Complete Beginner's Guide to Inheritance and Polymorphism

Part 1: Understanding Inheritance - The Family Tree Concept

Real-World Analogy

Think of inheritance like a family tree:

- **Grandparents** have certain traits (eye color, height)
- **Parents** inherit those traits but may add their own (profession, personality)
- **Children** inherit from parents and grandparents but have their unique qualities

In programming, this means:

- **Parent Class (Base/Super Class)**: The original class with common features
- **Child Class (Derived/Sub Class)**: Inherits everything from parent + adds its own features

---

Part 2: Single Inheritance - One Parent, One Child

Basic Example: Animal Kingdom

python

```python
# Parent Class (Base Class)
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
        self.is_alive = True

    def eat(self):
        return f"{self.name} is eating"

    def sleep(self):
        return f"{self.name} is sleeping"

    def make_sound(self):
        return f"{self.name} makes a generic animal sound"

# Child Class (inherits from Animal)
class Dog(Animal):
    def __init__(self, name, breed):
        # Call parent's __init__ using super()
        super().__init__(name, "Canine")  # All dogs are Canines
        self.breed = breed
        self.tricks = []

    # Override parent's method with dog-specific behavior
    def make_sound(self):
        return f"{self.name} barks: Woof! Woof!"

    # Add dog-specific methods
    def learn_trick(self, trick):
        self.tricks.append(trick)
        return f"{self.name} learned to {trick}!"

    def fetch(self):
        return f"{self.name} fetches the ball!"

# Using inheritance
my_dog = Dog("Buddy", "Golden Retriever")

# Methods inherited from Animal
print(my_dog.eat())        # Buddy is eating
print(my_dog.sleep())      # Buddy is sleeping
```

```python
# Overridden method (Dog's version)
print(my_dog.make_sound()) # Buddy barks: Woof! Woof!

# Dog-specific methods
print(my_dog.learn_trick("sit"))  # Buddy learned to sit!
print(my_dog.fetch())          # Buddy fetches the ball!

# Inherited attributes
print(f"Species: {my_dog.species}")  # Species: Canine
print(f"Alive: {my_dog.is_alive}")   # Alive: True
```

Understanding super()

`super()` is like calling your parent. It lets you use the parent class's methods:

python

```python
class Vehicle:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
        print(f"Vehicle created: {brand} {year}")

    def start_engine(self):
        return "Engine starting..."

class Car(Vehicle):
    def __init__(self, brand, year, doors):
        # Call parent's __init__ first
        super().__init__(brand, year)
        self.doors = doors
        print(f"Car details: {doors} doors")

    def start_engine(self):
        # Use parent's method AND add more
        parent_result = super().start_engine()
        return f"{parent_result} Car is ready to drive!"

my_car = Car("Toyota", 2023, 4)
# Output:
# Vehicle created: Toyota 2023
# Car details: 4 doors

print(my_car.start_engine())
# Output: Engine starting... Car is ready to drive!
```

Part 3: Multiple Inheritance - Multiple Parents

The Mixin Concept

Sometimes an object can have multiple "types" of behavior:

```python
```

```python
# Parent Class 1: Flying ability
class Flyer:
    def __init__(self):
        self.can_fly = True
        self.altitude = 0

    def fly(self):
        self.altitude = 1000
        return f"Flying at {self.altitude} feet!"

    def land(self):
        self.altitude = 0
        return "Landed safely!"

# Parent Class 2: Swimming ability
class Swimmer:
    def __init__(self):
        self.can_swim = True
        self.depth = 0

    def swim(self):
        self.depth = 10
        return f"Swimming at {self.depth} feet deep!"

    def surface(self):
        self.depth = 0
        return "Surfaced!"

# Child Class inheriting from BOTH parents
class Duck(Flyer, Swimmer):
    def __init__(self, name):
        # Call both parents' __init__
        Flyer.__init__(self)
        Swimmer.__init__(self)
        self.name = name

    def quack(self):
        return f"{self.name} says: Quack!"

# Using multiple inheritance
donald = Duck("Donald")

print(donald.quack())    # Donald says: Quack!
```

```python
print(donald.fly())     # Flying at 1000 feet!
print(donald.swim())    # Swimming at 10 feet deep!
print(donald.land())    # Landed safely!
print(donald.surface()) # Surfaced!
```

The Diamond Problem and MRO

python

```python
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(A):
    def greet(self):
        return "Hello from C"

class D(B, C):  # Multiple inheritance
    pass

# Which greet() method will be called?
obj = D()
print(obj.greet())  # Output: "Hello from B"

# Check the Method Resolution Order
print(D.__mro__)
# Output: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)

# Python searches in this order: D -> B -> C -> A -> object
```

MRO Rules:

1. Child classes come before parent classes
2. Left-to-right order in inheritance list
3. A class appears only once in MRO

