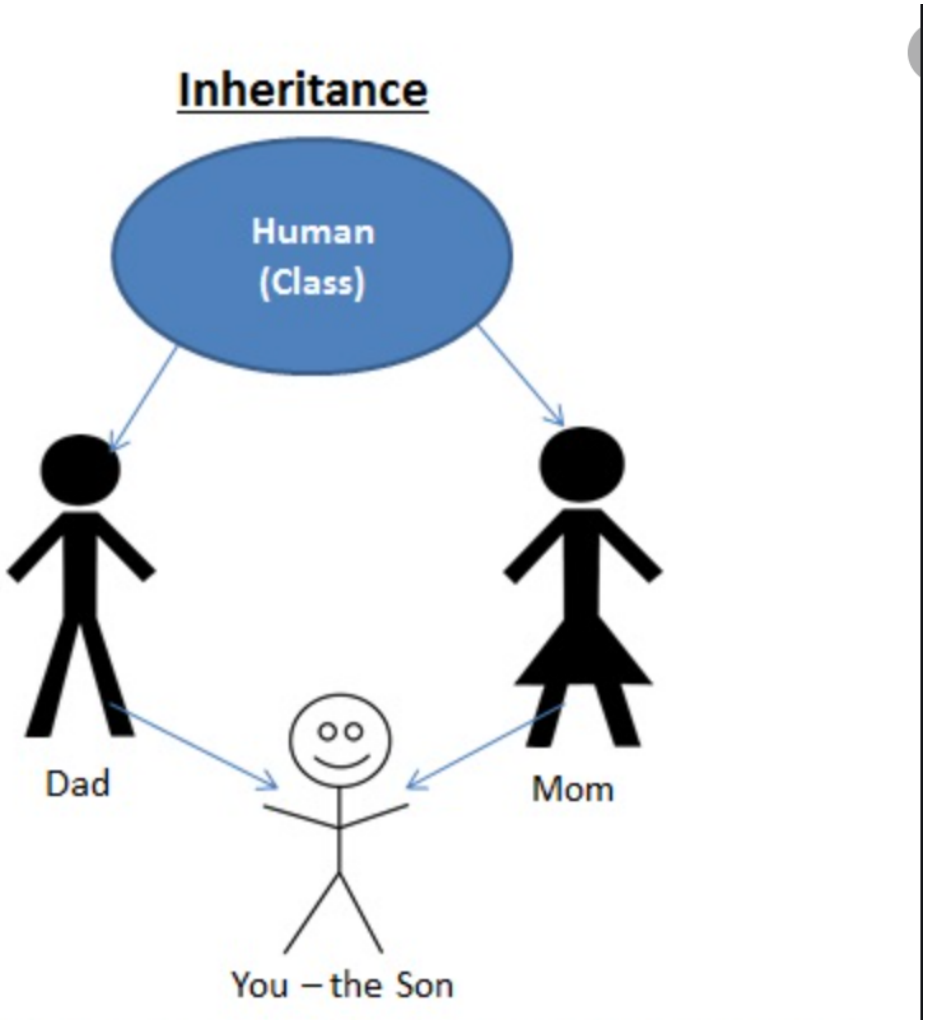
Python OOPs Inheritance and Encapsulations

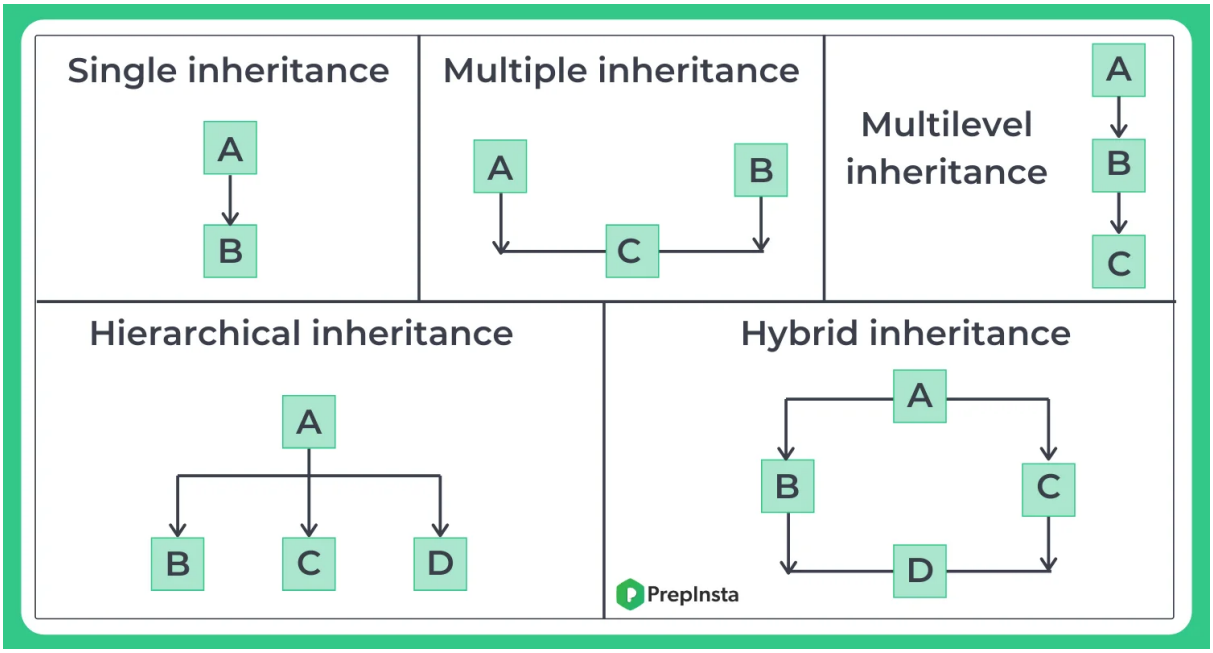
What is Inheritance ?

Inheritance allows a child class to inherit **properties and methods** from a parent class. There are different types of inheritance in Python. Useful for organizing similar entities that share common properties and behaviors while also allowing for specialization. It helps avoid code duplication and increases the maintainability of your codebase.



- **Single Inheritance**: A child class inherits from one parent class.
- **Multiple Inheritance**: A child class inherits from more than one parent class.
- **Multilevel Inheritance**: A class inherits from a class that has already inherited from another class.
- **Hierarchical Inheritance**: Multiple child classes inherit from the same parent class.

- **Hybrid Inheritance**: A combination of more than one type of inheritance.



Single Inheritance Example:

```
# Parent Class
class Vehicle:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def info(self):
        return f"Brand: {self.brand}, Speed: {self.speed}"

# Child Class
class Car(Vehicle):
    def __init__(self, brand, speed, fuel_type):
        # Call the parent class constructor
        super().__init__(brand, speed)
        self.fuel_type = fuel_type

    def car_info(self):
        return f"{self.info()}, Fuel Type: {self.fuel_type}"

# Create an instance of Car
car = Car("Toyota", 180, "Petrol")
print(car.car_info())  # Inherits and extends functionality
```

Multiple Inheritance Example:

```
# Parent Class 1
class Engine:
    def __init__(self, engine_type):
        self.engine_type = engine_type

    def engine_info(self):
        return f"Engine Type: {self.engine_type}"

# Parent Class 2
class Vehicle:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def vehicle_info(self):
        return f"Brand: {self.brand}, Speed: {self.speed}"

# Child Class inheriting from both Engine and Vehicle
class Car(Engine, Vehicle):
    def __init__(self, brand, speed, engine_type):
        Vehicle.__init__(self, brand, speed)  # Call Vehicle construct
        Engine.__init__(self, engine_type)    # Call Engine constructo

    def car_info(self):
        return f"{self.vehicle_info()}, {self.engine_info()}"

# Create an instance of Car
car = Car("Tesla", 250, "Electric")
print(car.car_info())  # Uses methods from both parent classes
```

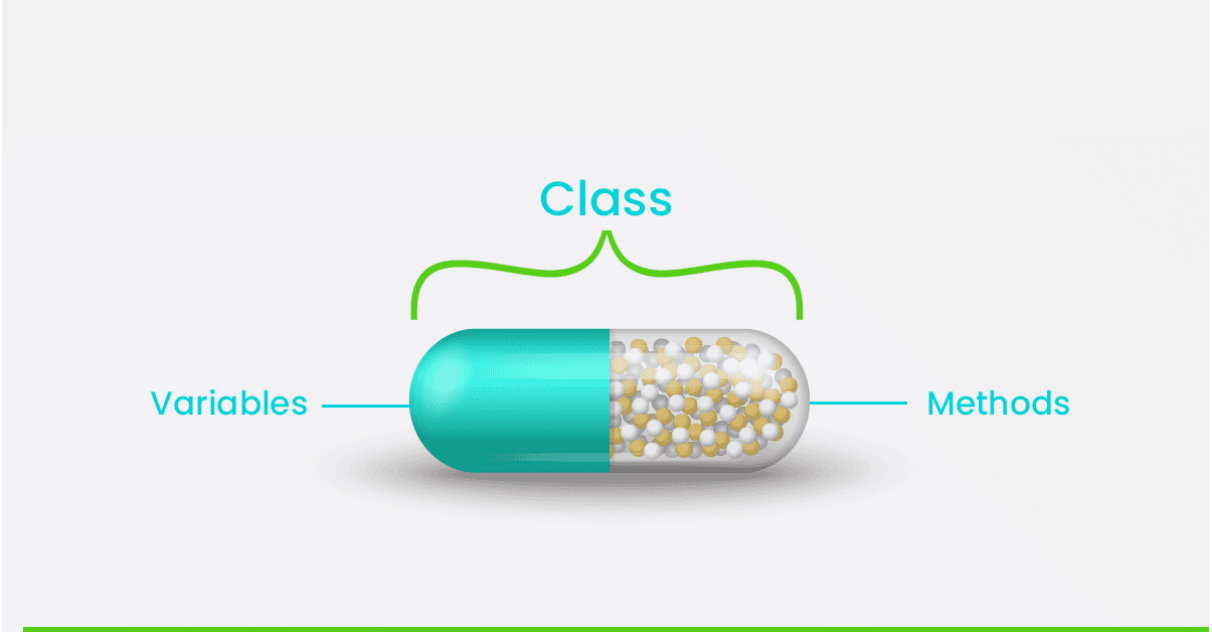
Multilevel Inheritance:

```
# Base Class
class Animal:
    def __init__(self, name):
        self.name = name

    def sound(self):
        return "Some generic animal sound"

# Intermediate Class
class Mammal(Animal):
```

```
    def __init__(self, name, habitat):
        super().__init__(name)
        self.habitat = habitat

    def mammal_info(self):
        return f"{self.name} lives in {self.habitat}"

# Derived Class
class Dog(Mammal):
    def __init__(self, name, habitat, breed):
        super().__init__(name, habitat)
        self.breed = breed

    def dog_info(self):
        return f"{self.mammal_info()}, Breed: {self.breed}"

# Create an instance of Dog
dog = Dog("Buddy", "Domestic", "Golden Retriever")
print(dog.dog_info())  # Inherits across multiple levels
```

What is Encapsulation?

Encapsulation is the concept of restricting access to certain attributes or methods to protect the internal state of an object. This is often done using **private** (`__`) or **protected** (`_`) attributes. Encapsulation helps in hiding data to prevent direct modification from outside the class.



Protects sensitive data from unauthorized access and modifications. This is crucial for real-world applications such as bank systems, user management, and employee management where certain information needs to be secure.

Private Members Example (with Getter and Setter methods):

```
class Employee:
    def __init__(self, name, salary):
        self.name = name           # Public attribute
        self.__salary = salary      # Private attribute

    # Getter method for salary
    def get_salary(self):
        return self.__salary

    # Setter method for salary
    def set_salary(self, salary):
        if salary > 0:
            self.__salary = salary
        else:
            print("Invalid salary amount")

# Create an instance of Employee
emp = Employee("John", 5000)

# Accessing the public attribute
print(emp.name)

# Accessing the private attribute via a getter
print(emp.get_salary())

# Modifying the private attribute via a setter
emp.set_salary(6000)
print(emp.get_salary())

# Attempting to access the private attribute directly (will raise an e
# print(emp.__salary)  # This will throw an AttributeError
```

In the above example, the `__salary` attribute is private and can only be accessed or modified through the `get_salary()` and `set_salary()` methods. Direct access to `__salary` is not allowed.

Protected Members Example (with Direct Access):

```python
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner          # Public attribute
        self._balance = balance     # Protected attribute (by conventi

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
        else:
            print("Invalid deposit amount")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
        else:
            print("Invalid withdrawal amount")

    def get_balance(self):
        return self._balance

# Create an instance of BankAccount
account = BankAccount("Alice", 1000)

# Access the protected member directly (allowed but discouraged)
print(account._balance)

# Deposit and withdraw
account.deposit(500)
account.withdraw(300)
print(account.get_balance())
```

In this case, the `_balance` attribute is **protected**, which means it is intended for internal use but is still accessible from outside the class. However, it's good practice to use getter and setter methods for protected attributes to ensure proper access control.

Examples:

Inheritance:

Inheritance in E-Commerce System:

```
# Parent Class
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def product_info(self):
        return f"Product: {self.name}, Price: {self.price}"

# Child Class for Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price)
        self.warranty = warranty

    def electronics_info(self):
        return f"{self.product_info()}, Warranty: {self.warranty} year

# Child Class for Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    def clothing_info(self):
        return f"{self.product_info()}, Size: {self.size}"

# Create instances of Electronics and Clothing
laptop = Electronics("Laptop", 1200, 2)
shirt = Clothing("Shirt", 40, "L")

print(laptop.electronics_info())
print(shirt.clothing_info())
```

Inheritance in a User Management System:

In a user management system, different types of users, such as **Admin**, **Editor**, and **Viewer**, may share common properties but have distinct roles and permissions. We can use inheritance to define common attributes in a parent class and extend the functionality for different types of users.

```python
# Parent Class
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

    def get_user_info(self):
        return f"User: {self.username}, Email: {self.email}"

    def has_permission(self):
        return False  # Default permission for regular users

# Child Class: Admin
class Admin(User):
    def __init__(self, username, email):
        super().__init__(username, email)

    def has_permission(self):
        return True  # Admin has full permissions

# Child Class: Editor
class Editor(User):
    def __init__(self, username, email):
        super().__init__(username, email)

    def has_permission(self):
        return True  # Editor has certain permissions

# Child Class: Viewer
class Viewer(User):
    def __init__(self, username, email):
        super().__init__(username, email)

    # Viewer inherits default permission (False)

# Test the system
admin = Admin("admin123", "admin@example.com")
editor = Editor("editor456", "editor@example.com")
viewer = Viewer("viewer789", "viewer@example.com")

print(admin.get_user_info(), "- Permission:", admin.has_permission())
```

```
print(editor.get_user_info(), "- Permission:", editor.has_permission()
print(viewer.get_user_info(), "- Permission:", viewer.has_permission()
```

In this example:

- **User** is the parent class containing common attributes like `username` and `email`.
- The `Admin`, `Editor`, and `Viewer` classes inherit from `User` but override behavior based on their roles. This is useful for user management systems where different users have different permissions.

Inheritance in a Shopping Cart System:

A typical e-commerce platform can have multiple types of products such as **Electronics**, **Clothing**, and **Food Items**. These products share common attributes like name and price, but they have specific attributes related to their type. Using inheritance, we can model this efficiently.

```
# Parent Class: Product
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_product_info(self):
        return f"Product: {self.name}, Price: ${self.price}"

# Child Class: Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price)
        self.warranty = warranty  # Additional attribute for electroni

    def get_product_info(self):
        return f"{super().get_product_info()}, Warranty: {self.warrant

# Child Class: Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size  # Additional attribute for clothing

    def get_product_info(self):
        return f"{super().get_product_info()}, Size: {self.size}"

# Child Class: Food Item
```

```
class FoodItem(Product):
    def __init__(self, name, price, expiry_date):
        super().__init__(name, price)
        self.expiry_date = expiry_date  # Additional attribute for foo

    def get_product_info(self):
        return f"{super().get_product_info()}, Expiry Date: {self.expi

# Create product instances
laptop = Electronics("Laptop", 1200, 2)
t_shirt = Clothing("T-shirt", 25, "M")
apple = FoodItem("Apple", 1, "2024-12-31")

# Output product information
print(laptop.get_product_info())  # Electronics info
print(t_shirt.get_product_info())  # Clothing info
print(apple.get_product_info())   # Food item info
```

In this shopping cart system:

- **Product** is the parent class with common attributes like `name` and `price`.
- **Electronics**, **Clothing**, and **FoodItem** classes inherit from `Product` and extend it with specific attributes (like `warranty`, `size`, and `expiry_date`).

Encapsulation:

Encapsulation in an Employee Management System:
An employee management system may contain sensitive information like an employee's **salary**, which should not be modified directly. The system should allow only authorized actions, like adjusting salary via promotions or deductions via penalties.

```
class Employee:
    def __init__(self, name, position, salary):
        self.name = name
        self.position = position
        self.__salary = salary  # Private attribute

    # Getter for salary (read-only access)
    def get_salary(self):
        return self.__salary

    # Method to increase salary (controlled access)
    def apply_promotion(self, raise_amount):
```

```
        if raise_amount > 0:
            self.__salary += raise_amount
            print(f"Promotion applied! New salary: ${self.__salary}")
        else:
            print("Invalid raise amount")

    # Method to apply penalty (controlled access)
    def apply_penalty(self, penalty_amount):
        if penalty_amount > 0 and penalty_amount <= self.__salary:
            self.__salary -= penalty_amount
            print(f"Penalty applied! New salary: ${self.__salary}")
        else:
            print("Invalid penalty amount")

# Create an employee
employee = Employee("Alice", "Software Engineer", 70000)

# Check employee salary using the getter method
print(f"Initial Salary: ${employee.get_salary()}")

# Apply a promotion and penalty
employee.apply_promotion(5000)   # Increase salary
employee.apply_penalty(2000)     # Deduct salary

# Attempting to directly access or modify salary will raise an error
# print(employee.__salary)  # Raises AttributeError
```

In this system:

- The employee's salary is encapsulated, preventing direct modification.
- Salary adjustments can only be made through controlled methods like `apply_promotion` and `apply_penalty`, ensuring that changes are made in a valid and secure manner.

Encapsulation in a Banking Application::
In a real banking system, you want to protect sensitive information like the user's **account balance** and **transaction history**. Direct modification should be restricted, and users should only interact with the system via predefined methods (such as `deposit` and `withdraw`).

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute to protect balan

    # Getter method for balance
```

```
    def get_balance(self):
        return self.__balance

    # Deposit money into the account
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"${amount} deposited. New balance: ${self.__balance
        else:
            print("Deposit amount must be positive")

    # Withdraw money from the account
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"${amount} withdrawn. Remaining balance: ${self.__b
        else:
            print("Invalid withdrawal amount or insufficient funds")

# Creating a bank account for a customer
customer_account = BankAccount("John Doe", 1000)

# Attempting to deposit and withdraw money
customer_account.deposit(500)   # Depositing $500
customer_account.withdraw(200)  # Withdrawing $200

# Check balance using the getter
print(f"Current balance: ${customer_account.get_balance()}")

# Trying to access the private attribute directly (will raise an error
# print(customer_account.__balance)  # Raises AttributeError
```

Here:

- The `__balance` attribute is private, and users can only interact with it via `deposit` and `withdraw` methods.
- This prevents accidental modification of the account balance and provides a controlled way to manage bank transactions.

Encapsulation for Secure Transaction:

```
class Customer:
    def __init__(self, name, credit_card_number):
        self.name = name
        self.__credit_card_number = credit_card_number  # Private attr
```

```
    # Getter method for credit card number
    def get_credit_card(self):
        return f"**** **** **** {self.__credit_card_number[-4:]}"  # M

    # Setter method for credit card number
    def set_credit_card(self, credit_card_number):
        if len(credit_card_number) == 16:
            self.__credit_card_number = credit_card_number
        else:
            print("Invalid credit card number")

# Create an instance of Customer
customer = Customer("Alice", "1234567812345678")

# Accessing masked credit card info
print(customer.get_credit_card())

# Updating credit card number
customer.set_credit_card("8765432187654321")
print(customer.get_credit_card())
```