Catch ’Em All

OOP Backend Mastery

OOP

Encapsulation

Abstraction

Inheritance

Polymorphism

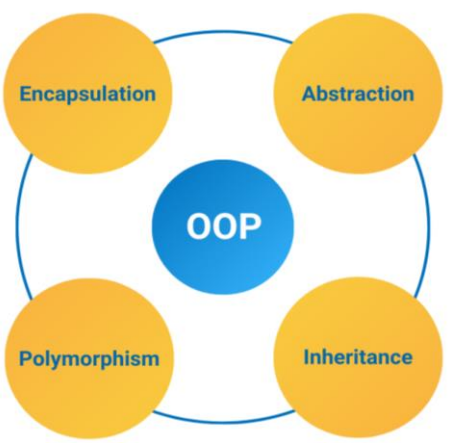
TETSAB

MASTERING OBJECT-ORIENTED PROGRAMMING

OOP with Pokémon

OOP is a way to organize code based on objects, just like Pokémon in the real world each has its own data (name, type, HP) and behaviors (attack, defend). This approach makes coding cleaner, easier to understand, and more reusable.

The four pillars of OOP are: **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism**.



01

Abstraction

Hiding Complexity, Showing Essentials

What is it?

Abstraction hides unnecessary details, showing only what’s important to use the object.

ABSTRACTION

Real example

When you play Pokémon, you don’t need to understand how the damage formula is calculated, how type effectiveness is determined, or how HP is subtracted in the background. You just select Attack, choose a move, and the game handles everything for you. The interface is simple, but behind it, there’s a lot of code working to make the battle happen.

Abstraction

```python
class Pokemon:
    def attack(self):
        print("Pokemon attacks!")

# Usage
pikachu = Pokemon()
pikachu.attack()

```

The player only calls attack(). The complex internal logic stays hidden.

02

Encapsulation

Protecting and Controlling Access

What is it?

Encapsulation keeps the Pokémon’s data safe, preventing direct manipulation. For example, you can’t just change HP whenever you want, it has to follow game rules.

ENCAPSULATION

Real example

In the game, you can't directly change Pikachu’s HP to 9999 just by clicking. The only way to change HP is by taking damage, using potions, or visiting a Pokémon Center. This protects the game balance and rules. The same happens in code: HP should only change through specific methods like `take_damage()` or `heal()`.

Encapsulation

```python
class Pokemon:
    def __init__(self, name, hp):
        self.name = name
        self.__hp = hp  # Private attribute

    def take_damage(self, amount):
        self.__hp = max(0, self.__hp - amount)

        print(f"{self.name} took {amount} damage.")

    def heal(self, amount):
        self.__hp += amount
        print(f"{self.name} healed {amount} HP.")

    def get_hp(self):
        return self.__hp

# Usage
pikachu = Pokemon("Pikachu", 100)
pikachu.take_damage(30)
pikachu.heal(20)
print(pikachu.get_hp())  # 90

```

The HP is protected (`__hp`) and can’t be modified directly — only through the provided methods.

03

Inheritance

Reusing and Extending Behavior

What is it?

Inheritance lets you create specific Pokémon based on a general Pokémon class, reusing common features.

INHERITANCE

Real example

All Pokémon can attack, defend, and level up, but each has unique moves and abilities. Instead of rewriting the same attack method for each Pokémon, you create a general Pokemon class, then create specific Pokémon like Pikachu or Charmander that inherit those basic abilities while adding their own.

Inheritance

```python
class Pokemon:
    def attack(self):
        print("Pokemon attacks!")

class Pikachu(Pokemon):
    def special_attack(self):
        print("Pikachu uses Thunderbolt!")

# Usage
pikachu = Pikachu()
pikachu.attack()          # Inherited from Pokemon
pikachu.special_attack() # Unique to Pikachu

```

`Pikachu` inherits `attack()` from `Pokemon` but also has its unique move `special_attack()`

04

Polymorphism

Same Method, Different Behaviors

What is it?

Polymorphism allows different Pokémon to use the same method name like `attack()`, but with their own specific behaviors.

POLYMORPHYSM

Real example

In battles, every Pokémon has an Attack option. However, the move isn’t the same — Pikachu uses Thunderbolt, while Charmander uses Flamethrower. From the player’s view, it’s the same action (Attack), but the effect depends on which Pokémon is used.

Polymorphysm

```
class Pikachu:
    def attack(self):
        print("Pikachu uses Thunderbolt!")

class Charmander:
    def attack(self):
        print("Charmander uses Flamethrower!")


# Usage
team = [Pikachu(), Charmander()]
for pokemon in team:
    pokemon.attack()

```

Both Pokémon respond to `attack()`, but the behavior depends on the Pokémon.

CONCLUSION

OOP makes coding more intuitive by mirroring how we think about the real world, in this case, a world full of Pokémon! Each Pokémon is an object with attributes (like name, type, HP) and behaviors (like attack, heal). Mastering **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism** helps you build better, cleaner, and smarter code, just like building the perfect Pokémon team.

Thank you for reading this far!

This E-book was generated by AI and formatted by a human.

The step-by-step process can be found on GitHub.

https://github.com/tetsab/oop-ebook

This content was created for educational and learning purposes. It has not undergone thorough human validation and may contain errors generated by AI.