Part 4: Multilevel Inheritance - Inheritance Chain

Building a Hierarchy

python

```python
# Grandparent Class
class LivingThing:
    def __init__(self, name):
        self.name = name
        self.is_alive = True

    def breathe(self):
        return f"{self.name} is breathing"

# Parent Class (inherits from LivingThing)
class Animal(LivingThing):
    def __init__(self, name, species):
        super().__init__(name)
        self.species = species

    def move(self):
        return f"{self.name} is moving"

    def eat(self):
        return f"{self.name} is eating"

# Child Class (inherits from Animal, which inherits from LivingThing)
class Mammal(Animal):
    def __init__(self, name, species, fur_color):
        super().__init__(name, species)
        self.fur_color = fur_color
        self.warm_blooded = True

    def nurse_young(self):
        return f"{self.name} is nursing its young"

# Grandchild Class
class Dog(Mammal):
    def __init__(self, name, breed, fur_color):
        super().__init__(name, "Canine", fur_color)
        self.breed = breed

    def bark(self):
        return f"{self.name} barks!"

# Using multilevel inheritance
my_dog = Dog("Rex", "German Shepherd", "Brown")
```

```
# Methods from all levels of inheritance
print(my_dog.breathe())    # From LivingThing
print(my_dog.move())       # From Animal
print(my_dog.nurse_young()) # From Mammal
print(my_dog.bark())       # From Dog

print(f"Breed: {my_dog.breed}")         # Dog attribute
print(f"Warm-blooded: {my_dog.warm_blooded}") # Mammal attribute
print(f"Species: {my_dog.species}")     # Animal attribute
print(f"Alive: {my_dog.is_alive}")      # LivingThing attribute
```

Part 5: Polymorphism - Many Forms, Same Interface

What is Polymorphism?

Poly = Many, **Morph** = Forms. One method name, different behaviors!

Method Overriding - Same Method, Different Behaviors

python

```python
class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        return "Area calculation not implemented"

    def perimeter(self):
        return "Perimeter calculation not implemented"

class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):  # Override parent method
        return self.width * self.height

    def perimeter(self):  # Override parent method
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):  # Override parent method
        return 3.14159 * self.radius ** 2

    def perimeter(self):  # Override parent method
        return 2 * 3.14159 * self.radius

# Polymorphism in action!
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Rectangle(2, 7)
]

# Same method call, different behaviors
for shape in shapes:
    print(f"{shape.name}:")
```

```
    print(f"  Area: {shape.area()}")
    print(f"  Perimeter: {shape.perimeter()}")
    print()

# Output:
# Rectangle:
#   Area: 15
#   Perimeter: 16
#
# Circle:
#   Area: 50.26544
#   Perimeter: 25.13274
#
# Rectangle:
#   Area: 14
#   Perimeter: 18
```

Part 6: Duck Typing - If It Quacks Like a Duck...

The Philosophy

"If it walks like a duck and quacks like a duck, it's a duck!"
In Python, we care about what an object CAN DO, not what it IS.

```
python
```

```python
class Duck:
    def swim(self):
        return "Duck is swimming"

    def fly(self):
        return "Duck is flying"

    def quack(self):
        return "Quack!"

class Airplane:
    def fly(self):
        return "Airplane is flying"

    def make_noise(self):
        return "Vroooom!"

class Fish:
    def swim(self):
        return "Fish is swimming"

    def breathe_underwater(self):
        return "Fish breathes through gills"

# Duck typing in action - we don't care about the TYPE
def make_it_fly(thing):
    # We assume if it has a fly() method, it can fly
    try:
        return thing.fly()
    except AttributeError:
        return f"{type(thing).__name__} can't fly!"

def make_it_swim(thing):
    # We assume if it has a swim() method, it can swim
    try:
        return thing.swim()
    except AttributeError:
        return f"{type(thing).__name__} can't swim!"

# Testing with different objects
duck = Duck()
plane = Airplane()
fish = Fish()
```

```python
print("Flying test:")
print(make_it_fly(duck))  # Duck is flying
print(make_it_fly(plane))  # Airplane is flying
print(make_it_fly(fish))  # Fish can't fly!

print("\nSwimming test:")
print(make_it_swim(duck))  # Duck is swimming
print(make_it_swim(fish))  # Fish is swimming
print(make_it_swim(plane)) # Airplane can't swim!
```

Advanced Duck Typing Example

python

```python
class FileWriter:
    def write(self, data):
        with open("file.txt", "w") as f:
            f.write(data)
        return "Written to file"

class DatabaseWriter:
    def write(self, data):
        # Imagine this writes to a database
        return f"Written '{data}' to database"

class EmailSender:
    def write(self, data):
        # Imagine this sends an email
        return f"Emailed: {data}"

# Polymorphic function using duck typing
def save_data(writer, data):
    # We don't care WHAT type of writer it is
    # We only care that it has a write() method
    return writer.write(data)

# All these work because they all have write() method
writers = [
    FileWriter(),
    DatabaseWriter(),
    EmailSender()
]

data = "Important information"

for writer in writers:
    result = save_data(writer, data)
    print(f"{type(writer).__name__}: {result}")

# Output:
# FileWriter: Written to file
# DatabaseWriter: Written 'Important information' to database
# EmailSender: Emailed: Important information
```

Part 7: Real-World Example - Game Character System

python

```python
# Base Character Class
class Character:
    def __init__(self, name, health, attack_power):
        self.name = name
        self.health = health
        self.max_health = health
        self.attack_power = attack_power
        self.is_alive = True

    def attack(self, target):
        damage = self.attack_power
        target.take_damage(damage)
        return f"{self.name} attacks {target.name} for {damage} damage!"

    def take_damage(self, damage):
        self.health -= damage
        if self.health <= 0:
            self.health = 0
            self.is_alive = False
            return f"{self.name} has been defeated!"
        return f"{self.name} takes {damage} damage! Health: {self.health}/{self.max_health}"

    def heal(self, amount):
        self.health = min(self.health + amount, self.max_health)
        return f"{self.name} heals for {amount}! Health: {self.health}/{self.max_health}"

# Warrior class - high health, melee attacks
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, health=120, attack_power=25)
        self.armor = 10

    def take_damage(self, damage):
        # Warriors have armor that reduces damage
        reduced_damage = max(1, damage - self.armor)
        return super().take_damage(reduced_damage)

    def shield_bash(self, target):
        damage = self.attack_power + 10
        target.take_damage(damage)
        return f"{self.name} shield bashes {target.name} for {damage} damage!"

# Mage class - low health, magic attacks
```

```python
class Mage(Character):
    def __init__(self, name):
        super().__init__(name, health=70, attack_power=35)
        self.mana = 100

    def fireball(self, target):
        if self.mana >= 20:
            self.mana -= 20
            damage = self.attack_power + 15
            target.take_damage(damage)
            return f"{self.name} casts Fireball on {target.name} for {damage} damage!"
        return f"{self.name} doesn't have enough mana!"

    def heal_spell(self, target):
        if self.mana >= 15:
            self.mana -= 15
            heal_amount = 30
            return target.heal(heal_amount)
        return f"{self.name} doesn't have enough mana!"

# Rogue class - medium health, high attack with critical hits
class Rogue(Character):
    def __init__(self, name):
        super().__init__(name, health=90, attack_power=30)
        self.stealth = False

    def sneak_attack(self, target):
        if self.stealth:
            damage = self.attack_power * 2
            self.stealth = False
            target.take_damage(damage)
            return f"{self.name} performs a sneak attack on {target.name} for {damage} damage!"
        return f"{self.name} needs to be stealthed to sneak attack!"

    def enter_stealth(self):
        self.stealth = True
        return f"{self.name} enters stealth mode!"

# Polymorphic battle function
def battle_simulation(characters):
    print("=== BATTLE SIMULATION ===")

    for i, char in enumerate(characters):
        print(f"{i+1}. {char.name} ({type(char).__name__}) - Health: {char.health}")
```

```
    print("\n=== BATTLE BEGINS ===")

    # Example battle sequence using polymorphism
    warrior = characters[0]
    mage = characters[1]
    rogue = characters[2]

    # Each character uses their unique abilities
    print(warrior.attack(mage))
    print(mage.fireball(warrior))
    print(rogue.enter_stealth())
    print(rogue.sneak_attack(mage))
    print(mage.heal_spell(mage))
    print(warrior.shield_bash(rogue))

# Create characters and run simulation
party = [
    Warrior("Thorgar"),
    Mage("Gandora"),
    Rogue("Shadow")
]

battle_simulation(party)
```

Part 8: Key Concepts Summary

Inheritance Types:

1. **Single Inheritance**: One parent → One child
2. **Multiple Inheritance**: Multiple parents → One child
3. **Multilevel Inheritance**: Chain of inheritance (grandparent → parent → child)

Important Methods:

- `super()`: Access parent class methods
- `__mro__`: See method resolution order
- `isinstance(obj, Class)`: Check if object is instance of class
- `issubclass(Child, Parent)`: Check if class inherits from another

Polymorphism Types:

1. **Method Overriding**: Child class redefines parent method
2. **Duck Typing**: If it has the right methods, it works!

Best Practices:

1. Use inheritance for "is-a" relationships (Dog IS an Animal)
2. Use composition for "has-a" relationships (Car HAS an Engine)
3. Keep inheritance hierarchies shallow (avoid deep chains)
4. Use abstract base classes for enforcing interfaces
5. Prefer composition over inheritance when possible

Practice Exercises

Exercise 1: Vehicle Hierarchy

Create a vehicle inheritance system:

- Base class: `Vehicle` (brand, year, start_engine)
- Child classes: `Car` (doors, trunk_size), `Motorcycle` (has_sidecar)
- Implement polymorphic behavior for `start_engine`

Exercise 2: Employee Management

Build an employee system with:

- Base class: `Employee` (name, salary, work)
- Child classes: `Manager` (team_size), `Developer` (programming_language), `Designer` (design_software)
- Use polymorphism for different `work()` behaviors

Exercise 3: Shape Calculator

Create a shape calculation system:

- Abstract base with `area()` and `perimeter()`
- Implement for Rectangle, Circle, Triangle
- Create a function that calculates total area of mixed shapes

Master these concepts and you'll understand 90% of object-oriented programming!