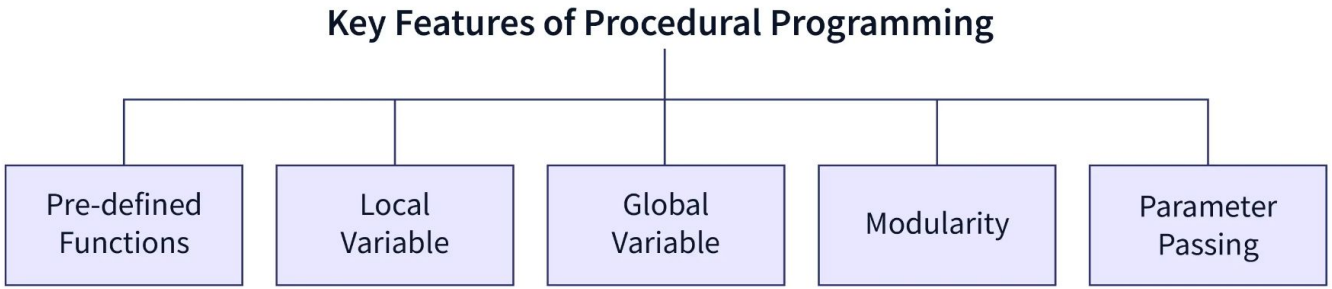
# Object Oriented Programming

## Motivation

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# What happened so far? Procedural Programming



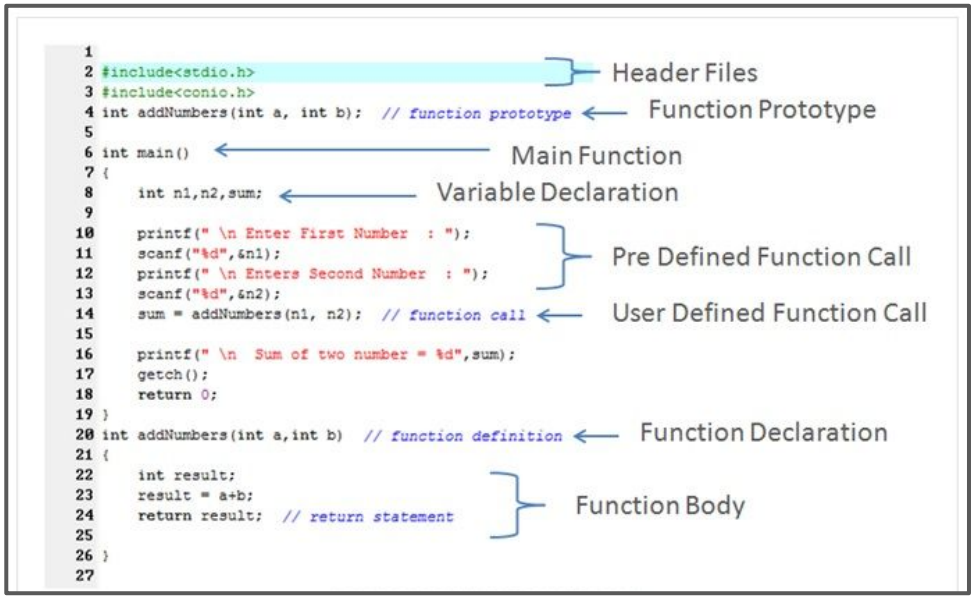

SCALER Topics

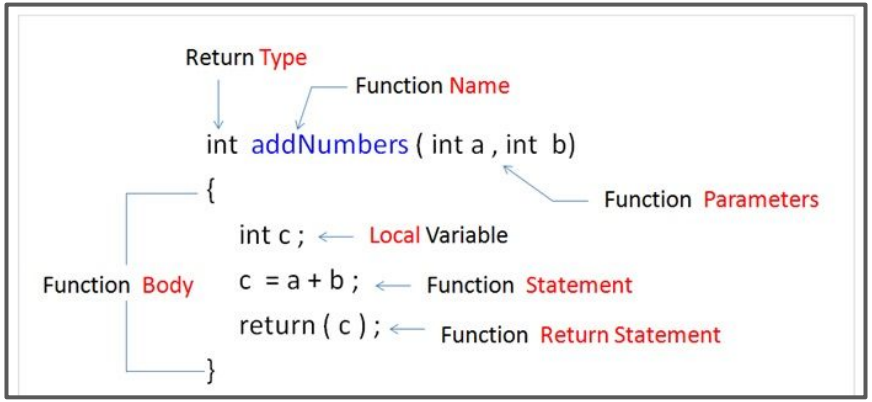
# Procedural Programming: Structure

```
1
2  #include<stdio.h>                                   } Header Files
3  #include<conio.h>
4  int addNumbers(int a, int b);  // function prototype  <- Function Prototype
5
6  int main()                                          <- Main Function
7  {
8      int n1,n2,sum;                                  <- Variable Declaration
9
10     printf(" \n Enter First Number  : ");           } Pre Defined Function Call
11     scanf("%d",&n1);
12     printf(" \n Enters Second Number  : ");
13     scanf("%d",&n2);
14     sum = addNumbers(n1, n2);  // function call     <- User Defined Function Call
15
16     printf(" \n  Sum of two number = %d",sum);
17     getch();
18     return 0;
19 }
20 int addNumbers(int a,int b)  // function definition  <- Function Declaration
21 {
22     int result;                                     } Function Body
23     result = a+b;
24     return result;  // return statement
25
26 }
27
```

```
#include <stdio.h>                                   <- Function Parameters
int addNumbers(int a, int b);                        <- Function Prototype
int main()                                           <- Main Function
{
    ... .. ...
    sum = addNumbers(n1, n2);                        <- Function Call Statement
    ... .. ...
}
int addNumbers(int a, int b)                         <- Function Declaration
{
   ... .. ...
   ... .. ...
   return result                                     <- Function Arguments
}
```

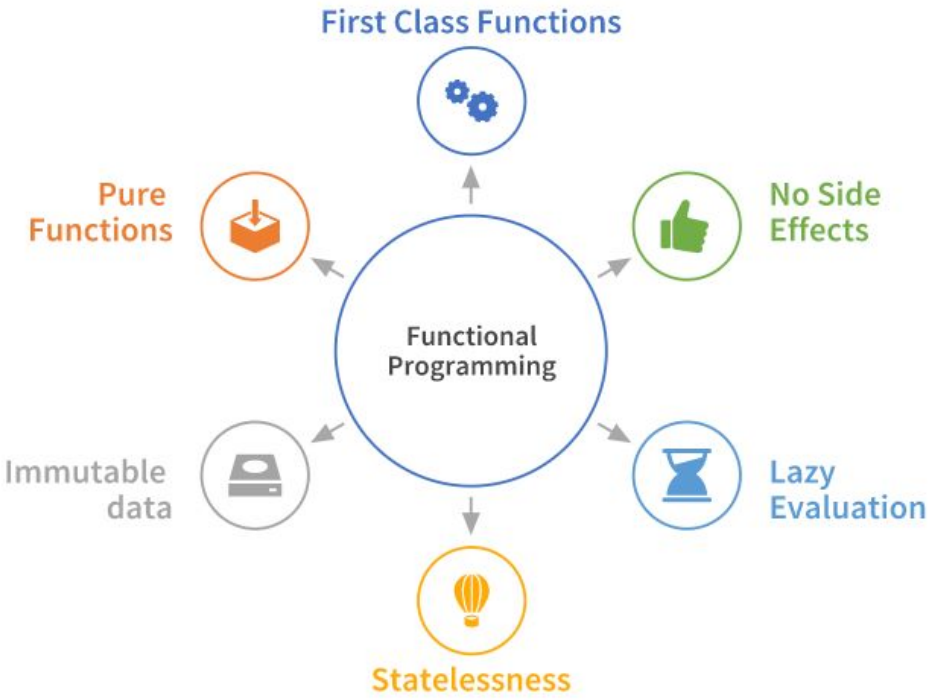
# Procedural Programming: Structure



# So then what's wrong with Procedural?

# Design Principles of Functional Programming



# Functional Programming: First-class Functions

```
#include <iostream>
#include <functional>

// A simple function that takes another function as a parameter

void executeFunction(const std::function<void(int)>& func, int value) {
    func(value); // Call the passed function with the given value
}

int main() {
    // Define a lambda function
    auto printSquare = [](int x) {
        std::cout << "The square of " << x << " is " << (x * x) << std::endl;
    };

    // Pass the lambda function to another function
    executeFunction(printSquare, 5);

    return 0;
}
```

**Capture clause**: no variable captured from the scope

First-class function

# Achieving SoC via Pure & First-class Functions

```cpp
#include <iostream>
#include <string>

void processUserInput() {
    std::string userInput;
    std::cout << "Enter a number: ";
    std::cin >> userInput;

    // Validate the input
    for (char c : userInput) {
        if (!isdigit(c)) {
            std::cout << "Invalid input!" <<
std::endl;
            return;
        }
    }

    // Convert the input and calculate the
square
    int number = std::stoi(userInput);
    std::cout << "The square is: " << (number *
number) << std::endl;
}

int main() {
    processUserInput();
    return 0;
}
```

Procedural Style

```cpp
#include <iostream>
#include <string>
#include <optional>
#include <functional>

// Pure function to get user input std::string
getInput() {
std::cout << "Enter a number: ";
std::string input;
std::cin >> input;
return input;
}

// Pure function to validate input
std::optional<int> validateAndConvert(const
std::string& input) {
for (char c : input) {
if (!isdigit(c)) {
return std::nullopt;
  // Return an empty optional for invalid input
 }
}
return std::stoi(input);
// Return the converted number if valid
}

// Pure function to process
(e.g., calculate the square)
int calculateSquare(int number) {
return number * number;
}
```

```cpp
// Functional composition: Combine pure functions
void processInput(const std::function<void(const
std::string&)>& successHandler, const
std::function<void()>&errorHandler) {
auto input = getInput();
auto validatedInput =validateAndConvert(input);
if (validatedInput) {
successHandler(input);
// Call success handler with valid input
} else {
errorHandler();
// Call error handler
 }
}

int main() {
processInput([](const std::string& input){
int number = std::stoi(input);
int square = calculateSquare(number);
std::cout << "The square is: " << square << std::endl; },
[]() {
 std::cout << "Invalid input!" << std::endl; }
);
return 0;
}
```

# Achieving Behavior Abstraction via Higher-order Functions

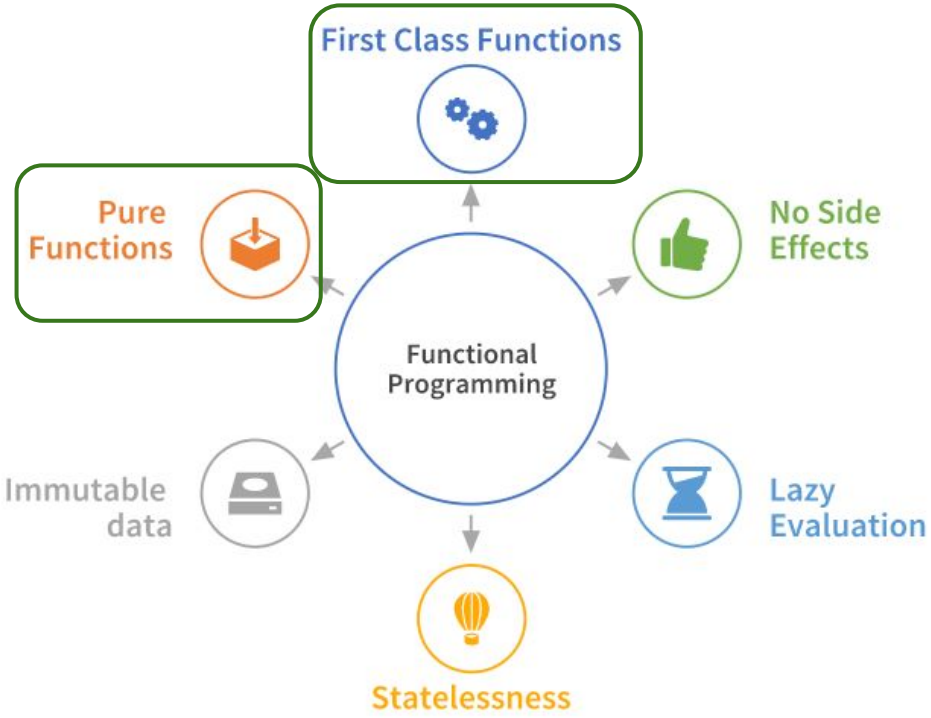
```
#include <iostream>
#include <vector>
#include <functional>

// Higher-order function for processing numbers
void processNumbers(const std::vector<int>& numbers,
                    const std::function<int(int)>&operation) {
    for (int num : numbers) {
        std::cout << operation(num) << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Pass different operations to processNumbers
    processNumbers(numbers, [](int x) { return x * x; });  // Print squares
    processNumbers(numbers, [](int x) { return x * x * x; });  // Print cubes
    processNumbers(numbers, [](int x) { return x + 10; });  // Add 10 to each

    return 0;
}
```

# Achieving Dynamic Behavior via Functional Map

```
#include <iostream>
#include <functional>
#include <map>

// Functional map of operations
std::map<char, std::function<int(int, int)>> getOperations()
{
    return {
      {'+', [](int a, int b) { return a + b; }},
      {'-', [](int a, int b) { return a - b; }},
      {'*', [](int a, int b) { return a * b; }},
      {'/', [](int a, int b) { return b != 0 ? a / b : 0; }}
        // Handle division by zero
   };
}

void executeOperation(char operation, int a, int b,
const std::map<char, std::function<int(int, int)>>&
operations) {
    if (operations.find(operation) != operations.end()) {
        std::cout << "Result: " <<
        operations.at(operation)(a, b) << std::endl;
    }
```

```
else {
        std::cout << "Unknown operation!" <<
std::endl;
    }
}

int main() {
    auto operations = getOperations();

    char operation;
    int a, b;

    std::cout << "Enter operation (+, -, *, /): ";
    std::cin >> operation;
    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    executeOperation(operation, a, b, operations);

    return 0;
}
```

# Design Principles of Functional Programming



# Functional Programming: Immutability vs. Statelessness

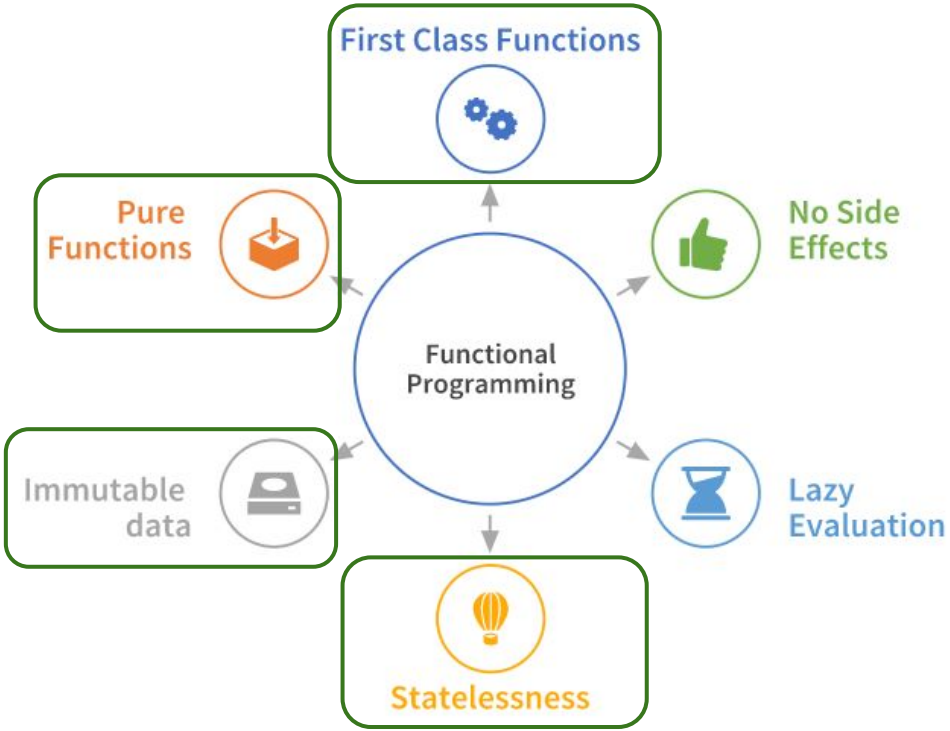
```
int globalCounter = 0;  // Mutable global state

int increment(int value) {
    return value + 1;
// Stateless function (no side effects, depends only on input)
}

// But the global state makes the system stateful:
globalCounter = increment(globalCounter);
```

```
const int immutableCounter = 5;

int incrementAndPrint() {
    std::cout << immutableCounter << std::endl;
// Has a side effect (stateful)
    return immutableCounter + 1;
}
```

# Functional Programming: Combining both

```
#include <iostream>
#include <vector>
#include <algorithm>

// Stateless function operating on immutable data
std::vector<int> doubleNumbers(const std::vector<int>& numbers) {
    std::vector<int> result = numbers; // Copy input to preserve immutability
    std::transform(result.begin(), result.end(), result.begin()[,](int x) { return x * 2; });
    return result;  // Return a new vector without modifying the original
}

int main() {
    const std::vector<int> numbers = {1, 2, 3, 4, 5}; // Immutable input
    std::vector<int> doubled = doubleNumbers(numbers);// Stateless function
    for (int num : doubled) {
        std::cout << num << " ";
    }
    return 0;
}
```

# Design Principles of Functional Programming

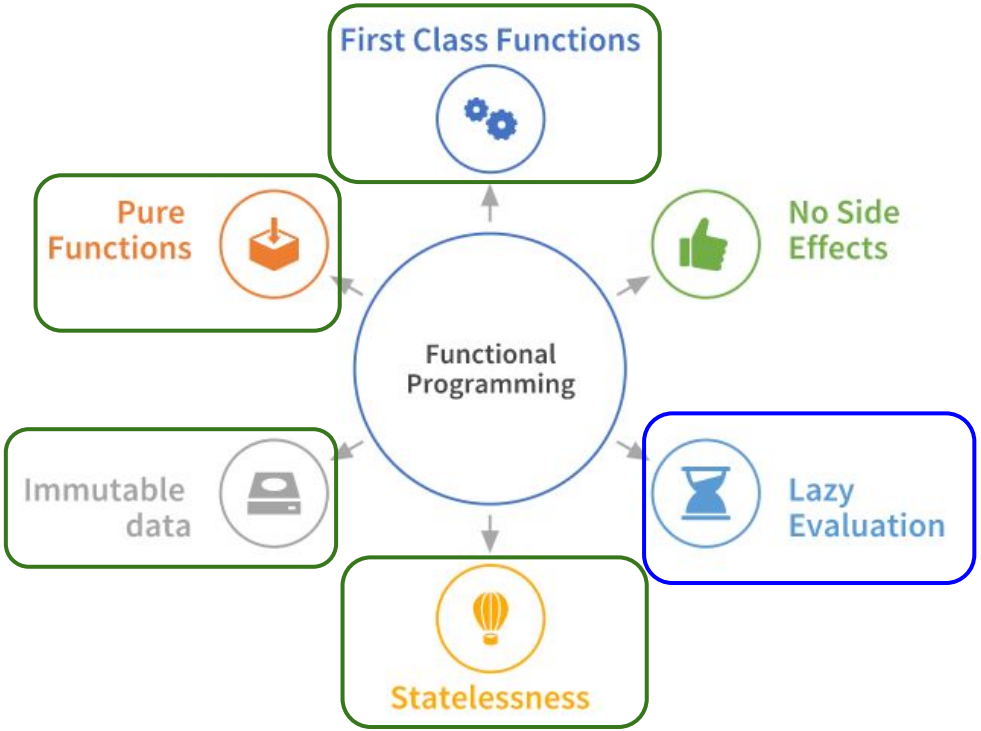


# Lazy Evaluation via Generators

```cpp
#include <iostream>
#include <functional>

#State-less Sequence Generator
class LazySequence {
public:
    LazySequence(int start, int step) :current(start), step(step) {
}
// member variables directly initialized before the body of the constructor is executed

    int next() {
        int value = current; // Capture the current value
        current += step; // Increment for the next value
        return value;
        // Generates the next value only when needed
    }

private:
    int current;
    // Tracks the current value in the sequence
    int step; // The step size for the sequence
};
```

```cpp
int main() {
    LazySequence sequence(0, 2);
    // Start at 0, increment by 2

    // Generate and print the first 5 elements lazily
    for (int i = 0; i < 5; ++i) {
        std::cout << sequence.next() << " ";
        // Outputs: 0 2 4 6 8
    }

    return 0;
}
```

# Design Principles of Functional Programming



# The final bit - Side-effects

```
int addAndPrint(int a, int b) {
    int result = a + b;
    std::cout << "The result is: " << result << std::endl;  // Side effect: I/O
    return result;
}
```

```
#include <fstream>

void writeFile(const std::string& filename, const std::string& content) {
    std::ofstream file(filename);
    file << content;  // Side-effect: Writes data to a file (external system interaction)
}

int main() {
    writeFile("example.txt", "Hello, file!");
    return 0;
}
```

# Removing Side-effects

```
void printResult(int result) {
    std::cout << "Result: " << result << std::endl; // Isolated side-effect
}

int add(int a, int b) {
    return a + b; // Pure function
}

int main() {
    int result = add(3, 4);
    printResult(result); // Side effect is isolated here
    return 0;
}
```

# Functional Programming: Key takeaways

1. **Dynamic Behavior**: Functions can be dynamically *assigned or modified at runtime*

2. **Separation of Concerns (SoC)**: *Decouple logic* (what to do) from implementation details (how to do it)

3. **Encapsulation of Behavior**: Functions *encapsulate behavior*, allowing them to be treated as reusable, interchangeable logic units.

4. **Runtime Composition**: *Design complex behaviors or pipelines* (without duplicating code) by composing or chaining functions dynamically.

