity:** Encapsulation promotes a modular design by ensuring that objects manage their own state.

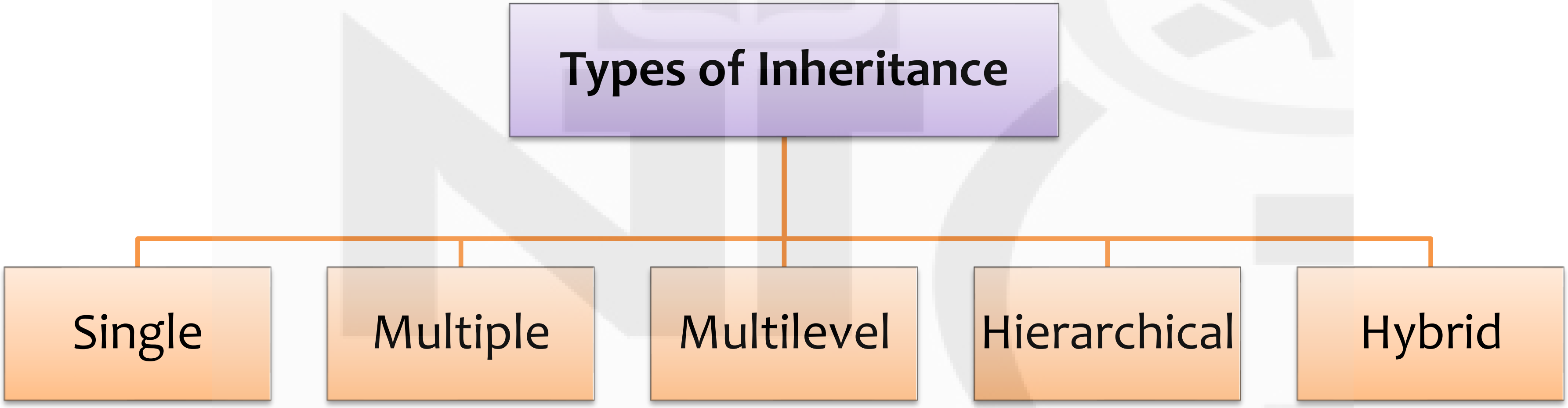
Benefits of Encapsulation

- **Maintainability**: Hiding implementation details and exposing only necessary methods aids in maintaining a clean codebase.
- **Reusability:** It allows for the creation of objects with clearly defined properties and behaviors, which facilitates the reuse of the code throughout the program.

INHERITANCE

Inheritance

- It allows a new class (**child/subclass**) to derive properties and behaviors (methods and attributes) from an existing class (**parent/superclass**).



Types of Inheritance

Single

Multiple

Multilevel

Hierarchical

Hybrid

Inheritance Types

- **Single Inheritance** – One child class inherits from one parent class.
- **Multiple Inheritance** – A child class inherits from multiple parent classes.
- **Multilevel Inheritance** – A class inherits from another derived class.
- **Hierarchical Inheritance** – Multiple child classes inherit from the same parent class.
- **Hybrid Inheritance** – A combination of different types of inheritance.

Inheritance Types

- Covered in py files

Inheritance Summary

Type of Inheritance	Description
Single Inheritance	One child inherits from one parent
Multiple Inheritance	One child inherits from multiple parents
Multilevel Inheritance	Parent → Child → Grandchild
Hierarchical Inheritance	One parent, multiple children
Hybrid Inheritance	Combination of different types

Why Use Inheritance?

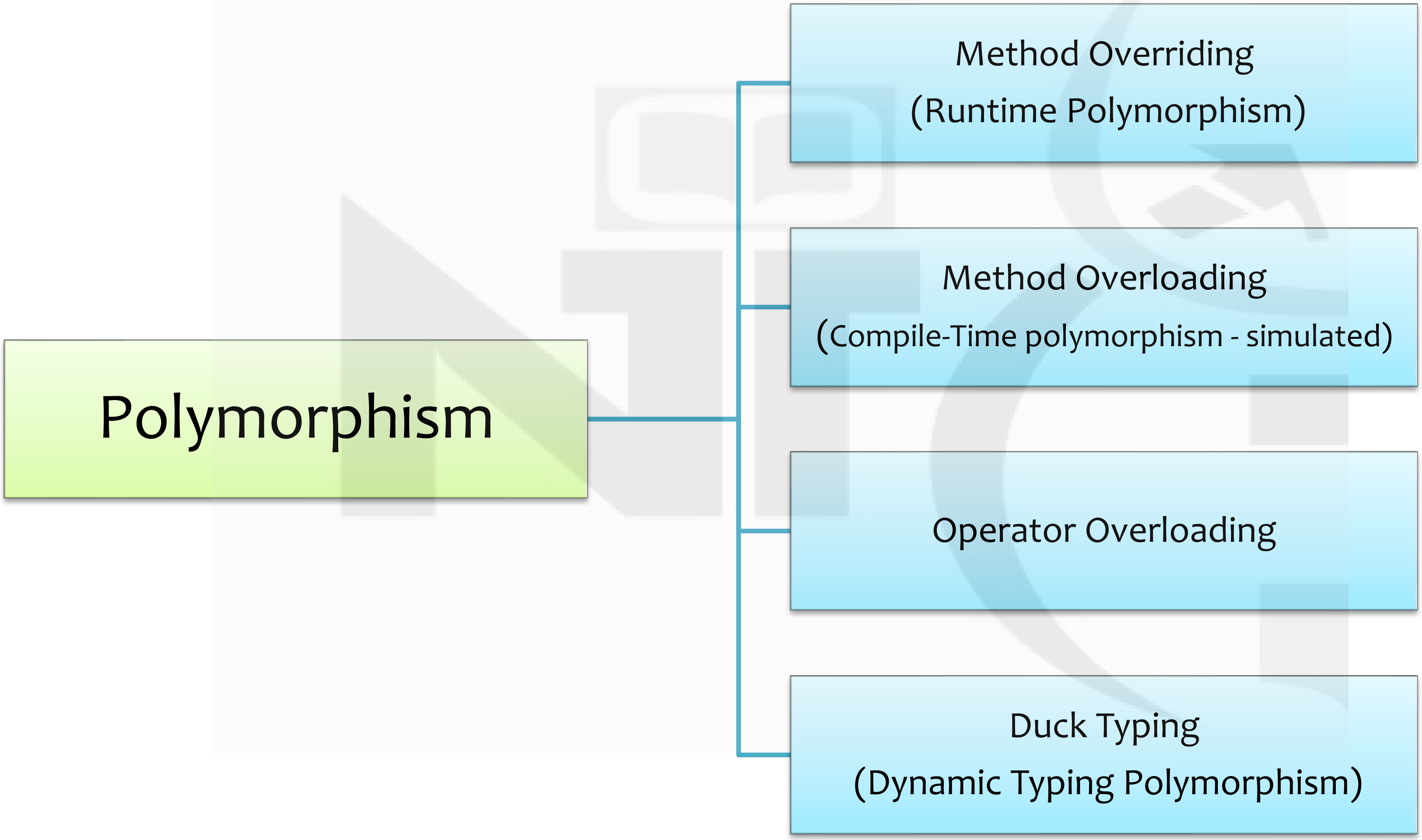
- **Code Reusability** – Avoids rewriting code by using functionality from a parent class.
- **Extensibility** – Allows modification or addition of new features without altering the original class.
- **Maintainability** – Reduces duplication, making code easier to manage and update.
- **Polymorphism Support** – Enables methods in child classes to override or extend those in the parent class.