# Then why OOP?

We will see next ….

# Object Oriented Programming

## Why OOP?

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Why not stick to Functional Programming?

| Drawback of Procedural Programming | How FP Addresses It | How OOP Addresses It |
|---|---|---|
| Tight coupling between data and functions | • Immutability<br>• Pure functions | • Encapsulates data and behavior within objects |
| Global state and side effects | • Immutability<br>• Pure functions | • Restricted access via Encapsulation<br>• Does not inherently avoid side effects |
| Code duplication and lack of reusability | • Reusable pure functions<br>• Higher-order functions | • Inheritance<br>• Polymorphism |
| Poor modularity and maintainability | • Function composition & modularity | • Self-contained classes and objects |
| Difficulty in scaling large applications | Scales well (via functional decomposition) for:<br>• Mathematical problems<br>• Stateless systems | • Modularity<br>• Encapsulation<br>• Inheritance |

# Why not stick to Functional Programming?

| Drawback of Procedural Programming | How FP Addresses It | How OOP Addresses It |
|---|---|---|
| Difficulty in modeling real-world systems | *No good answer!* | • Real-world entities as objects and classes |
| State management challenges | • Immutability (i.e., avoids mutable state)<br>• State mgmt. simplified! | • Objects are encapsulated<br>• Object state managed locally, allowing easy updates |
| No clear abstraction boundaries | • Higher-order functions | • Encapsulation<br>• Inheritance<br>• Interfaces |
| Poor team collaboration for large projects | • Modular code<br>• May require expertise to maintain clarity in team settings | • Code as objects and classes<br>• Easy for code delegation |

What happens when systems are large?

*Scalability in Functional Programming*

# Scaling large applications: Statelessness (btw, Python begins!)

```
def calculator(operation, a, b):
    if operation == "add":
        return a + b
    elif operation == "subtract":
        return a - b
    elif operation == "multiply":
        return a * b
    elif operation == "divide":
        return a / b if b != 0 else "Error: Division by zero"
    else:
        return "Error: Unsupported operation"

# Example usage:
print(calculator("add", 2, 3))       # Output: 5
print(calculator("multiply", 4, 5)) # Output: 20; Independent of previous call
```

- **Independence**: Each interaction is standalone
- **No Memory**: The system does not retain data or "state" between requests
- **Scalability**: Easy to scale because there's no need to manage or synchronize state

# Scaling large applications: Statelessness via Function Composition

```
from functools import reduce

def apply_operations(initial value, operations):
    return reduce(lambda acc, op: op(acc),operations, initial_value)

# Define operations as functions
operations = [
    lambda x: add(x, 5),         # Add 5
    lambda x: multiply(x, 2),    # Multiply by 2
    lambda x: subtract(x, 3),    # Subtract 3
    lambda x: divide(x, 7),      # Divide by 7
]

result = apply_operations(0, operations)  # Start with 0
print(f"Final result: {result}") # Output: 1.0
```

# Can any system be modeled as stateless? Ex: Shopping Cart

1. Add items to the cart
2. Update item quantities
3. Remove items from the cart
4. Persist the cart across multiple interactions or page reloads
5. Retain the cart state until the order is placed or the session ends

# Why a shopping cart ~~cannot~~ should not be stateless

1. **Persistent State**:
   - Must *retain the list of items a user has added and their respective quantities*
   - *Cannot forget* the cart contents after each request (e.g., when the user navigates to a different page or refreshes the page)

1. **User-Specific Data**:
   - Each user's cart is unique → need to *track* the *items added by a specific user, either in memory or a database*
   - Stateless systems cannot tie specific data to a user without re-sending all cart data with every request, which is *inefficient*

1. **Session Continuity**:
   - Needs to *persist across a user's session*, potentially *spanning multiple requests or even days* (if they come back later)
   - Stateless systems cannot retain session continuity because they discard all information between requests

1. **Performance Concerns**:
   - In a stateless system, the client (browser) would need to send the entire cart data (e.g., all items, quantities) with every request to the server → *inefficient* for larger carts and increases *network overhead*

# What's the stateful version? Closure-styled Encapsulation

```
def create_calculator():
    state = 0
    # Encapsulated state

    def add(value):
        nonlocal state
        state += value
        return state

    def subtract(value):
        nonlocal state
        state -= value
        return state

    def multiply(value):
        nonlocal state
        state *= value
        return state

    def divide(value):
        nonlocal state
        if value == 0:
            return "Error: Division by zero"
        state /= value
        return state

    def reset():
        nonlocal state
        state = 0
        return state

    def get_state():
        return state

    return add, subtract, multiply, divide, reset, get_state

# Example Usage
add, subtract, multiply, divide, reset, get_state = create_calculator()

print(add(5))
# Output: 5
print(multiply(2))
# Output: 10
print(subtract(3))
# Output: 7
print(divide(7))
# Output: 1.0
print(get_state())
# Output: 1.0
print(reset())
# Output: 0
```

# Is Closure scalable?

**Scalability**: **Not ideal for large-scale or distributed systems.**

- Localized and encapsulate state *within a single function*
- Works well for small, isolated systems
- Does not scale well when:
  - System needs to handle a large number of concurrent users
  - State needs to be shared or synchronized across multiple instances
- Closures also become harder to maintain and debug as the functionality grows

**Best for**: Small-scale, self-contained applications where each instance handles independent users or sessions.

# What would be a scalable version? Immutable Data Structure

```
def add(state, value):
    return state + value #creates and returns a new state

def subtract(state, value):
    return state - value

def multiply(state, value):
    return state * value

def divide(state, value):
    if value == 0:
        return "Error: Division by zero", state
    return state / value

# Example Usage
state = 0  # Initial state

state = add(state, 5)       # Add 5; returns a new state
state = multiply(state, 2)  # Multiply by 2
state = subtract(state, 3)  # Subtract 3
state = divide(state, 7)    # Divide by 7

print(state)  # Output: 1.0
```

# Key Takeaways

| Version | Scalability | Best Use Case |
|---|---|---|
| Closure-Based Stateful Calculator | Limited | Small-scale, session-based systems where each instance operates independently. |
| Stateful Calculator Using Generators | Limited | Small-scale systems requiring efficient, localized state handling. |
| Stateful Calculator with Explicit State | Moderately Scalable | Stateless systems that require functional principles and immutability. |
| Stateful Calculator Using Immutable Data Structure | Highly Scalable | Large-scale, distributed systems, or applications requiring horizontal scaling. |

Then why OOP? Why not Immutable Data Structures?

# Modeling real-world entities can make life “unreal”

# Modeling a Car in FP via Pure Functions & Explicit State Passing

```
def create_car(color, engine_size):
    return {"color": color, "engine_size": engine_size, "speed": 0}

def accelerate(car, increment):
    # Create a new car speed-state with updated speed
    new_car = car.copy()
    new_car["speed"] += increment
    return new_car

def brake(car, decrement):
    # Create a new car speed-state with updated speed
    new_car = car.copy()
    new_car["speed"] = max(0, new_car["speed"] - decrement)
    return new_car

def display_details(car):
    print(f"Car Details:")
    print(f"  Color: {car['color']}")
    print(f"  Engine Size: {car['engine_size']}L")
    print(f"  Speed: {car['speed']} km/h")
```

Dictionary emulating an object

Trivia: Same as double quote - `car["color"]`

```
## Example Usage
my_car = create_car("Red", 2.0)
display_details(my_car)

my_car = accelerate(my_car, 50)
display_details(my_car)

my_car = brake(my_car, 20)
display_details(my_car)
```

# Modeling Car in OOP via Encapsulation of *internally* *shared* state

```
class Car:
    def __init__(self, color, engine_size):
        self.color = color  # Attribute: color of the car
        self.engine_size = engine_size # Attribute: engine size of the car
        self.speed = 0  # Attribute: current speed of the car

    def accelerate(self, increment):
        self.speed += increment
        print(f"The car accelerates by {increment} km/h. Current speed:
{self.speed} km/h.")

    def brake(self, decrement):
        self.speed = max(0, self.speed - decrement)
        print(f"The car slows down by {decrement} km/h. Current speed:
{self.speed} km/h.")

    def display_details(self):
        print(f"Car Details:\nColor: {self.color}\nEngine Size:
{self.engine_size}L\nCurrent Speed: {self.speed} km/h")
```

```
# Example Usage
def main:
    my_car = Car("Red", 2.0)
    my_car.display_details()
    my_car.accelerate(50)
    my_car.brake(20)
    my_car.display_details()

#optional but norm
if __name__ == "__main__":
    main()
```

# Modeling real-world entities can be an overhead

# FP in action while modeling a City via Immutable DS

```
from pyrsistent import pmap, pvector

# Immutable data structure representing the city map
city = pmap({
    "districts": pvector([
        pmap({"name": "District A", "population": 10000}),
        pmap({"name": "District B", "population": 20000})
    ])
})

# Update the population of "District B"
def update_population(city, district_name, new_population):
    # Locate the district to update
    districts = city["districts"]
    updated_districts = districts.transform(
        [i for i, distr in enumerate(districts) if
distr["name"] == district_name][0],
        lambda distr: distr.set("population", new_population)
   #Update population of a district; [0]: denotes 1st match
    )
    # Return a new city with updated districts
    return city.set("districts", updated_districts)
```

hierarchical **immutable data structure** representing a city

```
{
    "districts": [
        {"name": "District A", "population": 10000},
        {"name": "District B", "population": 20000}
    ]
}
```

```
print("Original City:", city)
print("Updated City:", new_city)
```

# What can go wrong?

**Performance Costs**:

- Copying and reconstructing DS → significant **CPU and memory overhead**

- Worse if DS is:
  a. Deeply nested
  b. Contains a large number of elements

**Garbage Collection**:

- The original, unchanged data structure is no longer needed and will be garbage collected, adding to runtime costs

# Modeling City in OOP via Encapsulation of *internally shared* state

```
class District:
    # Constructor to initialize a district
    def __init__(self, name, population):
        self.name = name
        self.population = population

    def update_population(self, new_population):
        self.population = new_population

    def display(self):
        return f"{self.name}: Population
{self.population}"


class City:
    # Constructor to initialize a city with districts
    def __init__(self, districts):
        # Districts is a list of District objects
        self.districts = districts

    def update_district_population(self, district_name,
new_population):
        # Find the district and update its population
        for district in self.districts:
            if district.name == district_name:
                district.update_population(new_population)
                break

    def display(self):
        # Display all districts
        print("City Districts:")
        for district in self.districts:
            print(district.display())
```

# Modeling City in OOP via Encapsulation of *internally shared* state

```
# Example Usage
district_a = District("District A", 10000)
district_b = District("District B", 20000)

# Initialize the city with the districts
city = City([district_a, district_b])

# Display city information
city.display()

# Update the population of "District B"
city.update_district_population("District B", 25000)

# Display updated city information
city.display()
```

# Key Takeaways

| Criterion | FP with Immutable Data Structure | OOP Design |
|---|---|---|
| **State Management** | Works well for isolated or distributed state | Best for **localized**, **complex**, or **interconnected** state |
| **Modeling Real-World Systems** | Feels unnatural for real-world systems | Naturally represents real-world entities. |
| **Performance** | Can incur overhead from immutability | More efficient for stateful, performance-critical tasks |
| **Scalability** | Scales well for distributed systems | Scales well for object-heavy systems with local state |
| **Collaboration** | Harder to collaborate in large systems due to lack of cohesion | Easier to divide work with encapsulated classes |

# Object Oriented Programming

## A Tale of Constructors

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

```java
public class Person {
    private String name;
    private int age;

    // Default constructor
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display person details
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
```

```java
    public static void main(String[] args) {
        // Using the default constructor
        Person person1 = new Person();
        person1.display();

        // Using the parameterized constructor
        Person person2 = new Person("Alice", 25);
        person2.display();
    }
}
```

Guess the language?

```cpp
class Person {
private:
    string name;
    int age;

public:
    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
    }

    // Parameterized constructor
    Person(string personName, int personAge) {
        name = personName;
        age = personAge;
    }

    // Method to display person details
    void display() const {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    // Using the default constructor
    Person person1;
    person1.display();

    // Using the parameterized constructor
    Person person2("Alice", 25);
    person2.display();

    return 0;
}
```

Guess the language?

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
person = Person("Alice", 25)  # Constructor initializes name and age
```

Guess the language?

# Why Constructors? OOP begins at home!

| Purpose | Description |
|---|---|
| **Object Initialization** | Sets up the *initial state* of an object by initializing its attributes |
| **Encapsulation** | Hides *initialization details*, ensuring proper setup and reducing errors |
| **Abstraction** | Simplifies object creation by abstracting the *setup logic* |
| **Code Reusability** | Enables consistent reuse of *initialization logic* for multiple objects |

# Why Constructors? OOP begins at home!

| Purpose | Description |
|---|---|
| **Automatic Invocation** | Ensures the *constructor is called automatically* when an object is created |
| **Custom Behavior** | Allows *defining specific behaviors* (e.g., validations, derived properties) during object creation |
| **Overloading** | (In some languages) Provides flexibility by allowing *multiple ways to initialize* an object |
| **Improved Readability** | Enhances code clarity and usability by *standardizing* object creation and initialization |

# Are Constructors good enough?

# What’s wrong with this?

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# Client Code
def animal_sound(animal_type):
    if animal_type == "dog":
        animal = Dog()  # Tight coupling to the Dog class
    elif animal_type == "cat":
        animal = Cat()  # Tight coupling to the Cat class
    else:
        raise ValueError("Unknown animal type")

    return animal.speak()

# Usage
print(animal_sound("dog"))  # Output: Woof!
print(animal_sound("cat"))  # Output: Me
```

# Explicit Constructor is not needed if no setup needed

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# Client Code
def animal_sound(animal_type):
    if animal_type == "dog":
        animal = Dog()  # Tight coupling to the Dog class
    elif animal_type == "cat":
        animal = Cat()  # Tight coupling to the Cat class
    else:
        raise ValueError("Unknown animal type")

    return animal.speak()

# Usage
print(animal_sound("dog"))  # Output: Woof!
print(animal_sound("cat"))  # Output: Me
```

```python
class Animal:
    def __init__(self):
        print("Animal constructor called")

class Dog(Animal):
    def __init__(self):
        super().__init__()  # Call the Animal constructor
        print("Dog constructor called")

class Cat(Animal):
    def __init__(self):
        super().__init__()  # Call the Animal constructor
        print("Cat constructor called")
```

**Optional!**

# But there's something genuinely problematic!

```python
# Client Code
def animal_sound(animal_type):
    if animal_type == "dog":
        animal = Dog()  # Tight coupling to the Dog class
    elif animal_type == "cat":
        animal = Cat()  # Tight coupling to the Cat class
    else:
        raise ValueError("Unknown animal type")

    return animal.speak()
```

*Adding* a *new* animal *type* requires modifying `animal_sound`, violating the **Open-Closed Principle** (OCP)

The function is limited to the *specific* classes (Dog and Cat) and cannot handle new types *dynamically*

Mocking or *replacing* Dog or Cat in tests is difficult because they are hardcoded

**Tight Coupling!**

# Decoupling Constructors

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"
```

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

# Decoupling Constructors

```python
# Client Code
def animal_sound(animal_type):
    if animal_type == "dog":
        animal = Dog()  # Tight coupling t
    elif animal_type == "cat":
        animal = Cat()  # Tight coupling t
    else:
        raise ValueError("Unknown animal t

    return animal.speak()
```

```python
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError("Unknown animal type")
```

Adding a new animal type (e.g., Bird) requires only modifying the AnimalFactory without changing animal_sound.

No longer directly depends on the Dog or Cat classes. It delegates object creation to the AnimalFactory.

# Decoupling Constructors via Factory Design Pattern

```python
# Client Code
def animal_sound(animal_type):
    if animal_type == "dog":
        animal = Dog()  # Tight coupling to the Dog
    elif animal_type == "cat":
        animal = Cat()  # Tight coupling to the Cat
    else:
        raise ValueError("Unknown animal type")

    return animal.speak()
```

```python
# Client Code
def animal_sound(animal_type):
    animal = AnimalFactory.create_animal(animal_type)
    return animal.speak()
```

```python
# Usage
print(animal_sound("dog"))  # Output: Woof!
print(animal_sound("cat"))  # Output: Meow!
```

No longer directly depends on the Dog or Cat classes. It delegates object creation to the AnimalFactory.

# Achieving Open-Closed Principle (OCP)

```python
class Bird(Animal):
    def speak(self):
        return "Tweet!"

# Update Factory
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        elif animal_type == "bird":
            return Bird()
        else:
            raise ValueError("Unknown animal type")
```

**Open for Extension:**

- Add new functionality to the module (e.g., new features or behaviors)

**Closed for Modification:**

- The existing code of the module should not be modified when adding new functionality

# A bit about Static Methods …

```python
class Bird(Animal):
    def speak(self):
        return "Tweet!"

# Update Factory
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        elif animal_type == "bird":
            return Bird()
        else:
            raise ValueError("Unknown animal type")
```

**Utility Functions:**

- Need a helper function related to the class but *not dependent on instance or class-level data*

**Encapsulation:**

- To group related logic within the class for better readability and organization

**Factory Methods:**

- To create objects or perform operations that *don't need access to instance or class-specific data*

**Decorator**

# A bit about Static Methods …

```python
class MathUtils:
    def add(a, b):  # Without self, calling this method will raise an error
        return a + b

# Usage
# MathUtils.add(3, 5)  # Raises TypeError
```

```python
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

# Usage
result = MathUtils.add(3, 5)  # No instance required
print(result)  # Output: 8
```

**Utility Functions:**

- Need a helper function related to the class but *not dependent on instance or class-level data*

**Encapsulation:**

- To group related logic within the class for better readability and organization

**Factory Methods:**

- To create objects or perform operations that *don't need access to instance or class-specific data*

**Decorator**

# Meet its cousins - self and @classmethod

```
class Example:
    class_variable = "Class-Level Data"

    def __init__(self, value):
        self.value = value  # Instance attribute

    # Instance method using self
    def modify_instance_variable(self, new_value):
        self.value = new_value
        print(f"Instance variable modified to: {self.value}")
```

```
    # Class method using cls
    @classmethod
    def modify_class_variable(cls, new_value):
        cls.class_variable = new_value
        print(f"Class variable modified to: {cls.class_variable}")

    # Static method using no self or cls
    @staticmethod
    def utility_method(param):
        print(f"Static method called with param: {param}")
```

```
obj = Example(10)
obj.modify_instance_variable(20)  # Modifies obj's instance variable
```

```
Example.modify_class_variable("Modified by classmethod")
```

```
Example.utility_method("Static call")
```

# Animal is inherited by Dog and Cat as “*Base Class*”

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```python
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError("Unknown animal type")
```

# Why the base class *Animal*?

# Object Oriented Programming

## Principle of LSP

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Why the “Base (Abstract) Class” Animal?

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```python
animal = Animal()
```

```python
animal.speak()
```

Adheres to design principles like the **Open-Closed Principle (OCP)** and ***Liskov Substitution Principle (LSP)***

1. **Common interface (contract)** (`speak`) that ensures consistency
2. Enable **polymorphism**, allowing the client code to work with animals **generically**
3. Improve **extensibility**, making it easy to **add new** animals
4. Maintain a **clear and organized structure**

# What if we remove Base Class?

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def sound(self):  # Wrong method name
        return "Meow!"

# Client Code
def animal_sound(animal):
    return animal.speak()  # Runtime error for Cat
```

Removing the `Animal` class would lead to a less maintainable, error-prone, and less scalable system

# A *strictly* Abstract Class in Python

```python
from abc import ABC, abstractmethod

# Abstract Animal class
class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass  # Abstract method that must be implemented by subclasses
```

```python
animal = Animal()
```

# Liskov Substitution Principle (LSP)

"***Objects of a superclass should be replaceable with objects of its subclasses without altering the correctness of the program***"

- Any instance of a subclass should be able to replace an instance of its superclass without breaking the behavior expected by the client code

- Subclasses must fully honor the contract (interface and behavior) defined by the parent class

# LSP in Action

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
```

```python
def animal_sound(animal_type):
    animal = AnimalFactory.create_animal(animal_type)
    return animal.speak()
```

- The client code (`animal_sound`) interacts with objects through the `Animal` type.
- Doesn't need to know whether it's working with a `Dog`, `Cat`, or any other subclass of `Animal`.
- As long as each subclass of `Animal` properly implements the `speak` method, the client code will work correctly

# LSP in Action: Substitution w/o Breaking Behavior

```
animal = Dog()  # Substitute a Dog for Animal
print(animal.speak())  # Output: Woof!

animal = Cat()  # Substitute a Cat for Animal
print(animal.speak())  # Output: Meow!
```

- A `Dog` or `Cat` object can replace an `Animal` object in the `animal_sound` function *without requiring changes to the function's logic or causing errors*

**Consistency of the Interface (`speak`):**

- The `Animal` class defines the `speak` method as a contract.
- Every subclass (`Dog`, `Cat`) must implement `speak` with behavior that aligns with the expectations of the client code.

# LSP in Action: Extensibility w/o Violation

```python
class Bird(Animal):
    def speak(self):
        return "Tweet!"

# Usage:
bird = Bird()
print(bird.speak())  # Output: Tweet!
```

Adding a new animal (e.g., `Bird`) still adheres to the LSP

# What violates LSP?

# Breaking LSP: Violation of Expectation

```
class Dog(Animal):
    def speak(self):
        return 42  # Violates the expected return type (string)

# Client Code:
print(animal_sound("dog"))  # Breaks expectations, returns 42 instead of "Woof!"
```

# Breaking LSP: Unexpected Modifications

```
class Cat(Animal):
    def speak(self, volume):  # Incompatible method signature
        return "Meow!" * volume
```

# Breaking LSP: Unexpected Exceptions

```
class Bird(Animal):
    def speak(self):
        raise NotImplementedError("Birds don't speak")
```

# LSP

- **Ensures Polymorphism Works Correctly:**
  - The client code (`animal_sound`) can interact with any `Animal` object without needing to know the exact subclass.
  - Simplifies the design and allows the system to be extended with new animals seamlessly.
- **Prevents Fragile Code:**
  - If a subclass violates LSP, client code might break unexpectedly when interacting with that subclass, leading to fragile and hard-to-maintain code.
- **Supports Open-Closed Principle:**
  - LSP is closely tied to the **Open-Closed Principle (OCP)**.
  - By ensuring that subclasses respect the contract of the base class, you can add new functionality (new subclasses) without modifying existing client code.

# Can a code obey OCP but violate LSP?

# Breaking LSP w/o breaking OCP

```python
# Base Class
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

# Subclass 1: Dog
class Dog(Animal):
    def speak(self):
        return "Woof!"
```

```python
# Subclass 2: Cat
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Subclass 3: Fish (violates LSP)
class Fish(Animal):
    def speak(self):  # Fish can't "speak" in the traditional sense
        raise ValueError("Fish cannot speak!")
```

# Breaking LSP w/o breaking OCP

```python
# Factory to create animals
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        elif animal_type == "fish":
            return Fish()
        else:
            raise ValueError("Unknown animal type")
```

```python
# Client code
def animal_sound(animal_type):
    animal = AnimalFactory.create_animal(animal_type)  # Factory ensures OCP compliance
    return animal.speak()

# Usage
print(animal_sound("dog"))  # Output: Woof!
print(animal_sound("cat"))  # Output: Meow!
print(animal_sound("fish"))  # Raises ValueError: Fish cannot speak!
```

# Breaking LSP w/o breaking OCP (no Factory Pattern)

```
def animal_sound(animal: Animal):
    return animal.speak()

# Following OCP: New animals can be added without modifying `animal_sound`
print(animal_sound(Dog()))  # Output: Woof!
print(animal_sound(Cat()))  # Output: Meow!
print(animal_sound(Fish()))  # Raises ValueError: Fish cannot speak!
```

# Factory Update with OCP

```python
# Factory to create individual animals
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return
        else:
            raise
```

```python
class AnimalFactory:
    _animal_map = {
        "dog": Dog,
        "cat": Cat,
    }

    @staticmethod
    def register_animal(animal_type, animal_class):
        AnimalFactory._animal_map[animal_type] = animal_class


imal_map:
p[animal_type]()

al type: {animal_type}")
```

```python
class Bird(Animal):
    def speak(self):
        return "Tweet!"

# Registering Bird without modifying the factory's core logic
AnimalFactory.register_animal("bird", Bird)
```

# Object Oriented Programming

## Creational Design Patterns: Abstract Factory

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Relook at Factory Pattern

```python
# Base class
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

```python
# Factory to create individual animals
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError("Unknown
```

```python
# Usage
dog = AnimalFactory.create_animal("dog")
print(dog.speak())  # Output: Woof!

cat = AnimalFactory.create_animal("cat")
print(cat.speak())  # Output: Meow!
```

## What's missing??

- Only create *one* type of object (e.g., a *single* hierarchy of animals)
- Doesn't enforce consistency between *related* objects (e.g., "domestic" or "wild" versions of both `Dog` and `Cat`)

# Abstract Factory Design Pattern

```python
# Factory to create individual animals
class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError("Unknow
```

```python
# Abstract Factory
class AnimalFactory:
    def create_dog(self):
        raise NotImplementedError("Subclasses must implement this method")

    def create_cat(self):
        raise NotImplementedError("Subclasses must implement this method")
```

# Creating Concrete Factories out of Abstract Factory

```python
# Abstract Factory
class AnimalFactory:
    def create_dog(self):
        raise NotImplementedError("Subclasses must implement this method")

    def create_cat(self):
        raise NotImplementedError("Subclasses must implement this method")
```

```python
# Concrete Factory 1: Domestic Animals
class DomesticAnimalFactory(AnimalFactory):
    def create_dog(self):
        return DomesticDog()

    def create_cat(self):
        return DomesticCat()
```

```python
# Concrete Factory 2: Wild Animals
class WildAnimalFactory(AnimalFactory):
    def create_dog(self):
        return WildDog()

    def create_cat(self):
        return WildCat()
```

# Creating the Classes

```python
# Concrete Factory 1: Domestic Animals
class DomesticAnimalFactory(AnimalFactory):
    def create_dog(self):
        return DomesticDog()

    def create_cat(self):
        return DomesticCat()
```

```python
# Concrete Factory 2: Wild Animals
class WildAnimalFactory(AnimalFactory):
    def create_dog(self):
        return WildDog()

    def create_cat(self):
        return WildCat()
```

```python
# Base Classes
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")


class Dog(Animal):
    pass


class Cat(Animal):
    pass
```

```python
# Domestic Animals
class DomesticDog(Dog):
    def speak(self):
        return "Woof! (domestic)"


class DomesticCat(Cat):
    def speak(self):
        return "Meow! (domestic)"


# Wild Animals
class WildDog(Dog):
    def speak(self):
        return "Woof! (wild)"


class WildCat(Cat):
    def speak(self):
        return "Meow! (wild)"
```

# Creating Concrete Factories out of Abstract Factory

```
# Usage
dog = AnimalFactory.create_animal("dog")
print(dog.speak())  # Output: Woof!

cat = AnimalFactory.create_animal("cat")
print(cat.speak())  # Output: Meow!
```

```
# Usage
def animal_interaction(factory: AnimalFactory):
    dog = factory.create_dog()
    cat = factory.create_cat()
    print(dog.speak())
    print(cat.speak())


# Creating domestic animals
domestic_factory = DomesticAnimalFactory()
print("Domestic Animals:")
animal_interaction(domestic_factory)

# Creating wild animals
wild_factory = WildAnimalFactory()
print("\nWild Animals:")
animal_interaction(wild_factory)
```

# Creating Concrete Factories out of Abstract Factory

| Use Case | Factory Method | Abstract Factory |
|---|---|---|
| Single object creation | Works well | Overkill |
| *Families of related objects (eg. Domestic Animal Family)* | Cannot enforce consistency | Enforces consistency across object families |
| Need for multiple concrete factories | Harder to manage | Each concrete factory handles one family easily |
| Example | Create a `Dog` or `Cat` | Create a `DomesticDog` and `DomesticCat` pair |

# Object Oriented Programming

## Creational Design Patterns: Builder

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Creating Complex Objects: Building Step-by-step

```
class Animal:
    def __init__(self, species, sound, habitat, diet):
        self.species = species
        self.sound = sound
        self.habitat = habitat
        self.diet = diet

    def __str__(self):
        return (
            f"Animal: {self.species}, Sound: {self.sound}, "
            f"Habitat: {self.habitat}, Diet: {self.diet}"
        )
```

# Builder Design Pattern

```python
class AnimalBuilder:
    def __init__(self):
        # Default values
        self.species = None
        self.sound = None
        self.habitat = None
        self.diet = None

    def set_species(self, species):
        self.species = species
        return self

    def set_sound(self, sound):
        self.sound = sound
        return self

    def set_habitat(self, habitat):
        self.habitat = habitat
        return self

    def set_diet(self, diet):
        self.diet = diet
        return self
```

```python
def build(self):
    if not self.species or not self.sound:
        raise ValueError("Species and Sound are required!")
    return Animal(self.species, self.sound, self.habitat, self.diet)
```

**Builder Class** (`AnimalBuilder`):

- Handles the *step-by-step* construction of an `Animal` object.
- Provides methods (`set_species`, `set_sound`, etc.) to *configure* the attributes of the `Animal` object.
- Ensures defaults or constraints can be added easily (e.g., raising an error if `species` or `sound` is missing).

# Builder Design Pattern: A *detailed* Dog

```python
class AnimalBuilder:
    def __init__(self):
        # Default values
        self.species = None
        self.sound = None
        self.habitat = None
        self.diet = None

    def set_species(self, species):
        self.species = species
        return self

    def set_sound(self, sound):
        self.sound = sound
        return self

    def set_habitat(self, habitat):
        self.habitat = habitat
        return self

    def set_diet(self, diet):
        self.diet = diet
        return self
```

```python
def build(self):
    if not self.species or not self.sound:
        raise ValueError("Species and Sound are required!")
    return Animal(self.species, self.sound, self.habitat, self.diet)
```

```python
# Usage
def main():
    # Build a Dog
    dog = (
        AnimalBuilder()
        .set_species("Dog")
        .set_sound("Woof")
        .set_habitat("Domestic")
        .set_diet("Omnivore")
        .build()
    )
    print(dog)
```

```python
if __name__ == "__main__":
    main()
```

Chaining

# Builder Design Pattern: A *minimalist* Fish

```python
class AnimalBuilder:
    def __init__(self):
        # Default values
        self.species = None
        self.sound = None
        self.habitat = None
        self.diet = None

    def set_species(self, species):
        self.species = species
        return self

    def set_sound(self, sound):
        self.sound = sound
        return self

    def set_habitat(self, habitat):
        self.habitat = habitat
        return self

    def set_diet(self, diet):
        self.diet = diet
        return self
```

```python
def build(self):
    if not self.species or not self.sound:
        raise ValueError("Species and Sound are required!")
    return Animal(self.species, self.sound, self.habitat, self.diet)
```

```python
# Usage
def main():

    # Build a Fish with minimal attributes
    fish = (
        AnimalBuilder()
        .set_species("Fish")
        .set_sound("...")
        .set_habitat("Water")
        .build()
    )
    print(fish)
```

```python
if __name__ == "__main__":
    main()
```

# Why Builder Pattern?

| Advantage | Explanation | Example from Code |
|---|---|---|
| **Avoid Constructor Overload** | Simplifies object creation without requiring multiple constructors for different attribute combinations. | Instead of multiple `Animal` constructors (e.g., with/without habitat, diet, etc.), `AnimalBuilder` provides a step-by-step interface. |
| **Readable Object Creation** | Object construction is easier to read and understand with named methods for setting attributes. | `AnimalBuilder().set_species("Dog").set_sound("Woof").build()` is *more readable* than `Animal("Dog", "Woof", "Domestic", "Omnivore")`. |
| **Scalability** | New attributes can be added easily without modifying the existing class or constructor logic. | Adding a `set_lifespan()` method in `AnimalBuilder` doesn't affect existing `Animal` code. |

# Why Builder Pattern?

| | | |
|---|---|---|
| **Validation** | Ensures required fields are set and invalid configurations are avoided before object creation. | The `build()` method raises an error if `species` or `sound` is missing. |
| **Flexible Configuration** | Allows objects to be constructed with varying levels of detail, from minimal to fully detailed. | Example: A minimal `Fish` with only `species` and `sound`, or a full `Dog` with habitat and diet. |
| **Fluent Interface** | Enables chaining of method calls for concise and expressive code. | `AnimalBuilder().set_species("Dog").set_sound("Woof").build()` uses method chaining for clarity. |
| **Improved Maintainability** | Separates object creation logic from the class itself, making the code easier to manage and extend. | `AnimalBuilder` encapsulates the logic for setting attributes, keeping `Animal` clean and focused. |

# Object Oriented Programming

## Creational Design Patterns: Singleton

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Singleton Design Pattern : *Just one object!*

```python
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:  # Check if an instance already exists
            cls._instance = super(AnimalRegistry, cls).__new__(cls)
            # Create a new instance
            cls._instance.animals = {}
            # Initialize the dictionary for storing animals
        return cls._instance
        # Return the single instance
```

```python
def register_animal(self, species, details):
    """Registers an animal in the registry."""
    self.animals[species] = details

def get_animal_info(self, species):
    """Fetches details of a registered animal."""
    return self.animals.get(species, "Animal not found in registry.")

def get_all_animals(self):
    """Returns a dictionary of all registered animals."""
    return self.animals

def clear_registry(self):
    """Clears all registered animals."""
    self.animals = {}
```

Get method of dictionary `animals`

# Singleton Design Pattern : *Just one object!*

```python
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:  # Check if an instance already exists
            cls._instance = super(AnimalRegistry, cls).__new__(cls)
            # Create a new instance
            cls._instance.animals = {}
            # Initialize the dictionary for storing animals
        return cls._instance
        # Return the single instance
```

```python
# Usage
if __name__ == "__main__":
    registry1 = AnimalRegistry()
    registry1.register_animal("Dog", {"sound": "Woof", "habitat": "Domestic", "diet": "Omnivore"})
    registry1.register_animal("Cat", {"sound": "Meow", "habitat": "Domestic", "diet": "Carnivore"})

    # Fetch the same singleton instance
    registry2 = AnimalRegistry()

    # Checking that registry2 is the same as registry1
    print(registry1 is registry2)  # Output: True (Both point to the same instance)

    # Fetch animal details using registry2 (same as registry1)
    print(registry2.get_animal_info("Dog"))
    # Output: {'sound': 'Woof', 'habitat': 'Domestic', 'diet': 'Omnivore'}
    print(registry2.get_animal_info("Cat"))
    # Output: {'sound': 'Meow', 'habitat': 'Domestic', 'diet': 'Carnivore'}
    print(registry2.get_animal_info("Fish"))
    # Output: Animal not found in registry.
```

```
True  # Confirms that both registry1 and registry2 are the same instance
{'sound': 'Woof', 'habitat': 'Domestic', 'diet': 'Omnivore'}
{'sound': 'Meow', 'habitat': 'Domestic', 'diet': 'Carnivore'}
Animal not found in registry.
```

# Singleton Design Pattern : In C++

```cpp
#include <iostream>
#include <unordered_map>
#include <string>

class AnimalRegistry {
private:
    static AnimalRegistry* instance;  // Static instance (Singleton)
    std::unordered_map<std::string, std::string> animals; // Dictionary equivalent

    // Private constructor to prevent direct instantiation
    AnimalRegistry() {}

public:
    // Static method to get the Singleton instance
    static AnimalRegistry* getInstance() {
        if (instance == nullptr) {
            instance = new AnimalRegistry();
        }
        return instance;
    }
```

```cpp
animals["Dog"] = "Sound: Woof, Habitat: Domestic, Diet: Omnivore";
```

```cpp
    // Method to register an animal
    void registerAnimal(const std::string& species, const std::string& details) {
        animals[species] = details;
    }

    // Method to get animal information
    std::string getAnimalInfo(const std::string& species) {
        auto it = animals.find(species);
        if (it != animals.end()) {
            return it->second;
        }
        return "Animal not found in registry.";
    }
};

// Initialize static member
AnimalRegistry* AnimalRegistry::instance = nullptr;
```

Returns an *iterator* to the key-value pair

Don't want unnecessary copies! Read-only is needed here

Check if instance already exists

# Singleton Design Pattern : *In C++*

```cpp
#include <iostream>
#include <unordered_map>
#include <string>

class AnimalRegistry {
private:
    static AnimalRegistry* instance;  // Static instance (Singleton)
    std::unordered_map<std::string, std::string> animals; // Dictionary equivalent

    // Private constructor to prevent direct instantiation
    AnimalRegistry() {}

public:
    // Static method to get the Singleton instance
    static AnimalRegistry* getInstance() {
        if (instance == nullptr) {
            instance = new AnimalRegistry();
        }
        return instance;
    }
```

```cpp
int main() {
    // Get the Singleton instance
    AnimalRegistry* registry1 = AnimalRegistry::getInstance();

    // Register animals
    registry1->registerAnimal("Dog", "Sound: Woof, Habitat: Domestic, Diet: Omnivore");
    registry1->registerAnimal("Cat", "Sound: Meow, Habitat: Domestic, Diet: Carnivore");

    // Fetch the same Singleton instance
    AnimalRegistry* registry2 = AnimalRegistry::getInstance();

    // Check if both instances are the same
    std::cout << "Are registry1 and registry2 the same instance? "
              << (registry1 == registry2 ? "Yes" : "No") << std::endl;

    // Fetch animal details
    std::cout << registry2->getAnimalInfo("Dog") << std::endl;  // ✅ Found
    std::cout << registry2->getAnimalInfo("Cat") << std::endl;  // ✅ Found
    std::cout << registry2->getAnimalInfo("Fish") << std::endl; // ❌ Not Found

    return 0;
}
```

# Singleton Design Pattern: Post-mortem

```
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:  # Check if an insta
            cls._instance.animals = {}
            # Initialize the dictionary for storing an
        return cls._instance
        # Return the single instance
```

Calls `object.__n`

Why not `object.__new__(cls)`?

✅ **Future-proofing for inheritance**

- If `AnimalRegistry` *ever* **inherits from another class**, using `super()` ensures we call the *correct* `__new__` method from the *proper* superclass.

✅ **Maintains consistency**

- Using `super()` makes it clear that we're using `__new__` in the **context of inheritance**, even if `AnimalRegistry` *currently doesn't inherit from anything*.

✅ **Best Practice**

- `super()` ensures Python's method resolution order (MRO) is followed, making the Singleton more maintainable.

# Singleton Design Pattern: w/o `super()`

```python
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:
            cls._instance = object.__new__(cls)  # Using object.__new__ instead of super()
        return cls._instance

# Create instances
r1 = AnimalRegistry()
r2 = AnimalRegistry()

print(r1 is r2)  # ✅ Output: True (Singleton works)
```

Works!!

# Singleton Design Pattern: w/o `super()`

```python
class BaseRegistry:
    def __new__(cls):
        print("BaseRegistry __new__ called")
        return super().__new__(cls)

    def __init__(self):
        print("BaseRegistry initialized")

class AnimalRegistry(BaseRegistry):
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = object.__new__(cls)  # Using object.__new__()
        return cls._instance

# Create instances
r1 = AnimalRegistry()
r2 = AnimalRegistry()

# Check if r1 and r2 are the same instance (Singleton check)
print(r1 is r2)  # Expected Output: True
```

Works!!

✅ **`r1 is r2` prints `True`**, confirming that `AnimalRegistry` is a **Singleton**.

❌ **`BaseRegistry.__new__()` is not called at all** because `object.__new__(cls)` is used, ***bypassing inheritance***.

❌ **`BaseRegistry.__init__()` is called twice** because `__init__` runs every time an instance is returned.

```
BaseRegistry initialized
BaseRegistry initialized
True
```

# Singleton Design Pattern: Handling Multiple Inheritance

```
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:  # Check if an instance already exists
            cls._instance = super().__new__(cls)
            cls._instance.animals = {}
            # Initialize the dictionary for storing animals
        return cls._instance
        # Return the single instance
```

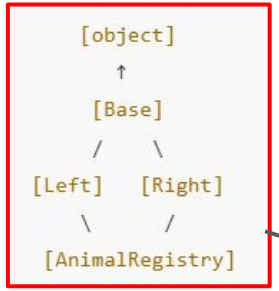
Why not `super().__new__(cls)`?

# Handling Multiple Inheritance using `super(AnimalRegistry,cls)`

```python
class Base:
    def log(self):
        print("Base log() called")

class Left(Base):
    def log(self):
        print("Left log() called")
        Base.log(self)  # Directly calling Base.log()

class Right(Base):
    def log(self):
        print("Right log() called")
        super().log()  # Calls next in MRO

class AnimalRegistry(Left, Right):
    def log(self):
        print("AnimalRegistry log() called")
        super().log()  # Calls next in MRO

registry = AnimalRegistry()
registry.log()
```

```
AnimalRegistry log() called
Left log() called
Right log() called
Base log() called
```

```
AnimalRegistry log() called
Left log() called
Base log() called  # ❌ Right.log() is skipped!
```




Use Python 3+ instead!!

`super() just works!`

**Method Resolution Order (MRO):**

*C3 Linearization: Left-right in same level*

*(the dreaded) Diamond*

# Handling Multiple Inheritance: Is C++ Any better?

```cpp
class Base {
public:
    void log() {
        std::cout << "Base log() called" << std::endl;
    }
};

class Left : public Base {
public:
    void log() {
        std::cout << "Left log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};

class Right : public Base {
public:
    void log() {
        std::cout << "Right log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};
```

```cpp
class AnimalRegistry : public Left, public Right {
public:
    void log() {
        std::cout << "AnimalRegistry log() called" << std::endl;
        Left::log();  // Explicitly calling Left::Log()
    }
};

int main() {
    AnimalRegistry registry;
    registry.log();  // Ambiguity issue!
    return 0;
}
```

```
AnimalRegistry log() called
Left log() called
Base log() called  ❌ (Right::log() is skipped!)
```

*Compiles and runs though!!*

*No super() !!*

```
[object]
   ↑
[Base]
  /   \
[Left]  [Right]
  \     /
[AnimalRegistry]
```

**Method Resolution Order (MRO):**

*C3 Linearization: Left-right in same level*

# Handling Multiple Inheritance: The Dreaded Diamond

```cpp
class Base {
public:
    int value;  // Shared variable
    void log() {
        std::cout << "Base log() called" << std::endl;
    }
};
```

```cpp
class Left : public Base {
public:
    void log() {
        std::cout << "Left log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};

class Right : public Base {
public:
    void log() {
        std::cout << "Right log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};
```

```cpp
class AnimalRegistry : public Left, public Right {
public:
    void log() {
        std::cout << "AnimalRegistry log() called" << std::endl;
        Left::log();  // Explicitly calling Left::log()
    }
};

int main() {
    AnimalRegistry registry;
    registry.log();  // Ambiguity issue!
    return 0;
}
```

```
error: request for member 'value' is ambiguous
```

```
    Base (Has `value`)
   /    \
Left    Right   (Each has its own copy of `Base`)
   \    /
 AnimalRegistry  (Now has two `Base` instances)
```

# Handling Multiple Inheritance: Solving the Dreaded Diamond

```cpp
class Base {
public:
    void log() {
        std::cout << "Base log() called" << std::endl;
    }
};

class Left : public Base {
public:
    void log() {
        std::cout << "Left log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};

class Right : public Base {
public:
    void log() {
        std::cout << "Right log() called" << std::endl;
        Base::log();  // Direct call to Base::log()
    }
};
```

```cpp
class AnimalRegistry : public Left, public Right {
public:
    void log() {
        std::cout << "AnimalRegistry log() called" << std::endl;
        Left::log();  // Explicitly calling Left::log()
    }
};

int main() {
    AnimalRegistry registry;
    registry.log();
    return 0;
}
```

```cpp
class Left : virtual public Base {};  // ✅ Virtual inheritance
class Right : virtual public Base {}; // ✅ Virtual inheritance
```

```
      Base (value)  ✅ (Only one shared instance!)
     /     \
  Left     Right   (Both refer to the same `Base` instance)
     \     /
   AnimalRegistry  (Contains only ONE `Base`!)
```

# Be careful with Singletons!

# Singleton Design Pattern : *Beware of global instances!*

```python
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.value = 0
        return cls._instance

# Usage
s1 = Singleton()
s1.value = 42

s2 = Singleton()
print(s2.value)  # Output: 42 (Modified globally!)
```

# Singleton Design Pattern : *Beware of silent dependencies!*

```python
class Logger:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new
        return cls._instance

class Service:
    def do_something(self):
        Logger().log("Action performed!")
      # Implicit Singleton dependency
```

`Logger` swappable

```python
class Service:
    def __init__(self, logger):
        self.logger = logger

    def do_something(self):
        self.logger.log("Action performed!")
      # Explicit dependency
```

Dependency Injection
(testing/alternatives easy!!)

# Singleton Design Pattern: *Breaking Single Responsibility Principle (SRP)*

```python
class Logger:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.logs = []
        return cls._instance

    def log(self, message):
        self.logs.append(message)  # Storing logs (Separate responsibility)
```

```python
class Logger:
    def log(self, message):
        print(message)

class LoggerSingleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = Logger()
        return cls._instance
```

# Singleton Design Pattern : *Beware of multi-threading!*

```python
class AnimalRegistry:
    _instance = None  # Stores the single instance

    def __new__(cls):
        if cls._instance is None:  # Check if an instance already exists
            cls._instance = super(AnimalR
            # Create a new instance
            cls._instance.animals = {}
            # Initialize the dictionary f
        return cls._instance
        # Return the single instance
```

```python
import threading

class AnimalRegistry:
    _instance = None
    _lock = threading.Lock()  # Lock for thread safety

    def __new__(cls):
        with cls._lock:  # Ensures thread safety
            if cls._instance is None:
                cls._instance = super(AnimalRegistry, cls).__new__(cls)
                cls._instance.animals = {}  # Dictionary to store registered animals
        return cls._instance
```

Prevent multiple threads from accessing *in parallel*

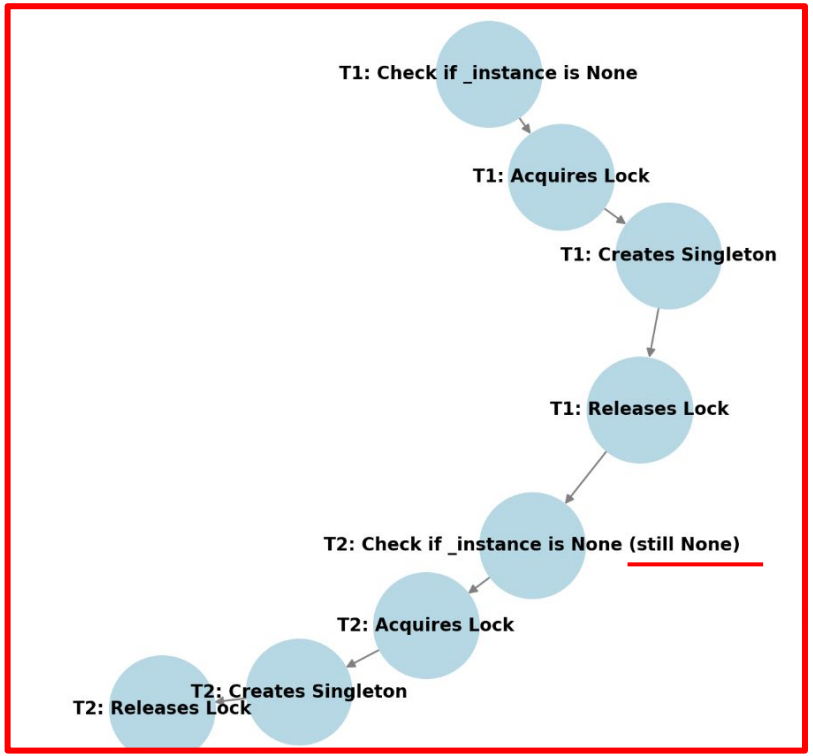
# Singleton Design Pattern : *Beware of concurrent threads!*

```python
import threading
import time

class Singleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is N
            time.sleep(0.01)
            # Artificial dela
            with cls._lock:
                if cls._instance is None:
                    print(f"Instance created b
                    cls._instance = super().__
        return cls._instance
```

```python
def create_instance():
    instance = Singleton()
    print(f"Instance ID: {id(instance)}")

# Start multiple threads
threads = [threading.Thread(target=create_instance) for _ in range(5)]
for t in threa
    t.start()
for t in threa
    t.join()
```

```
Instance created by Thread-1
Instance created by Thread-3  ❌ Another instance created!
Instance ID: 140250877951920
Instance ID: 140250877952240  ❌ Different instance!
Instance ID: 140250877951920
Instance ID: 140250877952240  ❌ Different instance!
Instance ID: 140250877951920
```

# Singleton Design Pattern : *Beware of concurrent threads!*




**Scenario: Two Threads ( `T1` and `T2` ) Running in Parallel**

| Step | Thread 1 (T1) | Thread 2 (T2) | Shared `_instance` |
|---|---|---|---|
| 1 | Checks `_instance is None` (✅ True) | Checks `_instance is None` (✅ True) | `_instance = None` |
| 2 | Acquires lock ✅ | *Waiting for lock* | `_instance = None` |
| 3 | Creates Singleton ( `_instance = New Object` ) | *Waiting for lock* | `_instance = 0x123ABC` |
| 4 | Releases lock ✅ | Acquires lock ✅ | `_instance = 0x123ABC` |
| 5 | *Finished Execution* | Checks `_instance is None` (❌ Still True, due to memory visibility issues) | `_instance = 0x123ABC` |
| 6 | *Finished Execution* | Creates Singleton ( `_instance = New Object` ) | `_instance = 0x456DEF` |
| 7 | *Finished Execution* | Releases lock ✅ | `_instance = 0x456DEF` |

# Singleton Design Pattern : *Beware of concurrent threads!*

```python
import threading
import time

class Singleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:  # 🚨 This ch
            time.sleep(0.01)
            # Artificial delay to force thread
            with cls._lock:
                if cls._instance is None:
                    print(f"Instance created by {threading.current_thread().name}")
                    cls._instance = super()
        return cls._instance
```

*Double-checking (performance booster)*

```python
import threading

class Singleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        with cls._lock:  # ✅ Lock before checking _instance
            if cls._instance is None:
                cls._instance = super().__new__(cls)
        return cls._instance
```

**If GIL then still ok, else Python can be tricky!**

# Singleton Design Pattern : *Beware of multi-threading!*

| Problem | Why is it Bad? |
|---|---|
| 1 **Introduces Global State** | Makes debugging harder because any part of the code can change the shared instance. |
| 2 **Hidden Dependencies (Bad Design)** | Code that depends on a Singleton does not explicitly state its dependency, making it harder to understand. |
| 3 **Breaks the Single Responsibility Principle (SRP)** | Singleton both **manages its instance** and **performs logic**, violating SRP. |
| 4 **Hard to Mock in Unit Tests** | Since the Singleton **forces a single instance**, replacing it with a mock object in tests is difficult. |
| 5 **Causes Thread-Safety Issues** | A poorly implemented Singleton can lead to **race conditions in multi-threaded environments**. |

# Object Oriented Programming

## Creational Design Patterns: Object Pool

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Object Pool Design Pattern : *Reusing without destroying*

```python
import time

class Animal:
    """Represents an animal in the zoo simulation."""

    def __init__(self, species):
        print(f"Creating a new {species}...")
        time.sleep(1)  # Simulate the time-consuming setup
        self.species = species
        self.active = False  # Indicates whether the animal is currently in use

    def interact(self, action):
        """Simulates an interaction with the animal."""
        self.active = True
        print(f"{self.species} is now {action}.")

    def release(self):
        """Releases the animal back to the pool."""
        self.active = False
        print(f"{self.species} is now resting in the pool.")
```

1. **Initialize a pool** with a fixed number of pre-created objects.
2. **Acquire an object** from the pool when needed.
3. **Use the object** as required.
4. ***Release the object*** *back into the pool for reuse*.

# Object Pool Design Pattern : *Reusing without destroying*

```python
class AnimalPool:
    """Manages a pool of reusable animals."""

    def __init__(self, species, size=3):
        """Creates a pool with a fixed number of animals."""
        self.pool = [Animal(species) for _ in range(size)]

    def acquire(self):
        """Gets an available animal from the pool. Creates a new one if none are available."""
        for animal in self.pool:
            if not animal.active:  # Find an inactive animal
                return animal
        new_animal = Animal(self.pool[0].species)  # Create a new one if the pool is empty
        self.pool.append(new_animal)
        return new_animal

    def release(self, animal):
        """Returns an animal to the pool for reuse."""
        animal.release()
```

**Trivia: Python's way of getting away w/o a variable**

# Object Pool Design Pattern : *Reusing without destroying*

***Every interaction is via the pool***

```
# Create an animal pool for Lions 🦁
lion_pool = AnimalPool("Lion", size=3)

# Acquire and interact with lions
lion1 = lion_pool.acquire()
lion1.interact("roaring")

lion2 = lion_pool.acquire()
lion2.interact("playing")

lion3 = lion_pool.acquire()
lion3.interact("eating")
```

```
# No available lions, so a new one is created
lion4 = lion_pool.acquire()
lion4.interact("resting")

# Release lions back to the pool
lion_pool.release(lion1)
lion_pool.release(lion2)

# Reuse released lions
lion5 = lion_pool.acquire()
lion5.interact("walking")

lion6 = lion_pool.acquire()
lion6.interact("hunting")
```

# Object Pool Design Pattern : *Reusing without destroying*

```
Creating a new Lion...  # Pre-creating 3 Lions
Creating a new Lion...
Creating a new Lion...

Lion is now roaring.
Lion is now playing.
Lion is now eating.

Creating a new Lion...  # New Lion created since the pool was empty
Lion is now resting.

Lion is now resting in the pool.
Lion is now resting in the pool.

Lion is now walking.  # Reuses a Lion from the pool
Lion is now hunting.  # Reuses another Lion from the pool
```

# Object Pool Design Pattern : Takeaways

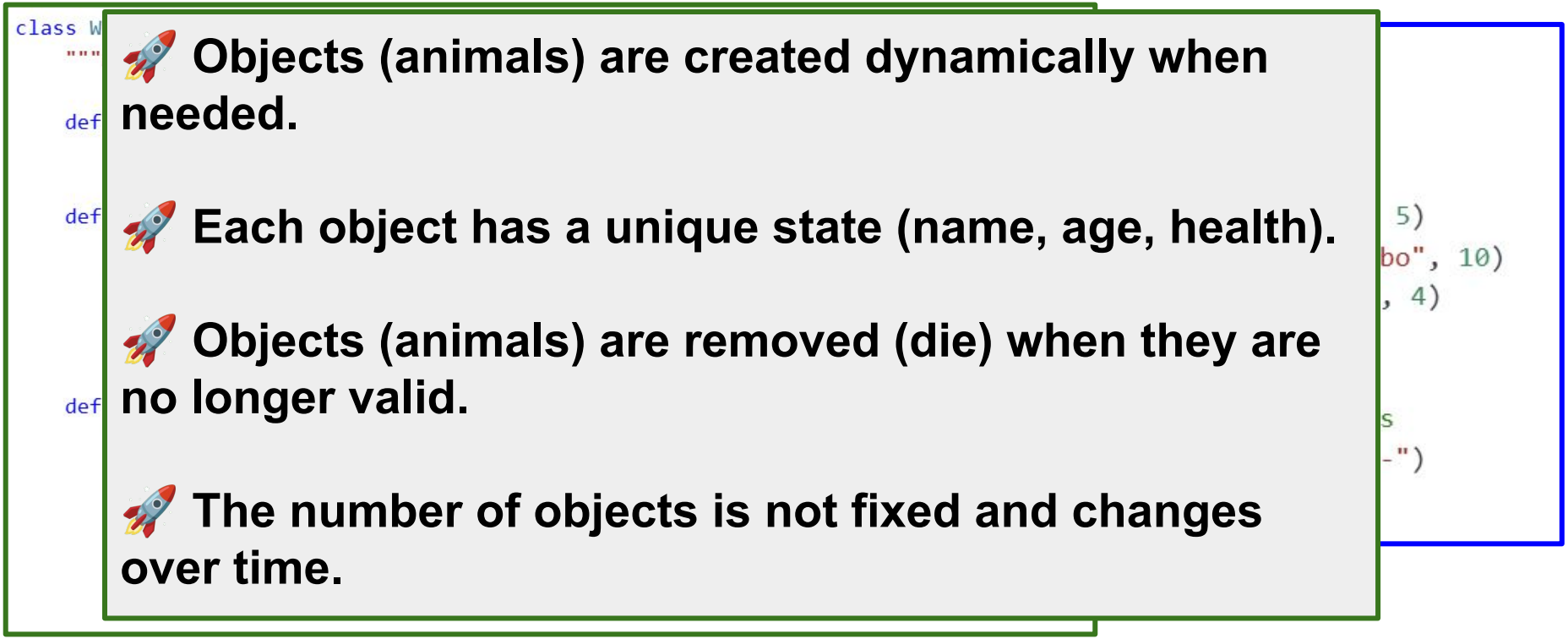
| Advantage | Explanation |
| --- | --- |
| **Performance Boost** | Avoids repeated object creation and destruction. |
| **Efficient Memory Usage** | Keeps a fixed number of objects active, reducing memory bloat. |
| **Ideal for Repeated Interactions** | Perfect when the ***same objects are used frequently***. |

# When does the pool get muddy?

# Object Pool Design Pattern : Takeaways

| Scenario | Why Object Pool is NOT Ideal? |
|---|---|
| **Animals require unique attributes** | If every animal must have a distinct identity, pooling might cause state issues. |
| **Short-lived animals** | If animals are rarely reused, pooling adds unnecessary complexity. |
| **Memory is not a concern** | If system resources are sufficient, object pooling may be overkill. |

# Dynamic Object Mgmt.: *Much better off without a pool*!




🚀 **Objects (animals) are created dynamically when needed.**

🚀 **Each object has a unique state (name, age, health).**

🚀 **Objects (animals) are removed (die) when they are no longer valid.**

🚀 **The number of objects is not fixed and changes over time.**

# Object Pool Design Pattern : Takeaways

| Issue | Why Object Pool Fails? |
|---|---|
| **Animals are not reusable** | Once an animal **dies**, it cannot be returned to the pool. |
| **Each animal is unique** | Different names, ages, and health conditions mean animals **cannot be simply reset and reused**. |
| **State retention is required** | Object pools usually **reset object states**, but these animals have **progressive aging and health decline**. |
| **Variable object count** | Object pools **assume a fixed number of objects**, but here, animals are **born and die dynamically**. |

# Object Oriented Programming

## Creational Design Patterns: Prototype

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Prototype Design Pattern : Let's *clone from a template!*

```python
import copy
import random

class Animal:
    """Represents an animal that can be cloned using the Prototype Pattern."""

    def __init__(self, species, name, age, health=None):
        self.species = species
        self.name = name
        self.age = age
        self.health = health if health is not None else random.randint(50, 100)

    def live_a_year(self):
        """Simulates the animal aging."""
        self.age += 1
        self.health -= random.randint(5, 20)  # Health decreases over time
        print(f"{self.name} (a {self.species}) is now {self.age} years old with health {self.health}")

    def clone(self, name):
        """Clones the animal and assigns a new name."""
        new_animal = copy.deepcopy(self)  # Ensures independent attributes
        new_animal.name = name  # Give the clone a new name
        return new_animal

    def describe(self):
        """Displays animal details."""
        print(f"Species: {self.species}, Name: {self.name}, Age: {self.age}, Health: {self.health}")
```

# Prototype Design Pattern : Let's not re-initialize *common stuffs*

```
# Create an original lion named Simba
lion_prototype = Animal("Lion", "Simba", 5)

# Clone the lion and give it a new name
lion_clone1 = lion_prototype.clone("Nala")
lion_clone2 = lion_prototype.clone("Mufasa")

# Modify clones independently
lion_clone1.age += 2
lion_clone2.health -= 15

# Display original and cloned animals
print("\nOriginal Lion:")
lion_prototype.describe()

print("\nCloned Lion 1:")
lion_clone1.describe()

print("\nCloned Lion 2:")
lion_clone2.describe()
```

*Only what is needed*

🚀 **Fast Object Creation** – Instead of reinitializing animals with all their properties

🚀 **Preserves Object State** – A copied animal inherits attributes like **age, species, health**

🚀 **Avoids Redundant Initialization** – If many animals are similar (e.g., multiple lions with slight variations), cloning is more efficient

Can we get a “manager” (zoo-keeper) instead?

(somewhat similar to Object Pool)

# Prototype Design Pattern : Let's create a prototype manager

```python
class Zoo:
    """Manages animals using the Prototype Pattern."""

    def __init__(self):
        self.prototypes = {}  # Stores animal prototypes

    def add_prototype(self, species, animal):
        """Registers an animal as a prototype."""
        self.prototypes[species] = animal

    def clone_animal(self, species, name):
        """Clones an animal prototype and gives it a new name
        if species in self.prototypes:
            return self.prototypes[species].clone(name)
        else:
            print(f"No prototype found for species: {species}
            return None
```

```python
# Create a zoo and register prototypes
zoo = Zoo()

# Create prototype animals
zoo.add_prototype("Lion", Animal("Lion", "Simba", 5))
zoo.add_prototype("Elephant", Animal("Elephant", "Dumbo", 10))

# Clone animals based on prototype
lion1 = zoo.clone_animal("Lion", "Scar")
elephant1 = zoo.clone_animal("Elephant", "Babar")

# Modify cloned animals
lion1.health -= 10
elephant1.age += 3

# Display cloned animals
print("\nCloned Lion:")
lion1.describe()

print("\nCloned Elephant:")
elephant1.describe()
```

# Prototype Design Pattern : Let's *clone with C++!*

```cpp
#include <iostream>
#include <cstring>  // For strcpy and strlen

class Animal {
private:
    char* species;  // Dynamically allocated memory for species name
    char* name;     // Dynamically allocated memory for animal's name
    int age;
    int health;

public:
    // Constructor
    Animal(const char* speciesName, const char* animalName, int animalAge, int animalHealth)
        : age(animalAge), health(animalHealth) {

        species = new char[strlen(speciesName) + 1];  // Allocate memory
        strcpy(species, speciesName);                 // Copy species name

        name = new char[strlen(animalName) + 1];      // Allocate memory
        strcpy(name, animalName);                     // Copy animal name
    }

    // Deep Copy Constructor
    Animal(const Animal& other)
        : age(other.age), health(other.health) {  // Copy primitive attributes

        species = new char[strlen(other.species) + 1];  // Allocate memory for species
        strcpy(species, other.species);

        name = new char[strlen(other.name) + 1];  // Allocate memory for name
        strcpy(name, other.name);
    }
```

Direct initialization

**BEWARE!!**

```cpp
class Animal {
public:
    Animal(Animal other) {  // ❌ Wrong: Passed by value (calls copy constructor)
        std::cout << "Copy Constructor" << std::endl;
    }
};

int main() {
    Animal a1;   // Triggers copy constructor
    Animal a2 = a1;  // Calls copy constructor → Recursively calls itself forever
}
```

```cpp
Animal(const Animal& other) {  // ✅ Safe: `other` cannot be modified
    // Copy logic
}
```

# Prototype Design Pattern : Let's *clone with C++!*

```
    // Destructor
    ~Animal() {
        delete[] species;
        delete[] name;
    }

    // Clone function (returns a deep copy of the object)
    Animal* clone(const char* newName) {
        return new Animal(species, newName, age, health);
    }

    // Method to simulate aging
    void live_a_year() {
        age++;
        health -= (rand() % 16) + 5;  // Reduce health by a random value between 5-20
    }

    // Display details
    void describe() const {
        std::cout << "Species: " << species << ", Name: " << name
                  << ", Age: " << age << ", Health: " << health << std::endl;
    }
};
```

```
// Main function to test cloning
int main() {
    // Create original animal
    Animal lion("Lion", "Simba", 5, 90);

    // Clone the lion and give it a new name
    Animal* lionClone1 = lion.clone("Nala");
    Animal* lionClone2 = lion.clone("Mufasa");

    // Modify the cloned animals
    lionClone1->live_a_year();  // Nala ages
    lionClone2->live_a_year();  // Mufasa ages

    // Display original and cloned objects
    std::cout << "\nOriginal Lion:" << std::endl;
    lion.describe();

    std::cout << "\nCloned Lion 1:" << std::endl;
    lionClone1->describe();

    std::cout << "\nCloned Lion 2:" << std::endl;
    lionClone2->describe();

    // Free memory for cloned objects
    delete lionClone1;
    delete lionClone2;

    return 0;
}
```

# Prototype: Key Takeaways

| Advantage | Descriptions |
|---|---|
| **Fast Object Creation** | Cloning is faster than re-initializing every attribute |
| **Encapsulated Cloning Logic** | *Objects know how to clone themselves*, keeping code clean |
| **Independent Objects** | Each cloned animal is a separate object in memory and can be modified independently<br>*NT: Does NOT automatically assign a new unique ID* |
| **Ideal for Large Systems** | Useful in **games, simulations, and automation** where predefined templates are needed |

# Prototype: Regular vs Deep Constructors

| Feature | Regular Constructor | Deep Copy Constructor |
|---|---|---|
| **Purpose** | Initializes a new object | Creates a copy of an existing object |
| **Used When?** | `Animal a1("Lion", 5);` | `Animal a2 = a1;` |
| **Handles Dynamic Memory?** | No | Yes (Allocates new memory) |
| **Prevents Memory Issues?** | No | Yes (Avoids shared pointers) |
| **When Needed?** | Always for new objects | Needed when objects have pointers |

# Prototype: Python vs C++

| Feature | Python | C++ |
|---|---|---|
| **Memory Management** | Handled automatically | Manual allocation (`new`) and deallocation (`delete`) |
| **Cloning Method** | Uses `copy.deepcopy()` | Uses a deep copy constructor and `clone()` function |
| ***String Storage*** | *Uses Python `str` (immutable)* | *Uses dynamically allocated `char*`* |
| **Garbage Collection** | Automatic | Needs explicit destructor |

# Prototype: Python vs C++

```python
a = "Lion"
b = a
b = "Tiger"  # Creates a new string object; `a` remains "Lion"

print(a)  # Output: Lion
print(b)  # Output: Tiger
```

String Default: *Immutable*

```cpp
char a[] = "Lion";
char* b = a;
b[0] = 'T';  // Modifies `a` as well

std::cout << a;  // Output: "Tion"
```

String Default: *Mutable*

# Prototype: Python vs C++

```python
a = "Lion"
b = a
b = "Tiger"  # Creates a new string object; `a` remains "Lion"

print(a)  # Output: Lion
print(b)  # Output: Tiger
```

String Default: *Immutable*

`species` and `name` are immutable (`str`) → *no need of deep copying*.

```python
def clone(self, name):
    """Clones the animal and assigns a new name."""
    new_animal = copy.deepcopy(self)  # Ensures independent attributes
    new_animal.name = name  # Give the clone a new name
    return new_animal
```

# Prototype: Python vs C++

```
char a[] = "Lion";
char* b = a;
b[0] = 'T';  // Modifies `a` as well

std::cout << a;  // Output: "Tion"
```

String Default: *mutable*

***Explicit*** deep copy of `char*` to prevent memory issues.

```
// Deep Copy Constructor
Animal(const Animal& other)
    : age(other.age), health(other.health) {  // ✅ Copy primitive values

    // ✅ Allocate new memory for species and copy content
    species = new char[strlen(other.species) + 1];
    strcpy(species, other.species);

    // ✅ Allocate new memory for name and copy content
    name = new char[strlen(other.name) + 1];
    strcpy(name, other.name);

    std::cout << "Deep Copy Constructor: Copied " << name << "!\n";
}
```

# Can cloning go haywire?

# Prototype: When to avoid?

| Criteria | Prototype is a BAD Choice When... | Why? |
|---|---|---|
| **Objects Have Unique Identity** | Each object has a unique **ID, serial number, or real-world identity**. | Cloning would create **duplicate IDs**, breaking identity uniqueness. |
| **Objects Contain Constantly Changing Data** | The object's attributes (e.g., **GPS, health status, timestamps**) change frequently. | A cloned object **copies outdated values**, leading to **inaccurate data**. |
| **Object Creation is Simple and Efficient** | The object's **construction is cheap** (e.g., primitive types, lightweight objects). | Creating a new object from scratch **is more efficient** than cloning. |

# Prototype: When to avoid?

| Criteria | Prototype is a BAD Choice When... | Why? |
|---|---|---|
| **Cloning Would Cause Logical Errors** | Objects require **strict uniqueness** (e.g., **wildlife tracking, transaction records**). | Cloning creates **misleading copies**, making real-world tracking **impossible**. |
| **Objects Have Complex Dependencies** | Objects rely on **external resources (e.g., database connections, network sockets)**. | Cloning would **copy references**, leading to **shared state issues**. |
| **Object's State Cannot Be Copied Reliably** | Objects hold **unique or non-copyable attributes** (e.g., **file handles, hardware locks**). | Cloning such objects **causes runtime errors** or **unintended side effects**. |

# Object Oriented Programming

## Structural Design Patterns: Adapter

IT560


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Creational Design Pattern : Object creation is just the first step

- Do not help in ***structuring*** **relationships between objects** to form larger more complex objects

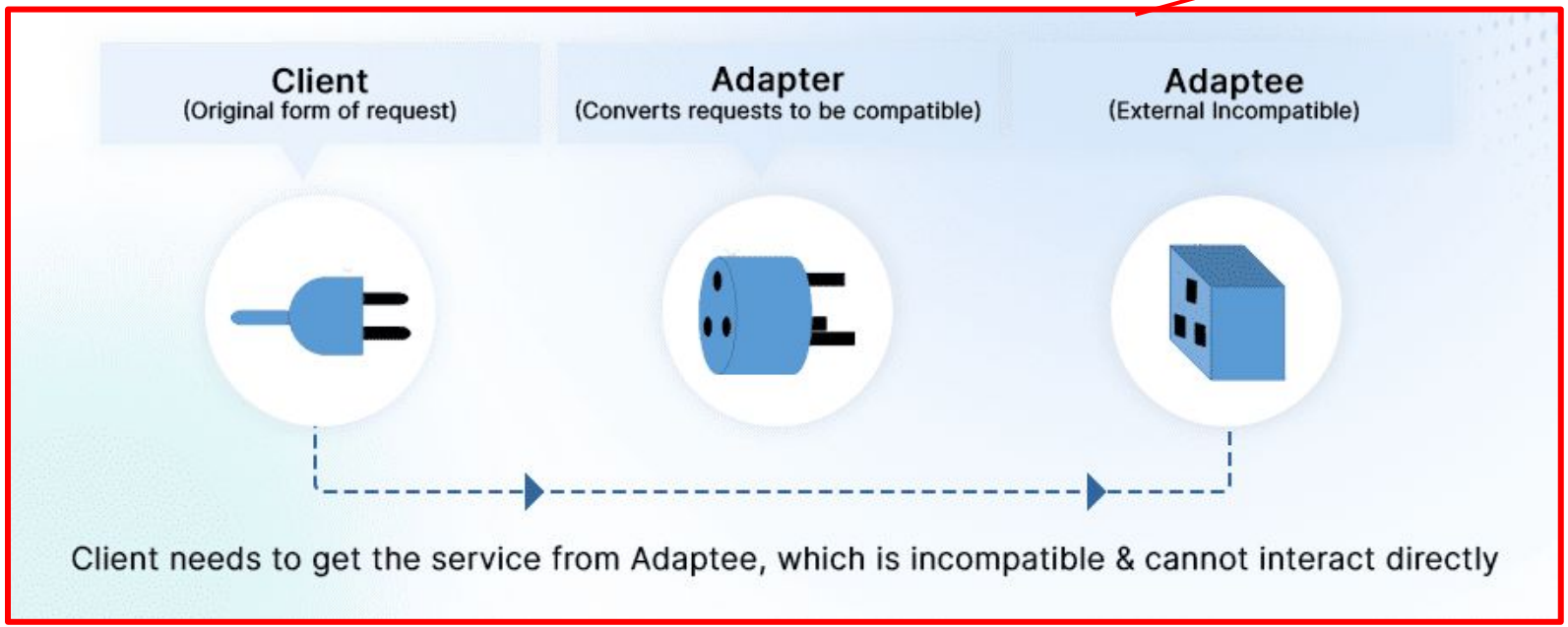
- Do not help in avoiding tightly coupled objects (just takes care of tightly coupled client-server relationships)

- Do not help much in object code duplication (just focuses on reducing construction code duplication)

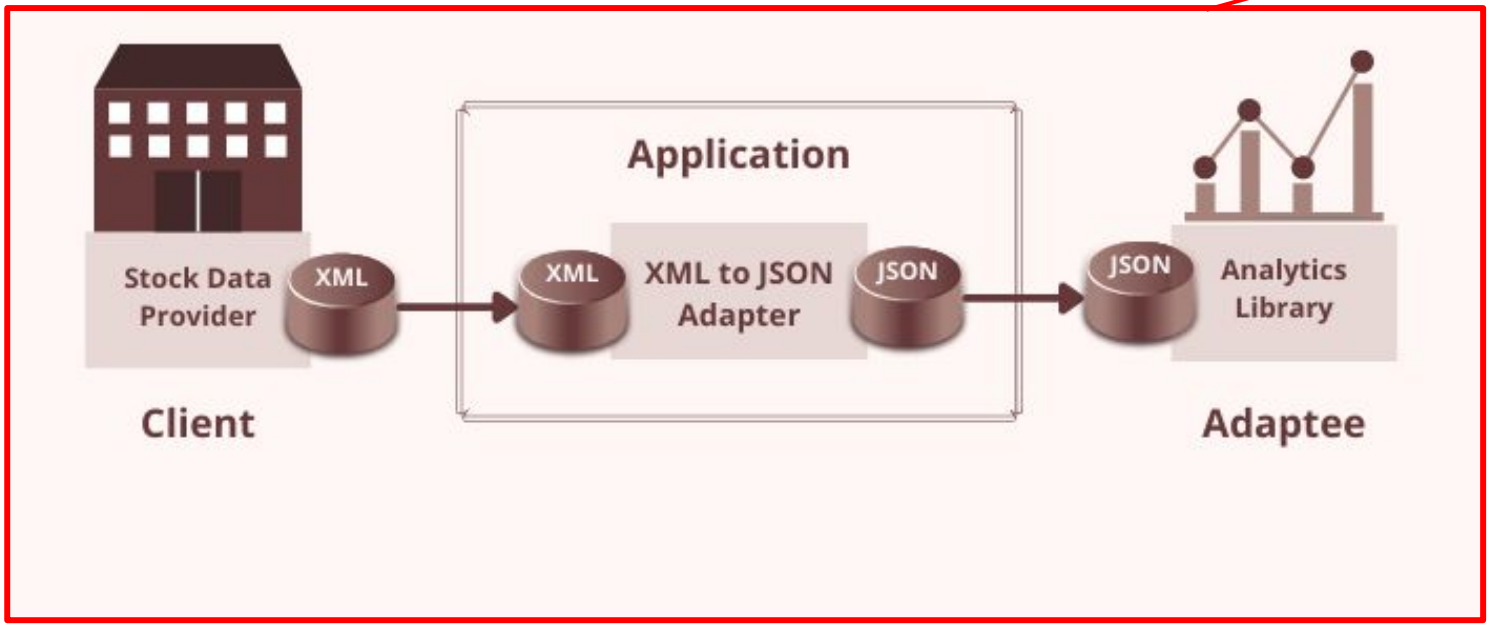
# Structural Design Pattern: A game of lego

- The art of how to make *complex, larger* objects
- Requires perfect "*fitting*" of the components (smaller objects) - component relationship
- Requires *solving incompatibility* between components

# Adapter Design Pattern : Making objects compatible



# Adapter Design Pattern : Making objects compatible



# Client incompatible with newly introduced Class

```python
# Target Interface
class Animal:
    def make_sound(self):
        pass

# Concrete implementation of Animal: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Adaptee: Cat with an incompatible interface
class Cat:
    def meow(self):
        return "Meow!"
```

```python
# Client Code: A function that uses the Animal interface
def animal_sound(animal: Animal):
    print(animal.make_sound())
```

```python
# Usage:
# Dog directly implements Animal
dog = Dog()
# Cat does not implement Animal
cat = Cat()

# The client code works for Dog but not for Cat:
animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Error!!
```

# Adapter Design Pattern : Making objects compatible

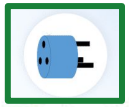
```python
# Target Interface
class Animal:
    def make_sound(self):
        pass

# Concrete implementation of Animal: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Adaptee: Cat with an incompatible interface
class Cat:
    def meow(self):
        return "Meow!"
```

```python
# Adapter: Makes a Cat behave like an Animal
class CatAdapter(Animal):
    def __init__(self, cat):
        self.cat = cat

    def make_sound(self):
        # Translate the call to the Cat's meow method
        return self.cat.meow()
```

# Adapter Design Pattern : Making objects compatible

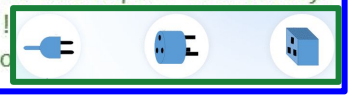
```python
# Target Interface
class Animal:
    def make_sound(self):
        pass

# Concrete implementation of Animal: Dog
class Dog(Animal):
    def make_sound(self):
```

```python
# Adapter: Makes a Cat behave like an Animal
class CatAdapter(Animal):
    def __init__(self, cat):
        self.cat = cat

    def make_sound(self):
        # Translate the call to the Cat's meow method
        return self.cat.meow()
```



```python
# Usage:
# Dog directly implements Animal
dog = Dog()
# Cat does not implement Animal
cat = Cat()

# The client code works for Dog but not for Cat:
animal_sound(dog)  # Output: Woof!
animal_sound(cat)  # Error!!
```

```python
# Usage:
# Dog directly implements Animal
dog = Dog()
# Cat does not implement Animal
cat = Cat()
# Adapter wraps the Cat
cat_adapter = CatAdapter(cat)

# The client code works with both Dog and CatAdapter seamlessly:
animal_sound(dog)         # Output: Woof!
animal_sound(cat_adapter)  # Output: Meo
```



# Adapter: Key Takeaways

| Advantage | Description |
|---|---|
| **Interface Compatibility** | Allows disparate interfaces to work together without modifying existing code. |
| **Reusability** | Enables the reuse of legacy or third-party classes by adapting them to a required interface. |
| **Decoupling** | Reduces direct dependencies between components, making them easier to maintain and extend. |
| **Flexibility** | Simplifies integration of new components or systems, even if their interfaces differ. |
| **Scalability** | Facilitates system growth by allowing components to be added or replaced with minimal disruption. |

# Adapter Design Pattern : New adapter for each adaptee

```python
# Target Interface
class Animal:
    def make_sound(self):
        pass

# Concrete implementation of Animal: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Adaptee: Cat with an incompatible interface
class Cat:
    def meow(self):
        return "Meow!"
```

```python
# Adapter: Makes a Cat behave like an Animal
class CatAdapter(Animal):
    def __init__(self, cat):
        self.cat = cat

    def make_sound(self):
        # Translate the call to the Cat's meow method
        return self.cat.meow()
```

```python
# New Adaptee: Bird with a tweet() method
class Bird:
    def tweet(self):
        return "Tweet tweet!"
```

```python
# New Adapter for Bird
class BirdAdapter(Animal):
    def __init__(self, bird):
        self.bird = bird

    def make_sound(self):
        return self.bird.tweet()
```

Can we do better?

# Adapter Design Pattern : Generic adapter for all adaptee

```python
# Target Interface
class Animal:
    def make_sound(self):
        pass

# Concrete implementation of Animal: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

# Adaptee: Cat with an incompatible interface
class Cat:
    def meow(self):
        return "Meow!"
```

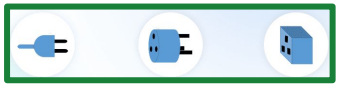
```python
# New Adaptee: Bird with a tweet() method
class Bird:
    def tweet(self):
        return "Tweet tweet!"
```

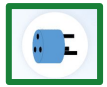
```python
class GenericAdapter(Animal):
    def __init__(self, adaptee, method_name):
        self.adaptee = adaptee
        self.method_name = method_name

    def make_sound(self):
        method = getattr(self.adaptee, self.method_name)
        return method()  # Dynamically call the correct method
```

```python
# Creating instances of different animals
dog = Dog()                    # Already implements Animal
cat = Cat()                    # Has meow() method
bird = Bird()                  # Has tweet() method

# Using the generic adapter for Cat and Bird
cat_adapter = GenericAdapter(cat, "meow")
bird_adapter = GenericAdapter(bird, "tweet")

# Client code remains unchanged
animal_sound(dog)          # Output: Woof!
animal_sound(cat_adapter)  # Output: Meow!
animal_sound(bird_adapter) # Output: Tweet tweet!
```

# Adapter Design Pattern : Let's go C++!!

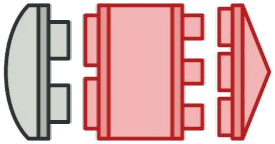
```cpp
#include <iostream>
#include <string>

// Target interface: Animal with a virtual makeSound method.
class Animal {
public:
    virtual std::string makeSound() const = 0;
    virtual ~Animal() {}
};

// Generic Adapter: Adapts any type T to the Animal interface.
/* It takes a pointer-to-member function (SoundMethod) that
   returns a std::string. */
template <typename T>
class GenericAdapter : public Animal {
public:
    /* Define a type alias for a pointer-to-member function that
    takes no parameters and returns std::string. */
    using SoundMethod = std::string (T::*)() const;

    /* Constructor takes a reference to the adaptee and
       a pointer-to-member function. */
    GenericAdapter(const T& adaptee, SoundMethod method)
        : adaptee_(adaptee), method_(method) {}

    /* Implement the Animal interface by calling
       the specified method on the adaptee.*/
    std::string makeSound() const override {
        return (adaptee_.*method_)();
    }

private:
    T adaptee_;
    SoundMethod method_;
};

class Dog {
public:
    std::string bark() const {
        return "Woof!";
    }
};

class Cat {
public:
    std::string meow() const {
        return "Meow!";
    }
};

int main() {
    Dog dog;
    Cat cat;

    // Use the generic adapter to adapt Cat to the Animal interface.
    GenericAdapter<Cat> catAdapter(cat, &Cat::meow);

    std::cout << "Dog: " << dog.makeSound() << std::endl;        // Output: Woof!
    std::cout << "Cat: " << catAdapter.makeSound() << std::endl;   // Output: Meow!

    return 0;
}
```





Is adapting always good?

# Adapter: Cautionary comments



| Con | Description |
|---|---|
| Increased Complexity | Adds extra layers of abstraction that *may* complicate system understanding. |
| Maintenance Overhead | Additional adapter code must be maintained, especially if underlying *interfaces change* |
| Masking Design Issues | Can hide underlying design flaws by serving as a quick fix rather than addressing core issues |
| Complex Testing | More layers (client, adapter, adaptee) mean additional testing is required to ensure proper integration |

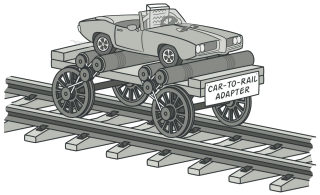
# Object Oriented Programming

## Structural Design Patterns: Facade

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

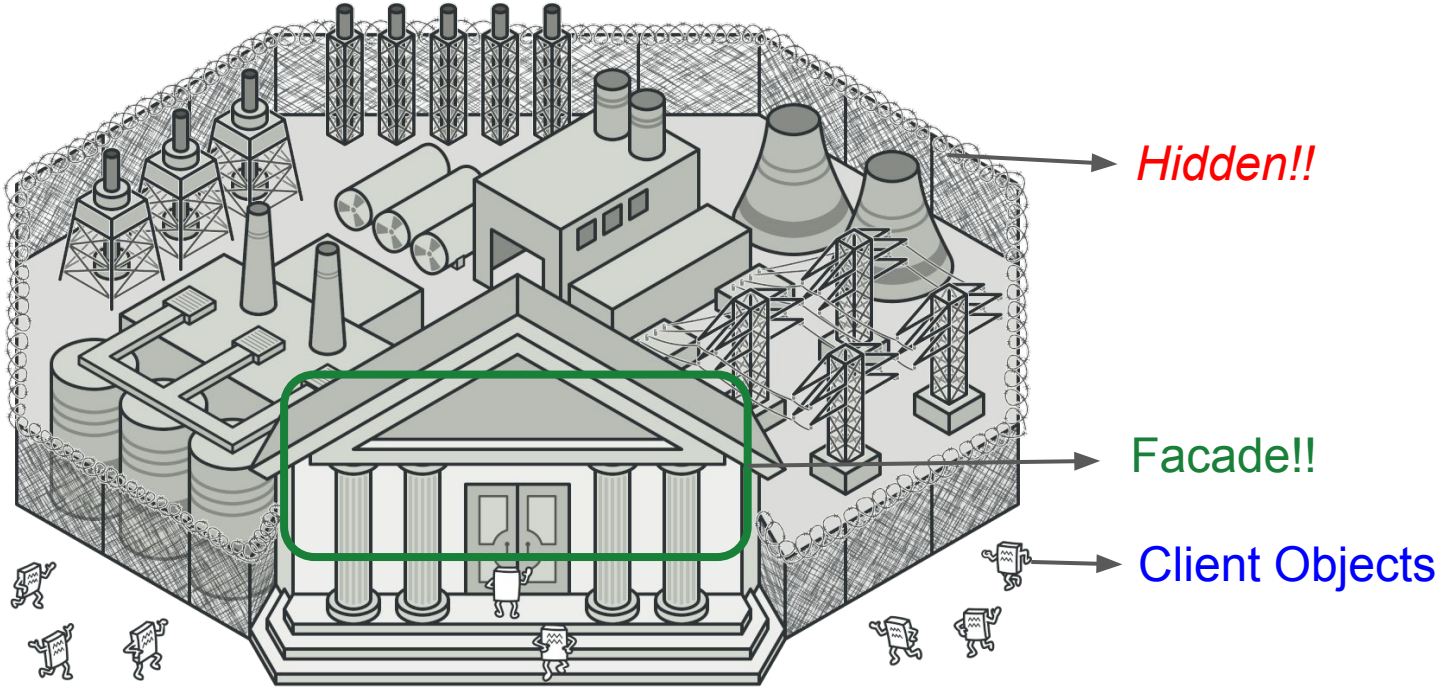
# Adapter Design Pattern : Compatibility with *single* interface



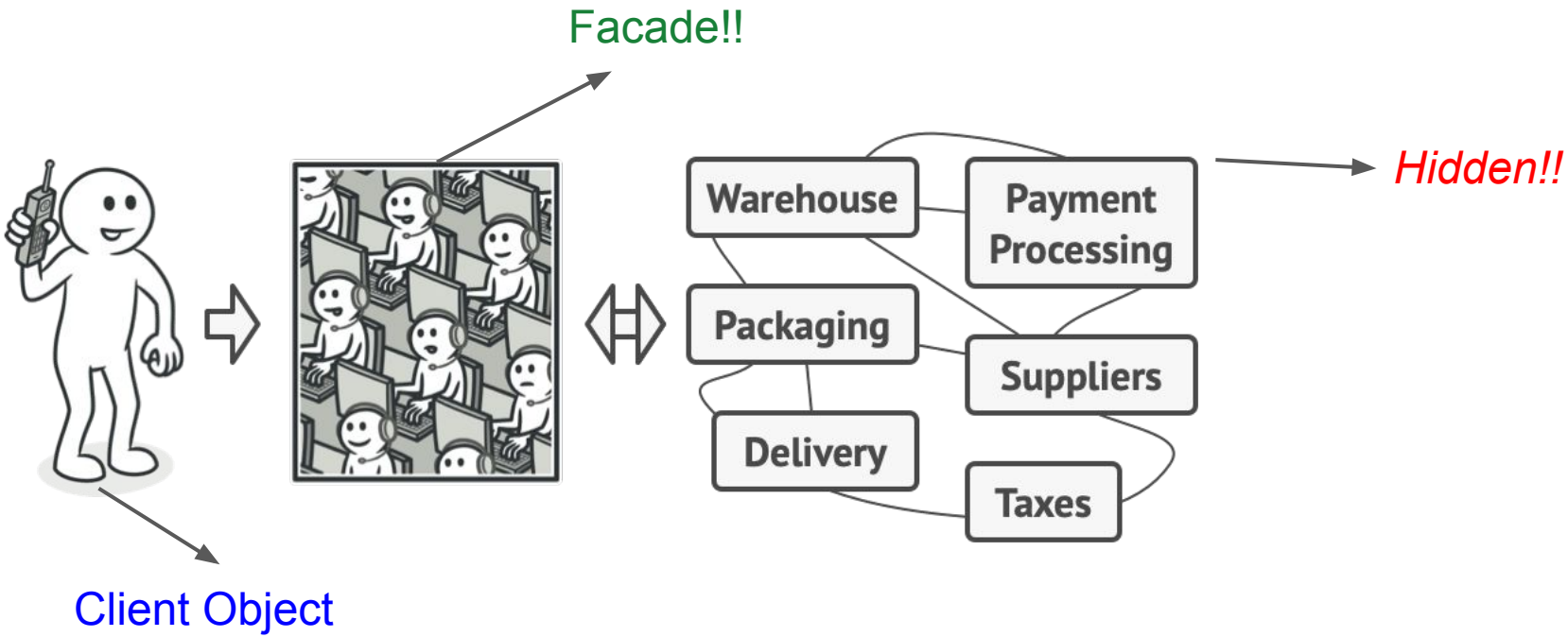

Car vs. Rail track

- Do not help in having a **unified** interface to a ***complex (multi) sub-system*** (with ***multiple operations***)
- Do not help in **coordination** within the complex sub-system
- Do not help decouple client with a complex sub-system

# What is a Facade?



# A concrete example: *Ordering over the phone*



# Key Takeaways: Facade

| Aspect | Description |
|---|---|
| **Purpose** | Simplifies interaction with complex subsystems by providing a unified interface. |
| **Benefits** | Decouples clients from subsystem details, hides complexity, and makes the system easier to use. |
| **Usage Example** | An order processing system that integrates inventory checks, payment processing, and shipping. |
| **Client Interaction** | The client calls a single method (e.g., `process_order()`) on the facade, instead of multiple subsystem calls. |

# Takeaways: Facade (vs. Adapter)

| Feature | Differences |
|---|---|
| **Subsystem Coordination** | *Orchestrates* interactions between multiple classes or subsystems<br><br>Vs. Only wraps a single interface |
| **Unified High-Level Abstraction** | Comprehensive, simplified interface to a ***complex system***, hiding the internal complexity.<br><br>Vs. Does not consolidate multiple operations into one cohesive interface. |
| **Decoupling from Complex Subsystems** | Decouples the client from an ***entire set*** of interrelated subsystems.<br><br>Vs. Solely focused on making one incompatible interface work with another. |

# Toy example of creating sub-systems out of similar classes

```python
# Define Animal Classes with Different Behaviors

class Dog:
    def bark(self):
        return "Woof!"

    def run(self):
        return "The dog runs swiftly."

    def eat(self):
        return "The dog eats kibble."

class Cat:
    def meow(self):
        return "Meow!"

    def jump(self):
        return "The cat jumps gracefully."

    def eat(self):
        return "The cat eats fish."

class Bird:
    def tweet(self):
        return "Tweet tweet!"

    def fly(self):
        return "The bird flies high."

    def eat(self):
        return "The bird eats seeds."
```

```python
# Subsystem 1: SoundSubsystem
class SoundSubsystem:
    def get_sound(self, animal):
        if isinstance(animal, Dog):
            return animal.bark()
        elif isinstance(animal, Cat):
            return animal.meow()
        elif isinstance(animal, Bird):
            return animal.tweet()
        else:
            return "Unknown s
```

```python
# Subsystem 2: MovementSubsystem
class MovementSubsystem:
    def get_movement(self, animal):
        if isinstance(animal, Dog):
            return animal.run()
        elif isinstance(animal, Cat):
            return animal.jump()
        elif isinstanc
            return ani
        else:
            return "No
```

```python
# Subsystem 3: FeedingSubsystem
class FeedingSubsystem:
    def get_feeding(self, animal):
        return animal.eat()
```

# Don’t want Facade? Going to give client a lot of headache!!

```
# Client code without using the Facade
if __name__ == "__main__":
    print("=== Without Facad
    dog = Dog()
    cat = Cat()
    bird = Bird()

    # Client must instantiat
    sound_system = SoundSubs
    movement_system = Moveme
    feeding_system = Feeding
```

```
for animal in [dog, cat, bird]:
    # The client is responsible for coordinating the calls.
    try:
        profile = {}
        # Mistake: The client might forget to call one subsystem or call them in the wrong order.
        profile['Sound'] = sound_system.get_sound(animal)
        # For demonstration, let's simulate an error by forgetting the movement call.
        # profile['Movement'] = movement_system.get_movement(animal)  # This line is accidentally omitted.
        profile['Feeding'] = feeding_system.get_feeding(animal)

        # Now, the profile is incomplete and using it might lead to runtime errors.
        if 'Movement' not in profile:
            raise ValueError("Incomplete profile: Missing Movement information!")

        print("Animal Profile:")
        for key, value in profile.items():
            print(f"  {key}: {value}")
        print()
    except Exception as e:
        print("Error while retrieving animal profile:", e)
```

# Facade abstracts the sub-systems!

```
# Facade: AnimalFacade
class AnimalFacade:
    def __init__(self):
        self.sound_system = SoundSubsystem()
        self.movement_system = MovementSubsystem()
        self.feeding_system = FeedingSubsystem()

    def get_animal_profile(self, animal):
        profile = {}
        profile['Sound'] = self.sound_system.get_sound(animal)
        profile['Movement'] = self.movement_system.get_movement(animal)
        profile['Feeding'] = self.feeding_system.get_feeding(animal)
        return profile
```

# Facade abstracts the sub-systems: *Client has no headache!*

```
# Client Code
if __name__ == "__main__":
    dog = Dog()
    cat = Cat()
    bird = Bird()

    facade = AnimalFacade()

    for animal in [dog, cat, bird]:
        profile = facade.get_animal_profile(animal)
        print("Animal Profile:")
        for key, value in profile.items():
            print(f"  {key}: {value}")
        print()
```

*Trivia: Extracts the tuple from dictionary*

Can we do better?

# FP-style optimization

```python
# Facade: AnimalFacade
class AnimalFacade:
    def __init__(self):
        self.sound_system = SoundSubsystem()
        self.movement_system = MovementSubsystem()
        self.feeding_system = FeedingSubsystem()

    def get_animal_profile(self, animal):
        profile = {}
        profile['Sound'] = self.sound_system.get_sound(animal)
        profile['Movement'] = self.movement_system.get_movement(animal)
        profile['Feeding'] = self.feeding_system.get_feeding(animal)
        return profile
```

*Trivia: Decorator enforcing lazy (i.e., dynamic) calling → call-when-needed:*
READ-ONLY

```python
class LazyAnimalFacade:
    def __init__(self):
        self._sound_system = None
        self._movement_system = None
        self._feeding_system = None

    @property
    def sound_system(self):
        if self._sound_system is None:
            self._sound_system = SoundSubsystem()
        return self._sound_system

    @property
    def movement_system(self):
        if self._movement_system is None:
            self._movement_system = MovementSubsystem()
        return self._movement_system

    @property
    def feeding_system(self):
        if self._feeding_system is None:
            self._feeding_system = FeedingSubsystem()
        return self._feeding_system

    def get_animal_profile(self, animal):
        return {
            "Sound": self.sound_system.get_sound(animal),
            "Movement": self.movement_system.get_movement(animal),
            "Feeding": self.feeding_system.get_feeding(animal),
        }
```

Keep it uninitialized

# Is creating facade always good?

# Cautionary comments

| Aspect | Advantages | Disadvantages |
|---|---|---|
| **Complexity Handling** | Reduces complexity by hiding subsystem details. | Can lead to unnecessary abstraction if overused. |
| **Dependency Management** | Decouples client from subsystem changes. | Clients need update if Facade needs update → *an adapter to the facade??* |
| **Performance** | Makes API calls easier and more structured. | Can introduce slight overhead when forwarding requests. |

# Object Oriented Programming

## Structural Design Patterns: Decorator

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

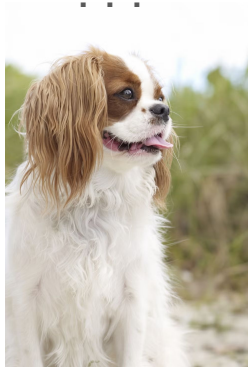
# Why should we think about Decorators?

```
class Dog:
    def make_sound(self):
        return "Woof"
```

```
# Loud Dog: A subclass that modifies the sound
class LoudDog(Dog):
    def make_sound(self):
        return "WOOF!!!"
```

```
# Whisper Dog: Another subclass that modifies
# the sound differently
class WhisperDog(Dog):
    def make_sound(self):
        return "woof..."
```

???



# What's wrong with class inheritance?

- **Class Explosion:** New subclass for each new behavior (e.g., loud, whisper, echo, etc.)
  - Combo behavior?? → Exponential growth!

- **Lack of Flexibility:** Want to add a new behavior? Or modify an existing one? Change the class hierarchy
  - No way to mix and match behaviors at runtime *without creating additional subclasses*

- **Maintenance Challenges:** A new subclass? → additional maintenance overhead
  - Base class Dog got changed? → Update all the subclasses to maintain consistency

# How does Decorator help? Modifying and extending on-the-fly

```python
# Base interface for Animal
class Animal:
    def make_sound(self):
        pass
```

```python
# Concrete implementation: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof"
```

```python
# Loud Dog: A subclass that modifies the sound
class LoudDog(Dog):
    def make_sound(self):
        return "WOOF!!!"
```

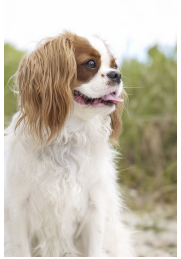
```python
# Base Decorator class that extends Animal
class AnimalDecorator(Animal):
    def __init__(self, animal: Animal):
        self._animal = animal

    def make_sound(self):
        return self._animal.make_sound()
```

```python
class LoudDecorator(AnimalDecorator):
    def make_sound(self):
        # Modify the original sound dynamically
        return self._animal.make_sound().upper() + "!!!"
```

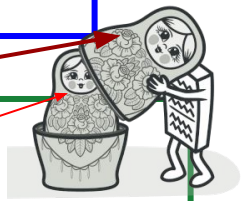
```python
class WhisperDecorator(AnimalDecorator):
    def make_sound(self):
        # Modify the original sound dynamically
        return self._animal.make_sound().lower() + "..."
```

# How does Decorator help? Modifying and extending on-the-fly

```
# Client code:
dog = Dog()
print("Original:", dog.make_sound())            # Woof

loud_dog = LoudDecorator(dog)
print("Loud:", loud_dog.make_sound())            # WOOF!!!

whisper_dog = WhisperDecorator(dog)
print("Whisper:", whisper_dog.make_sound())      # woof...
```



```
# Even chain decorators:
loud_whisper_dog = LoudDecorator(WhisperDecorator(dog))
print("Loud then Whisper:", loud_whisper_dog.make_sound())  # WOOF...!!!
```

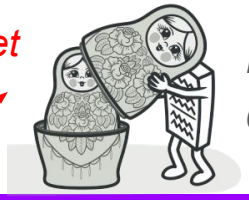


# How does Decorator help? Adding new behavior (eat) on-the-fly

```python
# Base interface for Animal
class Animal:
    def make_sound(self):
        pass
```

```python
# Concrete implementation: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof"
```

*No eating behavior yet*



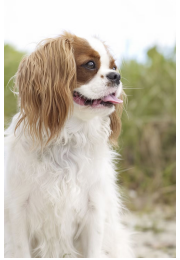
*Needs to be compatible!*

```python
# AnimalEatDecorator adds an 'eat' behavior while preserving other behaviors
class AnimalEatDecorator(AnimalDecorator):
    def eat(self, food: str):
        # Uses the current make_sound() to incorporate the loud behavior
        return f"{self.make_sound()} eats {food}"
```

```python
def make_sound(self):
    # Modify the original sound dynamically
    return self._animal.make_sound().upper() + "!!!"
```

```python
def make_sound(self):
    # Modify the original sound dynamically
    return self._animal.make_sound().lower() + "..."
```

# How does Decorator help? Adding new behavior (eat) on-the-fly



```
# Concrete implem
class Dog(Animal)
    def make_sour
        return "W
```

```
# Client Code
if __name__ == "__main__":
    # Create a base Dog
    dog = Dog()

    # First, decorate the dog to be loud
    loud_dog = LoudDecorator(dog)

    # Then, add the eat behavior on top of the loud dog
    loud_eating_dog = AnimalEatDecorator(loud_dog)

    # Testing the composed object:
    print(loud_eating_dog.make_sound())  # Expected output: "WOOF!!!"
    print(loud_eating_dog.eat("kibble"))   # Expected output: "WOOF!!! eats kibble"
```

```
# Base Decorator class that extends Animal
class AnimalDecorator(Animal):
    def __init__(self, animal: Animal):
        self._animal = animal

    def make_sound(self):
        return self._animal.make_sound()
```

Why does this work??

# Key takeaways

| Aspect | Description |
|---|---|
| **Purpose** | Dynamically add new behavior to objects without altering their structure. |
| **Key Benefit** | Adheres to the open/closed principle (SOLID); enhances flexibility and reusability. |
| **Implementation** | Involves creating a *base decorator class* and one or more concrete decorators. |
| **Usage** | When you need to extend the functionality of objects in a flexible and *combinable* way. |
| **Comparison to Subclassing** | More scalable and flexible than creating a separate subclass for each behavior combination. |

# How does the Decorator differ from the others?

| Aspect | Decorator | Adapter/ Facade |
|---|---|---|
| **Primary Purpose** | Dynamically *extend* or *modify* an object's behavior → wraps with additional functionality | Adapter converts an incompatible interface, while Facade simplifies interaction with complex subsystems. |
| **Behavior Extension** | Can add, remove, or combine behaviors at runtime *without altering the original object's code* | Typically static transformations or simplifications → no runtime modification of behavior |
| **Flexibility** | Enables *stacking* of multiple decorators for flexible composition of responsibilities | No dynamic behavior extension. |
| **Transparency to Client** | The client interacts with the decorated object using the same interface, *unaware of the added behaviors* | The client interacts with the adapted/facaded interface, but the primary goal is interface consistency or simplification |

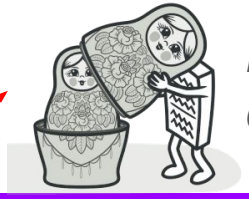
# Does Decorator ring a bell with FP?

| Aspect | Decorator Pattern | Functional Programming |
|---|---|---|
| **Extending Functionality** | Wraps objects to dynamically add or modify behavior without changing the original class. | Uses higher-order functions to create new functions by composing existing ones. |
| **Composability** | Decorators can be stacked or chained to combine multiple behaviors at runtime. | Functions can be composed together (e.g., using composition operators) to build complex operations. |
| **Flexibility** | Enables runtime modification and flexible extension of an object's capabilities. | Encourages building programs by composing small, reusable functions in different combinations. |

# Wait, but isn't this the subway making problem? (Builder)

| Aspect | Builder Pattern | Decorator Pattern |
|---|---|---|
| **Primary Purpose** | Focuses on constructing complex objects *step by step*, separating construction from representation. | Focuses on dynamically adding responsibilities or modifying the behavior of an object at runtime. |
| **Composition Type** | Composes various parts or components during object creation to form a new complex object. | Wraps an existing object to extend or alter its behavior without changing its interface. |
| **Timing** | The object is built once, with the *final* configuration being set at creation time. | The object's behavior can be modified or extended after creation, even at runtime. |
| **Use Case Example** | Building a complex configuration (like a house, or a complex document) where various parts need to be assembled in a specific order. | Adding extra functionalities (like logging, caching, or permission checks) to an object without modifying its existing code. |

# Can we make the decorators completely decoupled?



*Needs to be compatible!*

```
# AnimalEatDecorator adds an 'eat' behavior while preserving other behaviors
class AnimalEatDecorator(AnimalDecorator):
    def eat(self, food: str):
        # Uses the current make_sound() to incorporate the loud behavior
        return f"{self.make_sound()} eats {food}"
```

*Why should Eat decorator have to be aware of MakeSound decorator?*

# Why is decoupling needed? Reverse chaining may not work!

```python
# First eat
eat_dog = EatDecorator(dog)

# Then loud (this wraps the EatDecorator)
loud_eating_dog = LoudDecorator(eat_dog)

# Works:
print(loud_eating_dog.make_sound())  #

# FAILS:
print(loud_eating_dog.eat("kibble"))
# AttributeError: 'LoudDecorator' object has no attr
```

```python
# Base Decorator class that extends Animal
class AnimalDecorator(Animal):
    def __init__(self, animal: Animal):
        self._animal = animal

    def make_sound(self):
        return self._animal.make_sound()
```

```python
class LoudDecorator(AnimalDecorator):
    def make_sound(self):
        # Modify the original sound dynamically
        return self._animal.make_sound().upper() + "!!!"
```

```python
# Concrete implementation: Dog
class Dog(Animal):
    def make_sound(self):
        return "Woof"
```

Can we do better?

# How to make the chaining work in *any* order?

```python
# Base Decorator that extends Animal
class AnimalDecorator(A
    def __init__(self, a
        self._animal = a

    def make_sound(self)
        return self._ani

    def __getattr__(sel
        return getattr(s
```

```python
# MakeSoundDecorator that accepts a modality and applies it
class MakeSoundDecorator(AnimalDecorator):
    def __init__(self, animal: Animal, modality: str):
        super().__init__(animal)
        self.modality = modalit

    def make_sound(self):
        sound = self._animal.ma

        if self.modality == "lo
            return sound.upper(
        elif self.modality == "
            return sound.lower(
        else:
            raise ValueError(f"

    def __getattr__(self, name)
        return getattr(self._an
```

```python
# EatDecorator independently adds an 'eat' behavior
class EatDecorator(AnimalDecorator):
    def eat(self, food):
        base_animal = self._animal
        while hasattr(base_animal, '_animal'):
            base_animal = base_animal._animal

        class_name = base_animal.__class__.__name__
        return f"{class_name} is going to eat {food}"

    def __getattr__(self, name):
        return getattr(self._animal, name)
```

# Now any order works!

```
# ========================
# ✅ Client Code: Eating Loud vs Loud Eating
# ========================
if __name__ == "__main__":
    dog = Dog()

    print("\n=== Eating Loud Dog ===")
    eating_loud_dog = MakeSoundDecorator(EatDecorator(dog), "loud")
    print(eating_loud_dog.make_sound())  # WOOF!!!
    print(eating_loud_dog.eat("kibble")) # Dog is going to eat kibble

    print("\n=== Loud Eating Dog ===")
    loud_eating_dog = EatDecorator(MakeSoundDecorator(dog, "loud"))
    print(loud_eating_dog.make_sound())  # WOOF!!!
    print(loud_eating_dog.eat("kibble")) # Dog is going to eat kibble
```

Haven’t we seen decorators before?

Remember the @-styled functions?

# Decorator Design Pattern vs. Decorator Feature

| Decorator | Purpose |
|---|---|
| `@property` | Define a method as a read-only property |
| `@staticmethod` | Define a static method (no `self`) |
| `@classmethod` | Define a method that operates on the class |
| `@functools.lru_cache` | Cache results of expensive computations |
| `@functools.wraps` | Preserve function metadata in decorators |
| `@abc.abstractmethod` | Define abstract methods in base classes |
| `@dataclasses.dataclass` | Automatically generate data class boilerplate |
| `@contextlib.contextmanager` | Simplify creating context managers |

# Decorator Design Pattern vs. Decorator Feature

```python
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper
```

```python
class Dog:
    @logger  # <-- Python decorator feature
    def make_sound(self):
        return "Woof"
```

```python
dog = Dog()
dog.make_sound()
```

```
Calling make_sound
make_sound returned Woof
```

vs.

```
Woof
```

# Decorator Design Pattern vs. Decorator Feature

```python
def announce_action(func):
    def wrapper(*args, **kwargs):
        print("The animal is preparing to make a sound...")
        return func(*args, **kwargs)
    return wrapper
```

```python
class Dog:
    @announce_action  # <-- Decorator feature in action
    def make_sound(self):
        print("Woof")
```

```python
dog = Dog()
dog.make_sound()
```

```
The animal is preparing to make a sound...
Woof
```

vs.

```
Woof
```

# To summarize

| | Decorator Pattern | Decorator Feature (@) |
|---|---|---|
| **What** | Structural design pattern | Python syntax + language feature |
| **Purpose** | Add behavior to **objects** | Add behavior to **functions/classes** |
| **Method** | Class-based, runtime wrapping | Function/class wrapper, definition-time |
| **Examples** | GUI widgets, streams | `@property`, `@staticmethod`, `@lru_cache` |

Can we use Decorator feature for Decorator pattern?

# Using Decorator feature for Decorator Pattern

```python
def loud_decorator(cls):
    original_make_sound = cls.make_sound

    def new_make_sound(self):
        sound = original_make_sound(self)
        return sound.upper() + "!!!"

    cls.make_sound = new_make_sound
    return cls

def whisper_decorator(cls):
    original_make_sound = cls.make_sound

    def new_make_sound(self):
        sound = original_make_sound(self)
        return sound.lower() + "..."

    cls.make_sound = new_make_sound
    return cls
```

```python
@whisper_decorator
@loud_decorator
class Dog:
    def make_sound(self):
        return "Woof"

dog = Dog()
print(dog.make_sound())  #
```

???

```python
@loud_decorator
@whisper_decorator
class Dog:
    def make_sound(self):
        return "Woof"

dog = Dog()
print(dog.make_sound())
```

# Will decorating always look good?

# Cautionary comments for Decorator *pattern*

| When | Why It's Bad |
| --- | --- |
| Too many decorator layers | Hard to trace, maintain, and debug (`Decorator Hell`) |
| Inheritance is clearer/simpler | Simpler than wrapping objects unnecessarily |
| Performance is critical | Decorator overhead adds indirection and latency |
| Behavior order is tricky or unclear | Hard to understand the interaction of behaviors |
| Mutually exclusive behaviors exist | Allows illogical combinations without constraint |

# Cautionary comments for Decorator as a Python *feature*

| When | Why |
|---|---|
| Code becomes **hard to read or trace** | Wrapping hides behavior |
| You need **per-instance** behavior | Decorators are static at definition time |
| Debugging gets complicated | Decorators can obscure function identity |
| Performance is critical | Decorators add overhead |
| Behavior needs **dynamic control** | Decorators are static unless configured |
| Simple logic doesn't justify extra complexity | Decorators can be overkill |

# Object Oriented Programming

## Structural Design Patterns: Composite

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# Why should we think about Composite?

```python
class Animal:
    def make_sound(self):
        raise NotImplementedError

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")
```

*What if we need two ways of making them talk - individually and as a group?*

```python
def shout(animal_or_group):
    if isinstance(animal_or_group, list):
        for animal in animal_or_group:
            animal.make_sound()
    else:
        animal_or_group.make_sound()
```

Type may change!

*Repetitive check, fragile!*

# Can we treat a group (composite) as an individual?

*Group as an Individual animal*

```
def shout_group(anim
    for a in animals
        a.make_sound
```

```
def shout(animal: Animal):
    animal.make_sound()

# Usage
dog = Dog()
cat = Cat()

group = AnimalGroup()
group.add(dog)
group.add(cat)

# Works for both single animals and groups
shout(dog)    # → Woof!
shout(group)  # → Woof! Meow!
```

```
imal):
f):
: list[Animal] = []

imal: Animal):
.append(animal)

elf):
n self.members:
ake_sound()
```

# Can we make groups and then group them all? (nested)

```python
from abc import ABC, abstractmethod

# Component
class Animal(ABC):
    @abstractmethod
    def feed(self):
        pass

# Leaf
class Tiger(Animal):
    def feed(self):
        print("Feeding the tiger some me

class Parrot(Animal):
    def feed(self):
        print("Feeding the parrot some s
```

```python
tiger1 = Tiger()
tiger2 = Tiger()
parrot1 = Parrot()

big_cats = AnimalGroup("Big Cats")
big_cats.add(tiger1)
big_cats.add(tiger2)

birds = AnimalGroup("Birds")
birds.add(parrot1)

zoo = AnimalGroup("Whole Zoo")
zoo.add(big_cats)
zoo.add(birds)

# Single call to feed the entire zoo
zoo.feed()
```

```python
mal):
, name: str):
ame
 list[Animal] = []

mal: Animal):
append(animal)

ng group: {self.name}")
 self.members:
ed()
```

*Opens up nesting - why??*

# So how exactly Composite Pattern is different?

| Pattern | Primary Intent | Problem Solved | Key Benefit |
|---|---|---|---|
| Adapter | Convert one interface into another | Make incompatible interfaces work together | Reuse existing code without modifying it |
| Facade | Simplify a complex subsystem | Provide a single, easy-to-use API | Reduce coupling between clients and subsystems |
| Decorator | Add responsibilities dynamically | Extend object behavior without subclassing | Flexible, runtime behavior augmentation |
| **Composite** | Compose objects into **tree structures** | Treat individual objects and object groups uniformly | Simplify client code by handling leaf and composite nodes identically |

# Does grouping always lead to Composite?

# Object Oriented Programming

## Behavioral Design Patterns: Observer

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# What if objects want to notify each other?

```
class Sparrow:
    def alert(self, predator):
        print(f"Sparrow: Alert! A {predator}

class Robin:
    def alert(
        print(

class BlueJay:
    def alert(
        print(f"BlueJay: Heads up! There's a
```

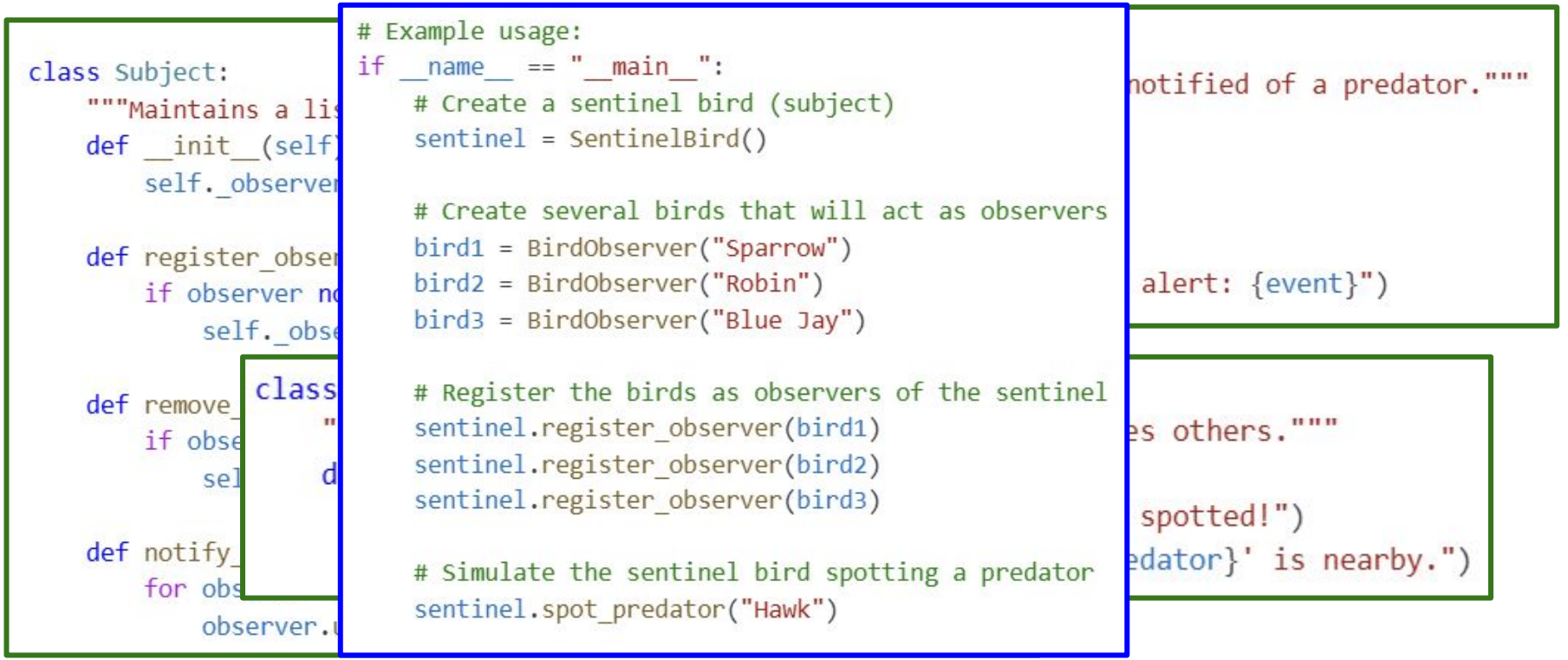
```
# Example usage:
sentinel = SentinelBird()
sentinel.spot_predator("hawk")
```

```
class SentinelBird:
    def __init__(self):
        self.sparrow = Sparrow()
        robin = Robin()
        bluejay = BlueJay()

    predator(self, predator):
    (f"Sentinel Bird: Spotted a {predator}!")
        sparrow.alert(predator)
        robin.alert(predator)
        self.bluejay.alert(predator)
```

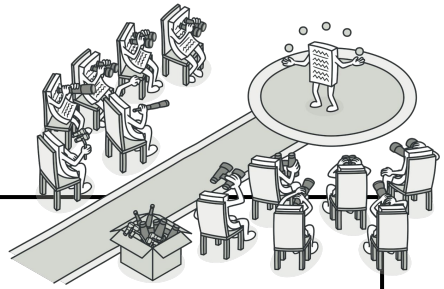
# What's wrong with the design?

| Disadvantage | Explanation |
|---|---|
| **Tight Coupling** | The `SentinelBird` directly references specific bird classes (e.g., `Sparrow`, `Robin`)<br>→ changes to one require modifying the subject class |
| **Scalability Issues** | `SentinelBird` becomes overloaded with direct dependencies as more birds add in |
| **Reduced Flexibility** | Dynamic addition or removal of bird observers at runtime not possible |

# What if objects want to notify each other? The Observer way!!



# Key Takeaways



| Advantage | Explanation |
|---|---|
| **Loose Coupling** | Decouples the subject from its observers, reducing dependencies → allows changes without widespread impact |
| **Dynamic Relationships** | Observers can be added or removed at runtime → scalable |
| **Improved Maintainability** | Separates the notification mechanism from the business logic → easier to update or extend parts of the system without affecting others |
| **Reusability** | Can be reused across different subjects or projects, reducing code duplication and promoting modular design |

# Synchronous vs. Asynchronous Notifications

# The (default) synchronous notification - Waiting is not good!

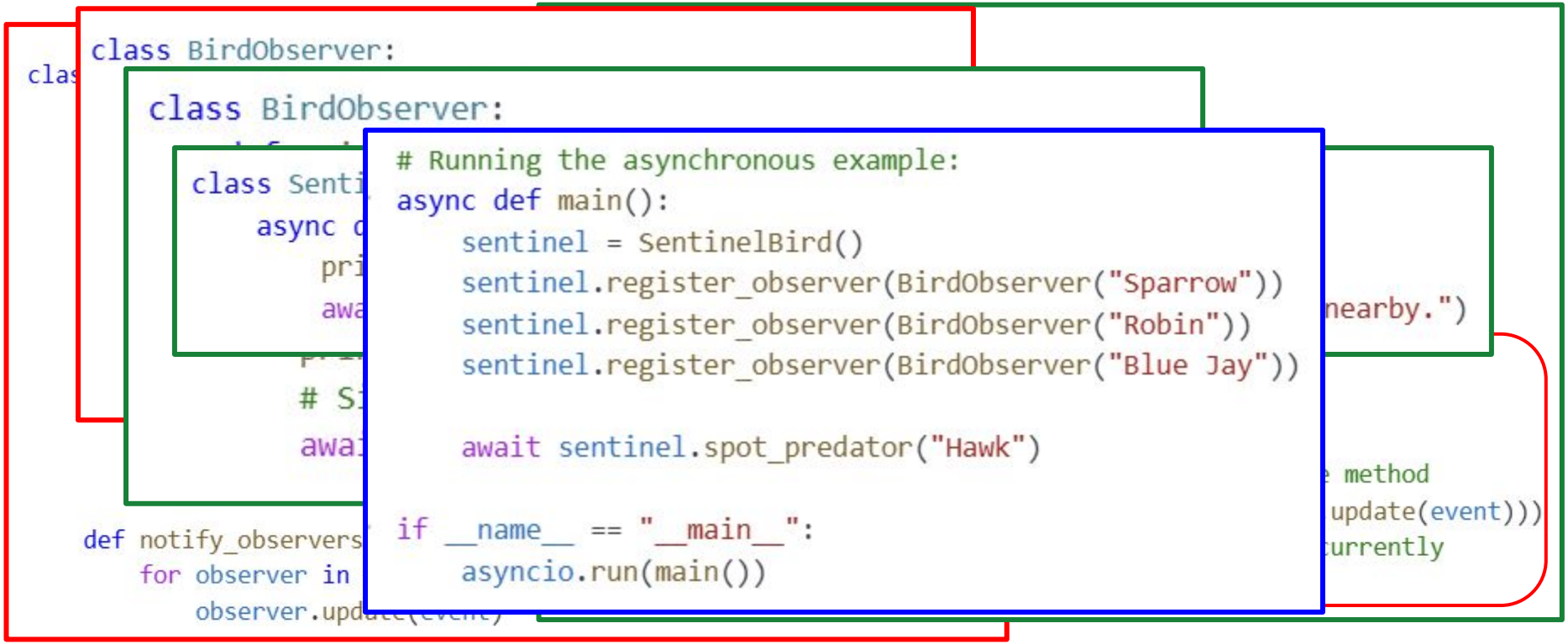
```
class BirdObserver:
    """An observer that reacts when notified of a predator."""
    def __i
        sel

    def upd
        pri
```

```
# Running the synchronous example:
if __name__ == "__main__":
    sentinel = SentinelBird()
    sentinel.register_observer(BirdObserver("Sparrow"))
    sentinel.register_observer(BirdObserver("Robin"))
    sentinel.register_observer(BirdObserver("Blue Jay"))

    sentinel.spot_predator("Hawk")
```

```
vent}")
```

*Sentinel Bird must wait for each bird to finish processing its alert before moving on to the next.*

# Notify each other *without having to wait* for anyone! Async



# Object Oriented Programming

## Behavioral Design Patterns: Strategy

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# What if objects want to share each others' behaviors *runtime*?

```python
from abc import ABC, abstractmethod

# Define the MovementStrategy interface
class MovementStrategy(ABC):
    @abstractmethod
    def move(self):
        pass
```

*Movements (walk, fly, swim) is no single animal's unique behavior*

```python
# Concrete strategy for walking
class WalkStrategy(MovementStrategy):
    def move(self):
        print("walking on land.")

# Concrete strategy for flying
class FlyStrategy(MovementStrategy):
    def move(self):
        print("flying in the sky.")

# Concrete strategy for swimming
class SwimStrategy(MovementStrategy):
    def move(self):
        print("swimming in water.")
```

# What if objects want to share each others' behaviors *runtime*?

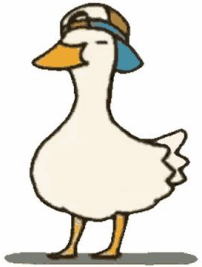
```python
# Concrete strategy for wal
class WalkStrategy(Movement
    def move(self):
        print("walking on l

# Concrete strategy for fly
class FlyStrategy(MovementS
    def move(self):
        print("flying in th

# Concrete strategy for swi
class SwimStrategy(Movement
    def move(self):
        print("swimming in
```

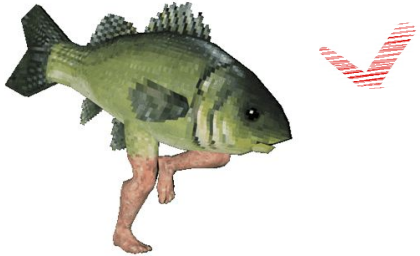
```python
# Animal class that uses a movement strategy
class Animal:
    def __init__(self, name, movement_strategy: MovementStrategy):
        self.name = name
        self.movement_strategy = movement_strategy

    def perform_move(self):
        # Delegate the movement behavior to the strategy
        print(f"{self.name} is ", end="")
        self.movement_strategy.move()

    def set_movement_strategy(self, new_strategy: MovementStrategy):
        # Allows changing the movement behavior at runtime
        self.movement_strategy = new_strategy
```

*Keeping it generic*

# What if objects want to share each others' behaviors *runtime*?

```
# Example usage
if __name__ == '__main__':
    # A duck that initially can fly
    duck = Animal("Duck", FlyStrategy())
    duck.perform_move()  # Output: Duck is flying in the sky.

    # Change the duck's movement strategy at runtime
    duck.set_movement_strategy(WalkStrategy())
    duck.perform_move()  # Output: Duck is walking on land.

    # A fish that swims
    fish = Animal("Fish", SwimStrategy())
    fish.perform_move()  # Output: Fish is swimming in water.
```





# Separating Object from Behaviors

# Wait! We added behavior earlier as well? Decorator!

## (Remember the dogs barking and eating??)

```python
# Base Decorator that ex
class AnimalDecorator(An
    def __init__(self, a
        self._animal = a

    def make_sound(self)
        return self._ani

    def __getattr__(self
        return getattr(s
```

```python
# MakeSoundDecorator that accepts a modality and applies it
class MakeSoundDecorator(AnimalDecorator):
    def __init__(self, animal: Animal, modality: str):
        super().__init__(anima
        self.modality = modali

    def make_sound(self):
        sound = self._animal.m

        if self.modality == "l
            return sound.upper
        elif self.modality ==
            return sound.lower
        else:
            raise ValueError(f

    def __getattr__(self, name
        return getattr(self._a
```

```python
# EatDecorator independently adds an 'eat' behavior
class EatDecorator(AnimalDecorator):
    def eat(self, food):
        base_animal = self._animal
        while hasattr(base_animal, '_animal'):
            base_animal = base_animal._animal

        class_name = base_animal.__class__.__name__
        return f"{class_name} is going to eat {food}"

    def __getattr__(self, name):
        return getattr(self._animal, name)
```

# Wait! We added behavior earlier as well? Decorator!

(Remember the dogs barking and eating??)

```
# ========================
# ✅ Client Code: Eating Loud vs Loud Eating
# ========================
if __name__ == "__main__":
    dog = Dog()

    print("\n=== Eating Loud Dog ===")
    eating_loud_dog = MakeSoundDecorator(EatDecorator(dog), "loud")
    print(eating_loud_dog.make_sound())  # WOOF!!!
    print(eating_loud_dog.eat("kibble")) # Dog is going to eat kibble

    print("\n=== Loud Eating Dog ===")
    loud_eating_dog = EatDecorator(MakeSoundDecorator(dog, "loud"))
    print(loud_eating_dog.make_sound())  # WOOF!!!
    print(loud_eating_dog.eat("kibble")) # Dog is going to eat kibble
```

# Then why not the Decorator way?

| Aspect | Decorator Pattern | Strategy Pattern |
|---|---|---|
| **Purpose** | **Add** or **augment** *functionalities* to objects dynamically without modifying their structure | Family of *algorithms* → **encapsulate** each → make them interchangeable<br><br>Algorithm can vary independently from clients |
| **Use Case** | When you need to attach additional responsibilities to an object dynamically and transparently. | When you have multiple algorithms for a specific task and want to *switch* between them at runtime. |
| **Structure** | Utilizes composition to **wrap** objects with new behavior | Utilizes composition to **delegate** specific behaviors to different strategy classes |

# Then why not the Decorator way?

| Aspect | Decorator Pattern | Strategy Pattern |
|---|---|---|
| **Flexibility** | Allows **combining** multiple behaviors by stacking decorators | Allows **changing** the behavior of a class by switching strategies (typically one strategy at a time) |
| **Example Scenario** | Adding scrolling, borders, or drop shadows to GUI | Implementing different sorting algorithms that can be selected based on the dataset. |
| ***Relation to Object*** | *Enhances the object by wrapping it, effectively changing its "skin".* | *Alters the object's behavior by changing its "brain", i.e., the underlying algorithm.* |

# Key takeaways

| Advantages | Explanation |
|---|---|
| **Variable Behavior** | Object's behavior is likely to change or vary *in different contexts* (e.g., a duck that can both fly and walk)<br>→ encapsulating behaviors separately allows easy *swapping* or *modification* |
| **Multiple Implementations** | Several ways to perform a specific action (e.g., different movement types: flying, walking, swimming)<br>→ reuse and clear organization |

# Key takeaways

| Advantages | Explanation |
| --- | --- |
| **Separation of Concerns** | Keeping the core class focused on its primary responsibilities while delegating secondary behaviors (like movement) to separate components makes the code easier to understand, maintain, and test. |
| **Adherence to SOLID Principles** | Specifically, the open/closed principle is supported—your core class (e.g., Animal) is closed for modification but open for extension through new behaviors without modifying its code. |
| **Dynamic Behavior Change** | If the behavior needs to change at runtime based on different conditions (like an animal adapting its movement in response to environmental changes), separating it allows for dynamic switching of algorithms. |

# Object Oriented Programming

## Behavioral Design Patterns: Command

IT620


**Sourish Dasgupta**
Assistant Professor, DA-IICT, Gandhinagar
*https://daiict.ac.in/faculty/sourish-dasgupta*

# What if objects want to talk to each other via "*messengers*"?

```python
# Receiver: Basic Animal class
class Animal:
    def speak(self):
        print("Animal makes a generic sound.")

    def run(self):
        print("Animal is running.")

# Receiver: Concrete Animal - Dog
class Dog(Animal):
    def speak(self):
        print("Woof!")

    def run(self):
        print("Dog runs energetically.")
```

```python
from abc import ABC, abstractmethod

# Command Interface
class Command(ABC):
    @abstractmethod
    def
```

```python
# Concrete Command to make the animal speak
class MakeSoundCommand(Command):
    def __init__(self, animal: Animal):
        self.animal = animal

    def execute(self):
        self.animal.speak()

# Concrete Command to make the animal run
class RunCommand(Command):
    def __init__(self, animal: Animal):
        self.animal = animal

    def execute(self):
        self.animal.run()
```

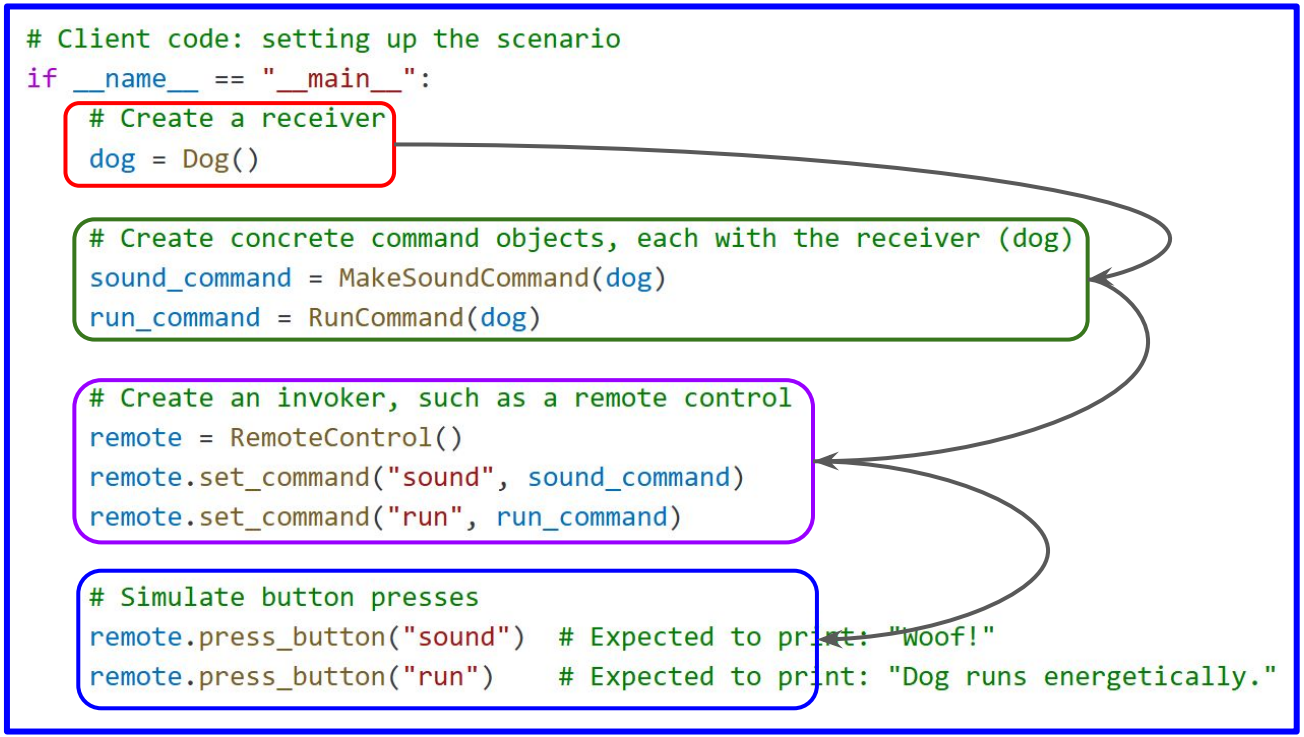
# What if objects want to talk to each other via "*messengers*"?

```python
from abc import ABC, abstractmethod

# Command Interface
class Command(ABC):
    @
    d
```

```python
# Concrete Command to make the animal speak
class MakeSoundCommand(Command):
    def __init__(self, animal: Animal):
        self.animal = animal

    def execute(self):
        self.animal.speak()

# Concrete Command to make the animal run
class RunCommand(Command):
    def __init__(self, animal: Animal):
        self.animal = animal

    def execute(self):
        self.animal.run()
```

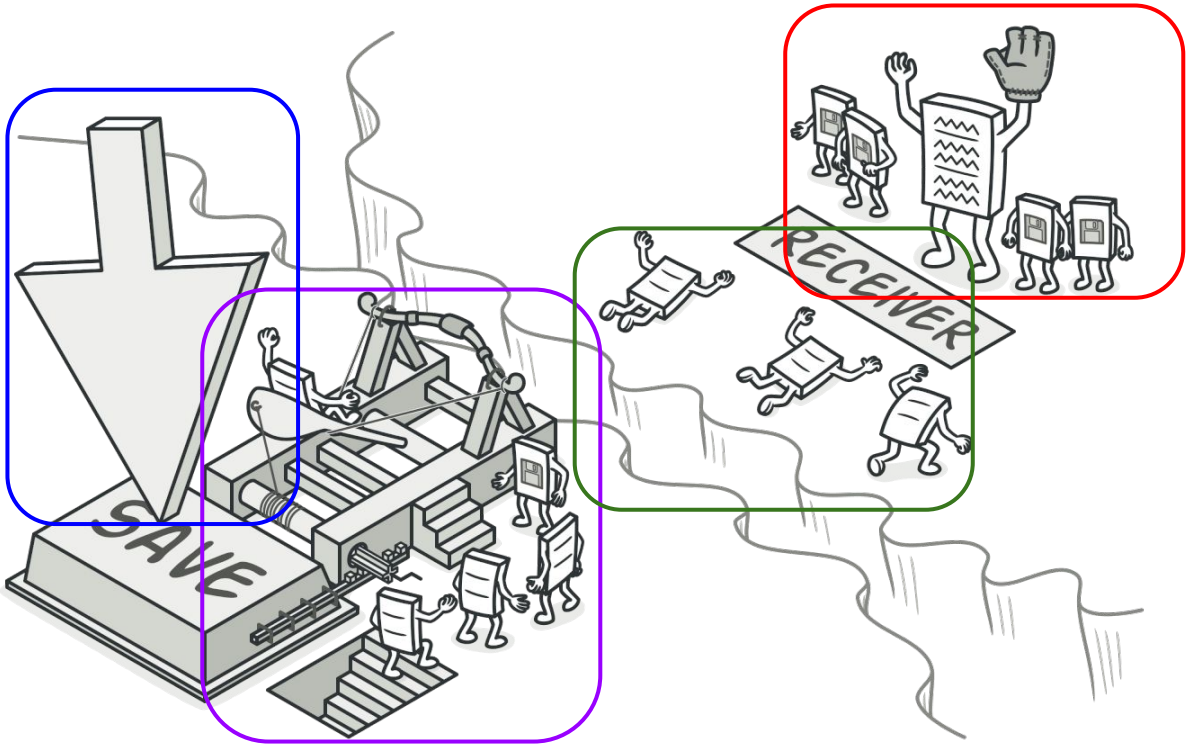
```python
# Invoker: A RemoteControl that triggers commands
class RemoteControl:
    def __init__(self):
        # A dictionary mapping command names to command objects.
        self.commands = {}

    def set_command(self, name: str, command: Command):
        """Assign a command to a button name."""
        self.commands[name] = command

    def press_button(self, name: str):
        """Simulate pressing a button to execute a command."""
        if name in self.commands:
            print(f"Executing '{name}' command:")
            self.commands[name].execute()
        else:
            print(f"No command found for '{name}'")
```

# The *sender* → *messenger* → *receiver* paradigm



```
# Client code: setting up the scenario
if __name__ == "__main__":
    # Create a receiver
    dog = Dog()

    # Create concrete command objects, each with the receiver (dog)
    sound_command = MakeSoundCommand(dog)
    run_command = RunCommand(dog)

    # Create an invoker, such as a remote control
    remote = RemoteControl()
    remote.set_command("sound", sound_command)
    remote.set_command("run", run_command)

    # Simulate button presses
    remote.press_button("sound")  # Expected to print: "Woof!"
    remote.press_button("run")    # Expected to print: "Dog runs energetically."
```

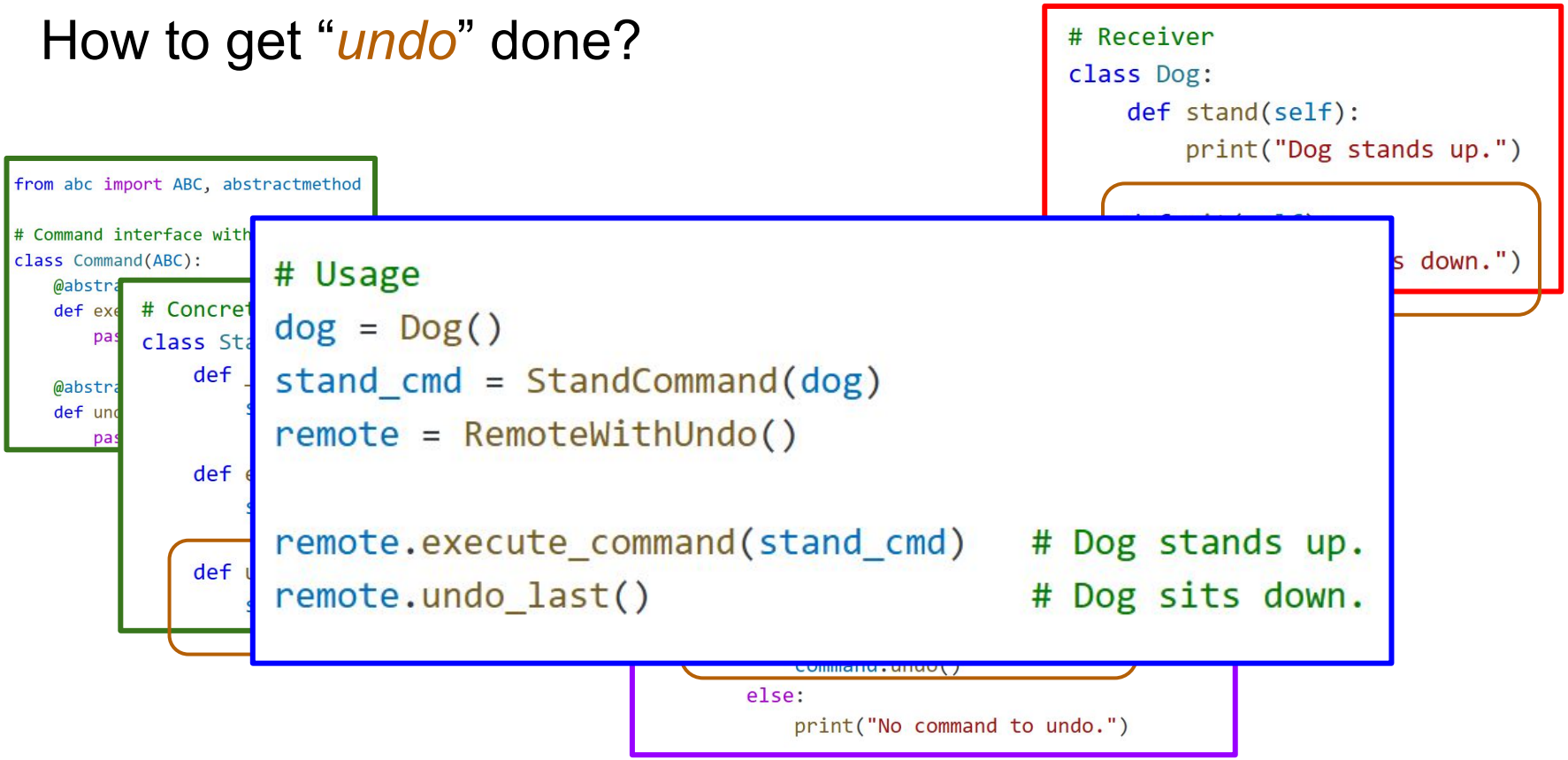
# The *sender* → *messenger* → *receiver* paradigm



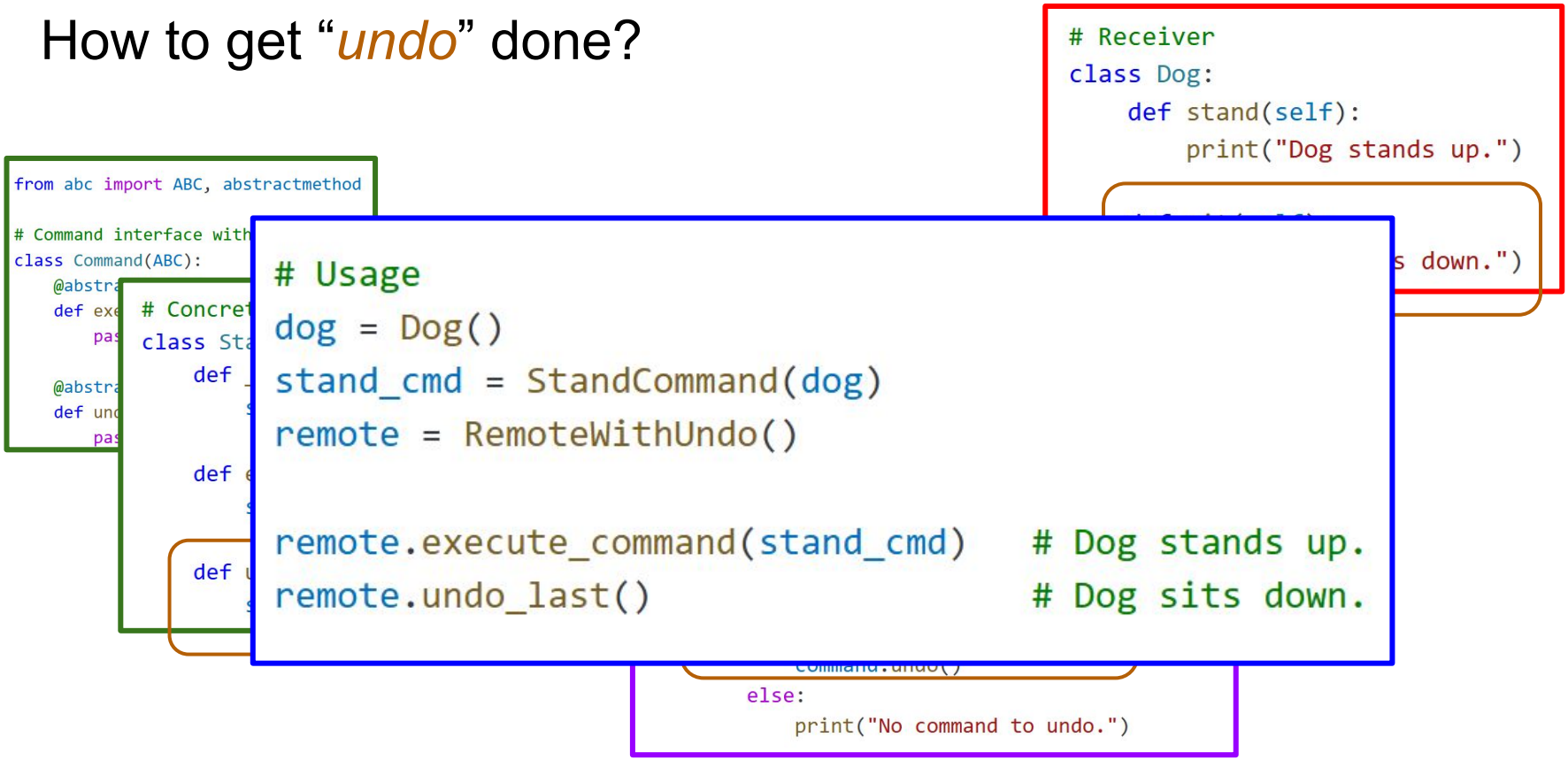
# Why Command?

| Benefit | Description |
|---|---|
| **Decoupling** | *Separates* the object that invokes the command (Invoker) from the one that knows how to execute it (Receiver) |
| **Extensibility (S*O*LID)** | New commands can be *added* without changing existing code (open/closed principle). |
| **Reusability** | Commands can be *reused* across different invokers or contexts. |
| **Undo/Redo Support** | Commands can store state for undo/redo functionality |
| **Command Queuing/Logging** | Commands can be stored, logged, or *executed later* (e.g., task queues) |
| **Macro Commands** | Multiple commands can be *combined* into one command to execute in sequence. |
| **Flexible Invocation** | Commands can be parameterized, passed as arguments, or triggered in various ways |

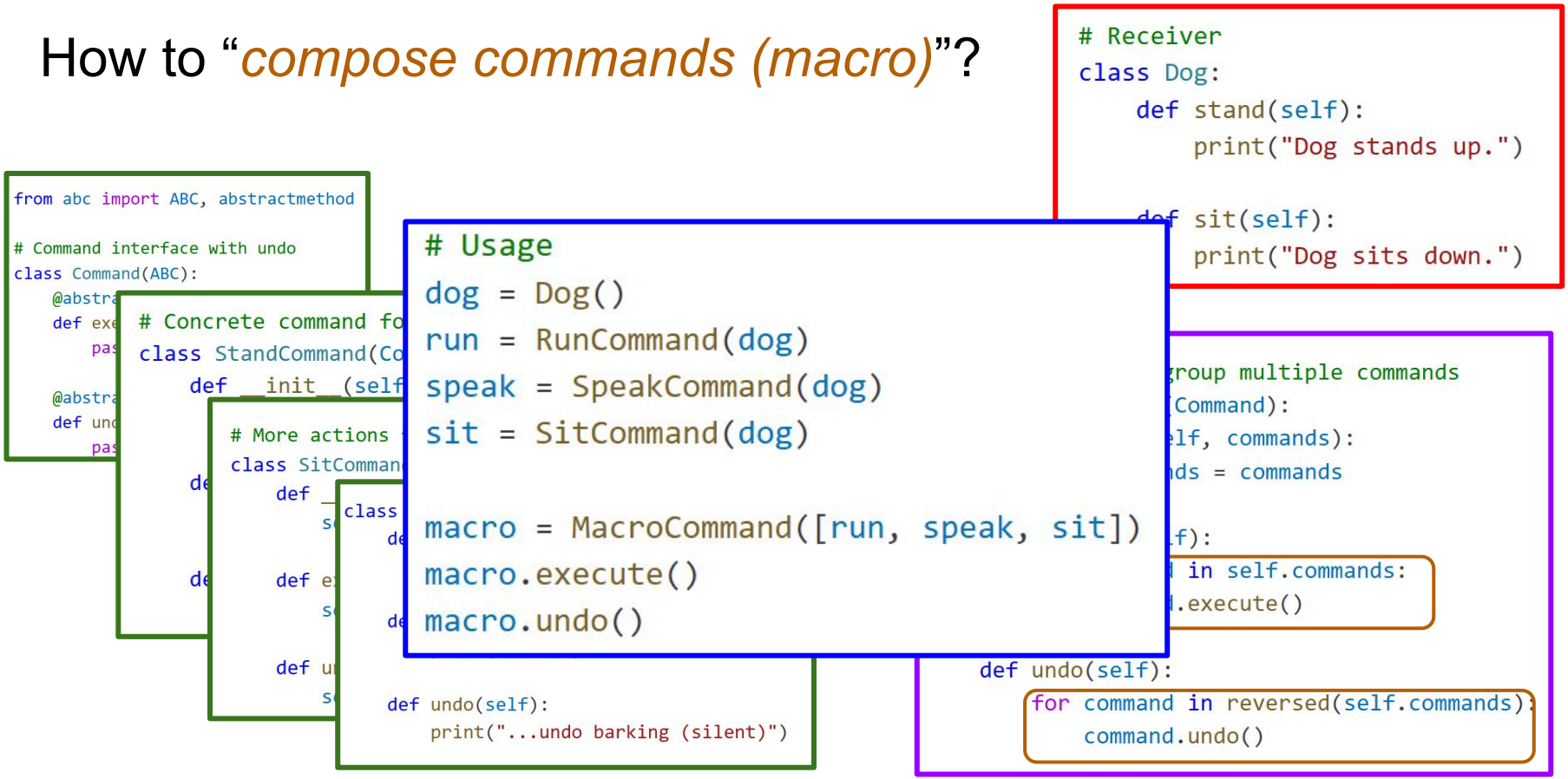
# How to get “*undo*” done?



# How to get “*undo*” done?

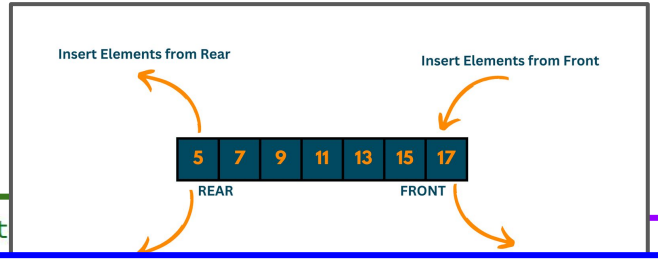


# How to "*compose commands (macro)*"?



# How to “*queue up commands*”?

```
# Receiver
class Dog:
    def stand(self):
        print("Dog stands up.")

    def sit(self):
        print("Dog sits down.")
```




```
from abc import ABC, abstractmethod

# Command interface with undo
class Command(ABC):
    @abstra
    def exe
        pas

    @abstra
    def und
        pas
```

```
# Concrete command for st
class StandCommand(Com
    def __init__(self,
        s

    def
        s

    def
        s
```

```
# More actions fo
class SitCommand(
    def __
        s

    def e
        s

    def u
        s
```

```
class S
    def

    def

    def
```

```
# Usage
dog = Dog()
queue = CommandQueue()
queue.add_command(StandCommand(dog))
queue.add_command(SpeakCommand(dog))
queue.add_command(SitCommand(dog))

# All commands are executed in order later
queue.run()
```

```
ctions import deque

andQueue:
init__(self):
lf.queue = deque()

d_command(self, command: Command):
lf.queue.append(command)

n(self):
ile self.queue:
    command = self.queue.popleft()
    command.execute()
```