POLYMORPHISM

Polymorphism

- The ability to take multiple forms.
- Polymorphism, meaning "many forms," is a concept in object-oriented programming where a function, method, or object can take on different forms in different contexts.
- In Python, polymorphism allows you to use the same function name for different types of objects, adapting its behavior based on the object's type

Python Polymorphism



Polymorphism

Method Overriding
(Runtime Polymorphism)

Method Overloading
(Compile-Time polymorphism - simulated)

Operator Overloading

Duck Typing
(Dynamic Typing Polymorphism)

Method Overriding
(Runtime Polymorphism)

- Method overriding occurs when a **child class provides a specific implementation of a method** that is already defined in its **parent class**.

- Refer the py files for examples

Method Overloading
(Simulated in Python)

- Method overloading means defining **multiple methods with the same name but different parameters.**

- Unlike **Java or C++,** Python does **not** support method overloading **directly** because Python functions only recognize the last defined method.

- Refer the py files for examples

Operator Overloading

- Python allows overloading built-in operators (like +, -, *, etc.) using magic methods (also called dunder methods, e.g., __add__, __sub__, etc.).

Common Magic Methods for Operator Overloading

Operator	Magic Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Duck Typing
(Dynamic Typing Polymorphism)

- Python follows **duck typing**, meaning **if an object behaves like a duck, we treat it as a duck**.

- This means that **Python does not enforce strict type checking**—if an object supports a method, we can call it, regardless of its class.

- Refer the py files for examples

Polymorphism Key points

Type of Polymorphism	Description	Example
Method Overriding	Child class redefines a method from the parent class	sound() in Dog and Cat classes
Method Overloading	A single method handles multiple argument types	show(a=None, b=None)
Operator Overloading	Overloading operators like +, -, *, etc.	__add__() in Point class
Duck Typing	Objects are used based on behavior, not class type	Function make_sound() calling sound()

Polymorphism Summary

- Polymorphism makes code more flexible, reusable, and maintainable.
- Python does not support strict method overloading like Java but allows it through default arguments or *args.
- Operator overloading makes custom objects work seamlessly with built-in operators.
- Duck typing allows writing functions that work on multiple types without explicit type checking.

ABSTRACTION

Abstraction

- **Definition:** Hides implementation details and exposes only necessary functionality.
- Refer the py files for examples

Key Aspects of Abstraction

- **Hiding Complexity:** Abstraction hides the internal workings of a system, presenting a simplified view to the user.

- **Essential Information:** It exposes only relevant data about an object, hiding all other details.

- **Managing Complexity:** Abstraction aids in managing complexity, enhancing code readability, and promoting reusability.

How is Abstraction Implemented in Python

- Python provides abstraction using abstract classes and abstract methods through the ABC (Abstract Base Class) module.

Abstract Class

- An abstract class is a class that cannot be instantiated.
- It serves as a blueprint for other classes.

Abstract Method

- An abstract method is a method that is declared but does not have an implementation in the base class.
- Any subclass inheriting from an abstract class must implement all abstract methods.

- If you attempt to create an instance of an abstract class without implementing all abstract methods, Python will raise a TypeError.

Partial Abstraction

- A class can have both abstract and concrete methods. This allows a base class to provide some common functionality while still enforcing the implementation of critical methods.

- Refer the py files for examples

Summary

- ✅ **Abstraction = Hiding the Implementation, Sharing Only the Structure**
  ✅ **Enforces a Common Contract** – Ensures every subclass follows a specific structure.
  ✅ **Separates Design from Implementation** – Teams work independently on their own implementations.
  ✅ **Scalability** – New implementations can be added without modifying existing code.

Summary and Q&A

- **Recap of OOP concepts:**
  - Classes and Objects
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction