Session 4 : OOP Part 2

Md Abdullah All Naim

Software Engineer Level II
Associate Program Manager @ CodeCamp by Astha.IT

Table of Contents

🧩 Abstraction, Polymorphism

⚡ Compile-time Polymorphism (Method Overloading)

🚀 Runtime Polymorphism (Virtual/Override)

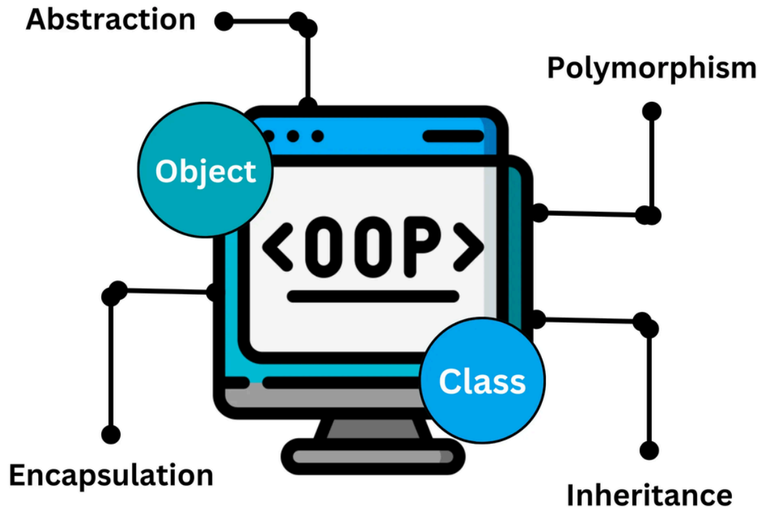
📐 Abstract Classes & Methods

📜 Interfaces: Definition, Implementation & Explicit Usage

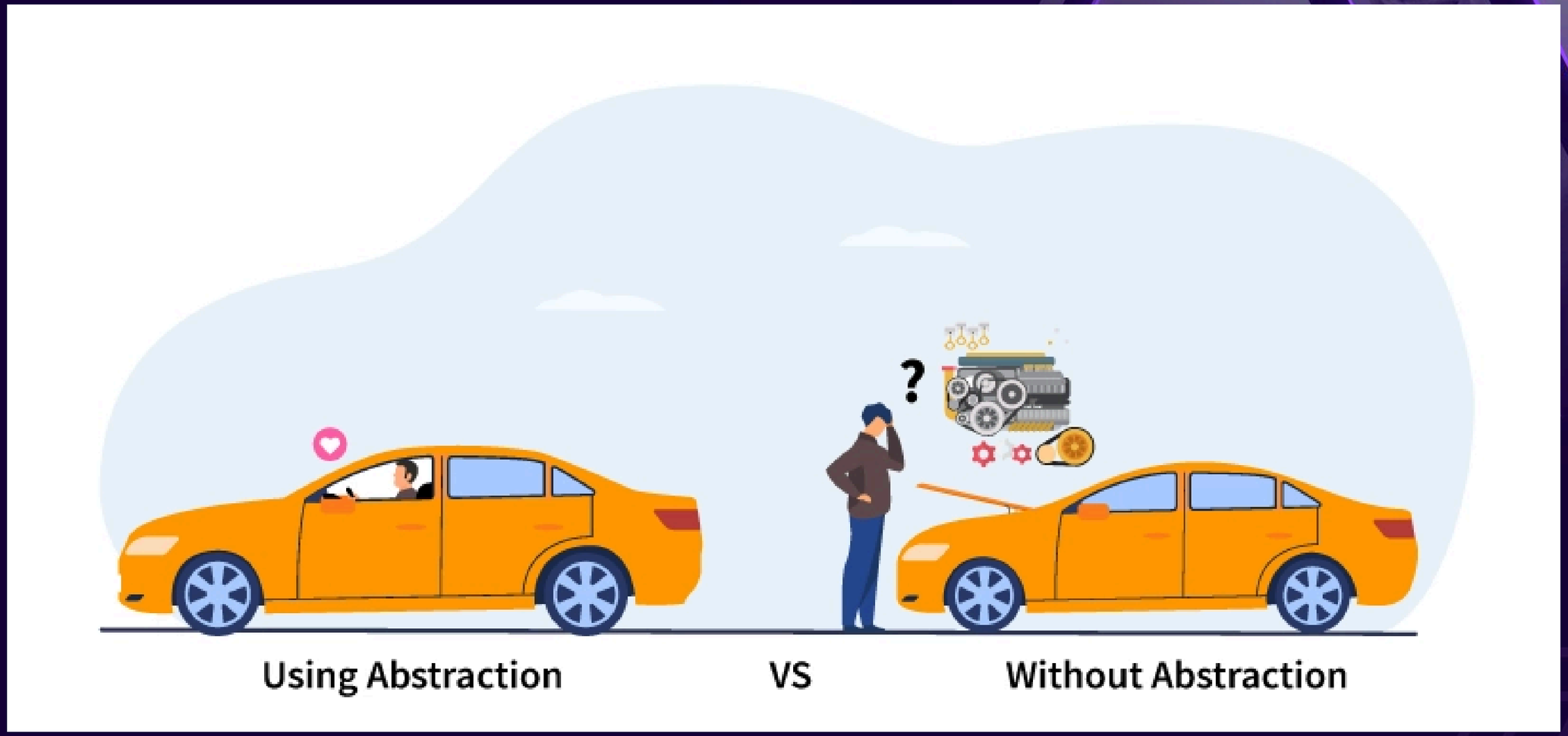
🛠️ Interface Segregation Principle & Default Methods (C# 8.0+)

⚖️ Interfaces vs Abstract Classes

🔗 Association, Aggregation & Composition



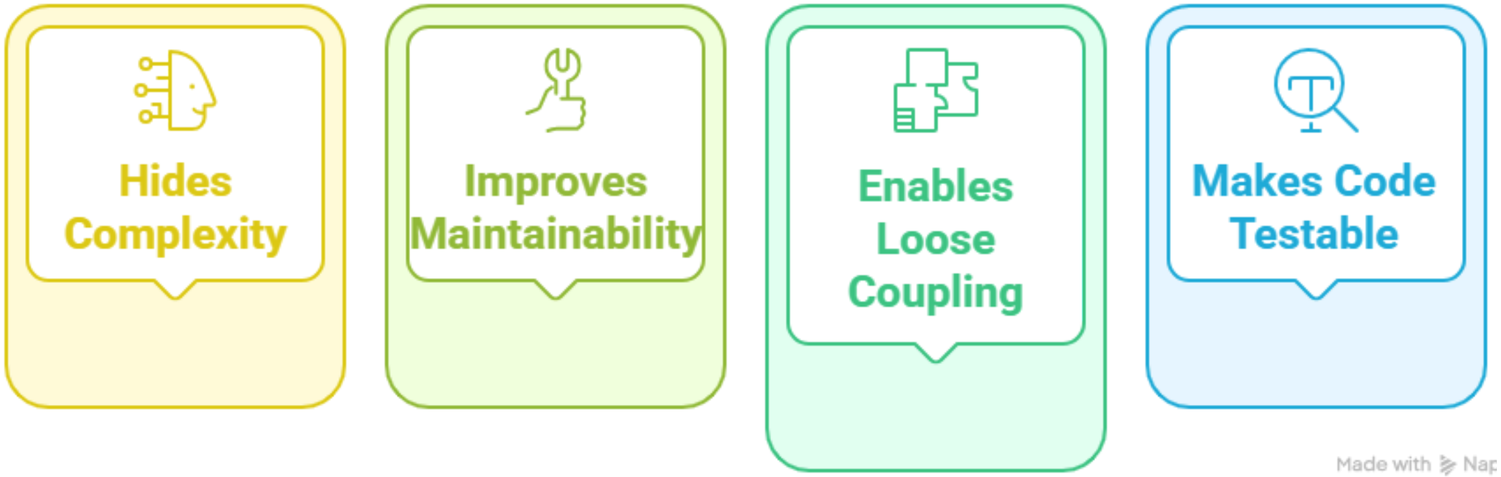
Abstraction



ABSTRACTION

Abstraction is the process of **hiding complex implementation** details and showing only the essential features of an object.

It allows developers to work with what an **object does, rather than how it does it.**

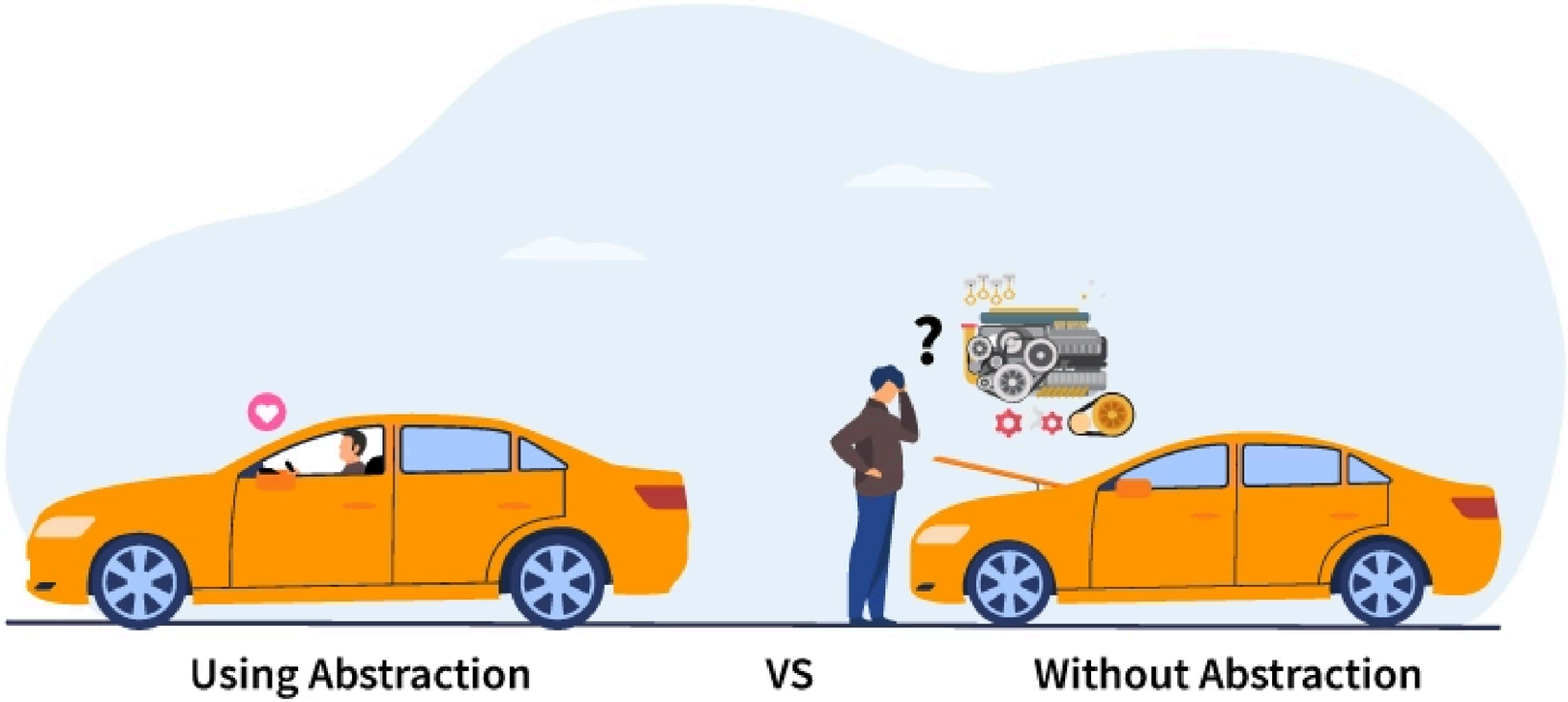


ABSTRACTION

✅ Ways to Achieve Abstraction in C#

Using Abstract Classes

Using Interfaces



ABSTRACTION

An abstract class:

An abstract class is a class that **cannot be instantiated** on its own and is designed to be a **base class for other classes**.

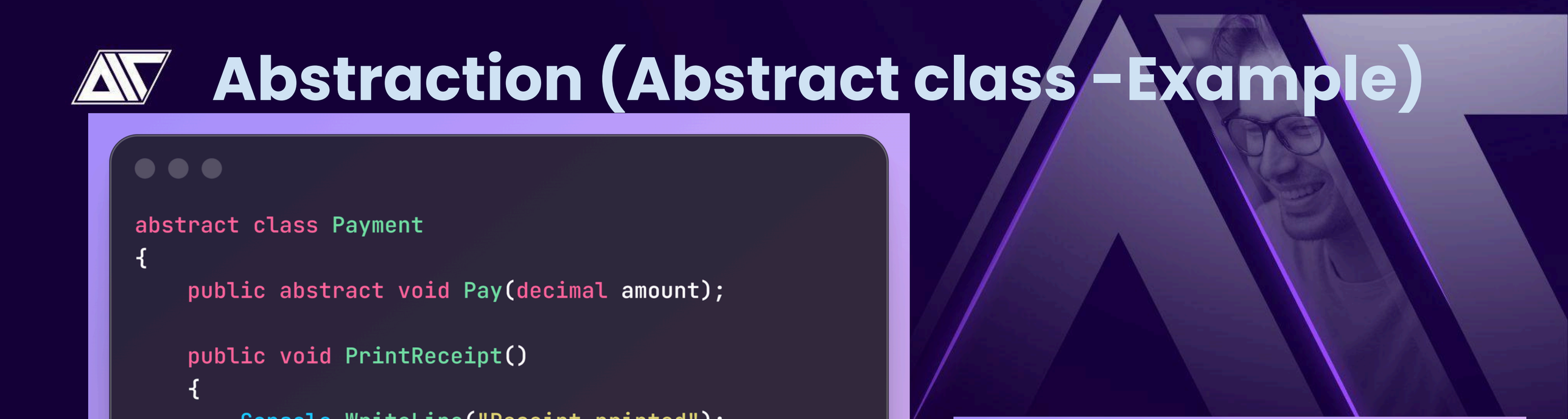
It can contain:

- **Abstract members** → methods or properties that **must be implemented** in derived classes
- **Concrete members** → normal methods or properties that **can be used as-is (optional)**

- Cannot be **instantiated** (you can't create objects of it)
- Can contain abstract **methods** (without implementation)
- Can also contain concrete methods (with implementation)
- Is meant to be **inherited**

Abstraction (Abstract class -Example)

```
abstract class Payment
{
    public abstract void Pay(decimal amount);

    public void PrintReceipt()
    {
        Console.WriteLine("Receipt printed");
    }
}
class CardPayment : Payment
{
    public override void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount} using card");
    }
}
Payment payment = new CardPayment();
payment.Pay(500);
payment.PrintReceipt();
```



- ✔ User only knows Pay(), not card logic
- ✔ Implementation is hidden

See Example 1

ABSTRACTION

An interface is a **contract** that defines what **a class must do**, but **not how it does it.**

It contains method, property, event, or indexer declarations without implementation (**implementation is provided by the class that implements it**).

Interfaces help to:

- Achieve **abstraction**
- Key Points
- No implementation (except default methods in C# 8.0+)
- Cannot have fields
- **Multiple interfaces** can be implemented (unlike inheritance from a single class)
- Helps in **decoupling high-level code** from low-level implementations
- Great for **polymorphism**

ABSTRACTION

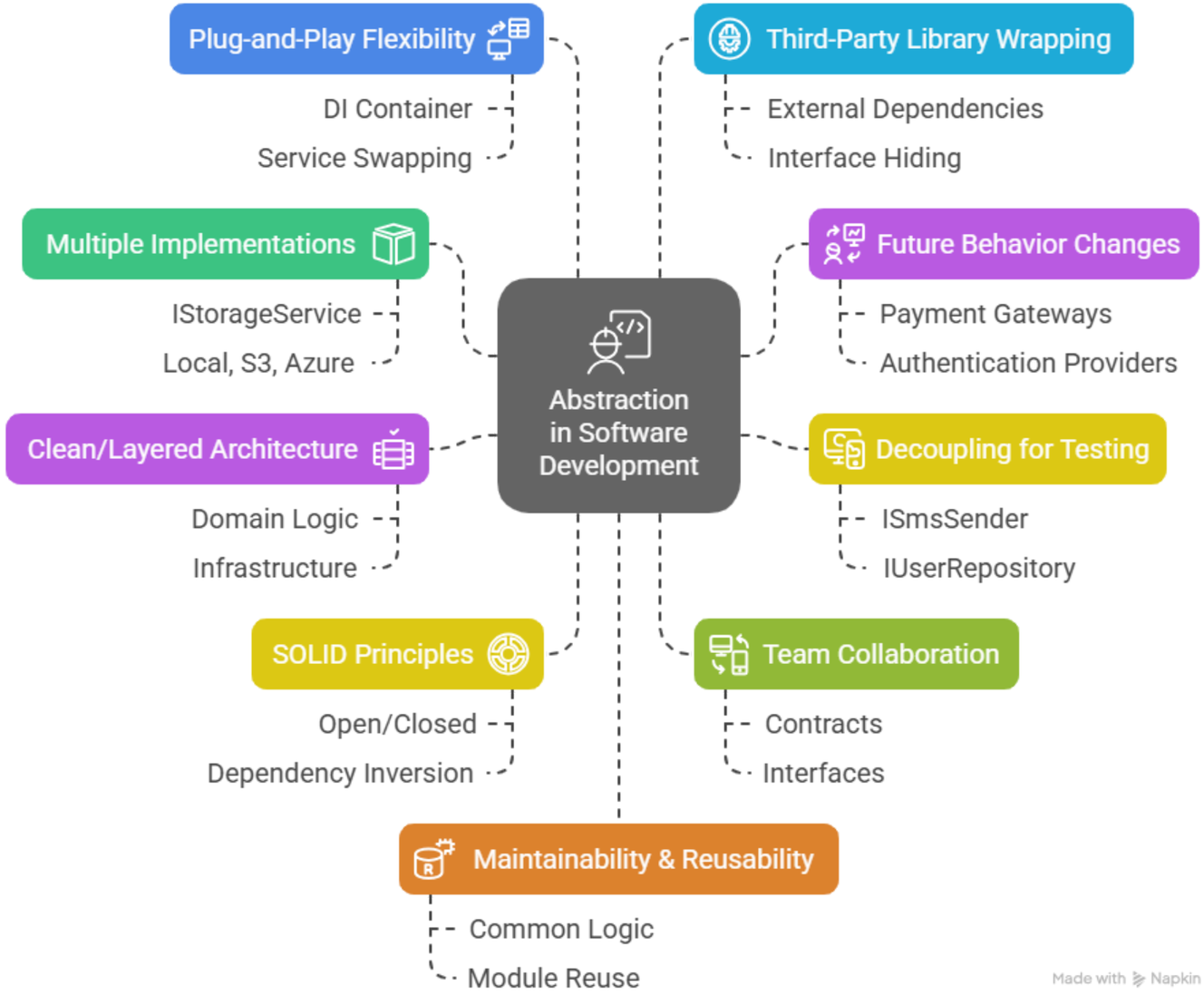
Use Interface when:

- You need **multiple** inheritance
- You're defining a contract **without behavior**
- You expect **unrelated classes** to implement it
- You only need to **define capabilities (what a class can do), not behavior**.

Use Abstract Class when:

- You need **constructors**, fields, or access modifiers
- Classes are related (inherit from the same base).
- You want to share common behavior (concrete methods, fields, properties).
- You want to **force certain methods to be implemented** in derived classes.

WHEN ABSTRACTION IS JUSTIFIED



WHEN ABSTRACTION IS JUSTIFIED

Abstraction is justified when it reduces complexity, improves flexibility, and protects your system from change.

You should not use abstraction "**just because OOP exists**" — **it must solve a real problem.**

When You Want to Hide Complex Implementation

If the internal logic is complex and users don't need to know it.

```
interface IPayment
{
    void Pay(decimal amount);
}
```

The user only calls Pay() —
validation, API calls, encryption are hidden.

✓ Justified because it simplifies usage

WHEN ABSTRACTION IS JUSTIFIED

When Behavior Can Have Multiple Implementations

If one behavior can be **implemented in different ways.**

Example

- Payment → Card, bKash, Cash
- Notification → Email, SMS, Push
- Storage → File, Database, Cloud

```
interface INotification
{
    void Send(string message);
}
```

✓ Justified because it supports change without modifying existing code

WHEN ABSTRACTION IS JUSTIFIED

When You Want Loose Coupling

When high-level code should not depend on concrete classes.

Boss কাজ করতে চায়, কিন্তু কোন Worker করবে সেটা Boss কে জানা দরকার নেই। Boss শুধু বলে “এই কাজ করো”।

Bad (tight coupling ❌)

```
class OrderService
{
    private CardPayment payment = new CardPayment();
}
```

Good (abstraction ✅)

```
class OrderService
{
    private readonly IPayment payment;

    public OrderService(IPayment payment)
    {
        this.payment = payment;
    }
}
```

✔ Justified because it follows **Dependency Inversion Principle (DIP)**

WHEN ABSTRACTION IS JUSTIFIED

When Future Changes Are Likely

If requirements may change or grow.

Example:

- Today → only CardPayment
- Tomorrow → bKash, Nagad, Stripe

Abstraction prepares your system for growth.

✓ **Justified because it avoids rewriting code later**

When Writing Testable Code

Mocks and fakes work only with abstractions.

```
IPayment mockPayment = new FakePayment();
```

✓ Justified because it enables **unit testing**

WHEN ABSTRACTION IS JUSTIFIED

When Defining System Boundaries

Used at layer boundaries:

- Controller → Service
- Service → Repository
- Application → External APIs

```
interface IUserRepository
{
    User GetById(int id);
}
```

✓ Justified because it isolates layers

When Multiple Teams or Modules Work Together

Interfaces act as contracts between teams.

✓ Backend team defines interface

✓ Implementation can be done independently

WHEN ABSTRACTION IS !JUSTIFIED

❌ **When Abstraction Is NOT Justified**

Avoid abstraction when:

❌ There is only **one implementation** and **no future plan**

❌ Logic is simple and **unlikely to change**

❌ You are writing **small scripts** or prototypes

❌ It adds unnecessary complexity (over-engineering)

IDENTIFY EXCESSIVE ABSTRACTION

- Too many layers for Simple Tasks
- Abstracting things that won't change
- Code is full of **Empty-Pass throughs**

```
Controller → Service → Interface → Adapter → Factory → Handler → ...
```

```
interface IDateTimeService
{
    DateTime Now();
}

class DateTimeService : IDateTimeService
{
    public DateTime Now() => DateTime.Now;
}
```

```
class OrderService
{
    private readonly OrderRepository _repo;

    public OrderService(OrderRepository repo)
    {
        _repo = repo;
    }

    public void CreateOrder(Order order)
    {
        _repo.Create(order); // only passing the call
    }
}
```

IDENTIFY EXCESSIVE ABSTRACTION

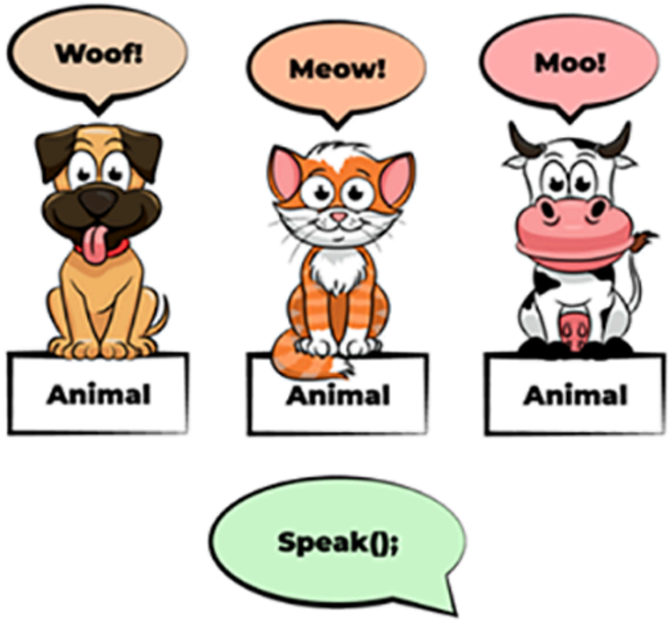
Empty-Pass throughs

```
class OrderService
{
    private readonly OrderRepository _repo;

    public OrderService(OrderRepository repo)
    {
        _repo = repo;
    }

    public void CreateOrder(Order order)
    {
        _repo.Create(order); // only passing the call
    }
}
```

Good Example

```
public void CreateOrder(Order order)
{
    if(order.Total <= 0)
        throw new Exception("Invalid order");

    order.CreatedAt = DateTime.Now;

    _repo.Create(order);
}
```

Polymorphism



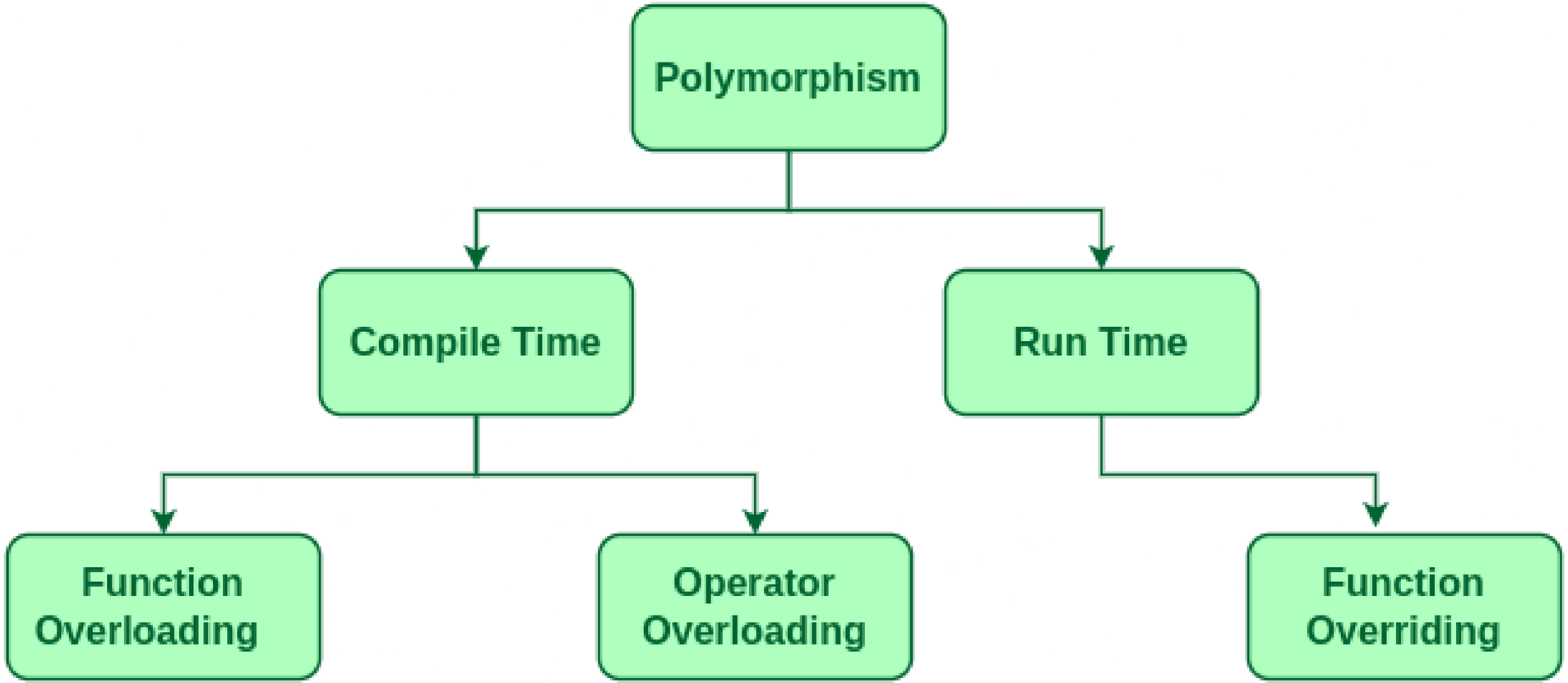
POLYMORPHISM

Polymorphism allows a **single interface** to be used for different underlying data types, **enabling one method or action** to **behave differently** based on the object it is acting upon.

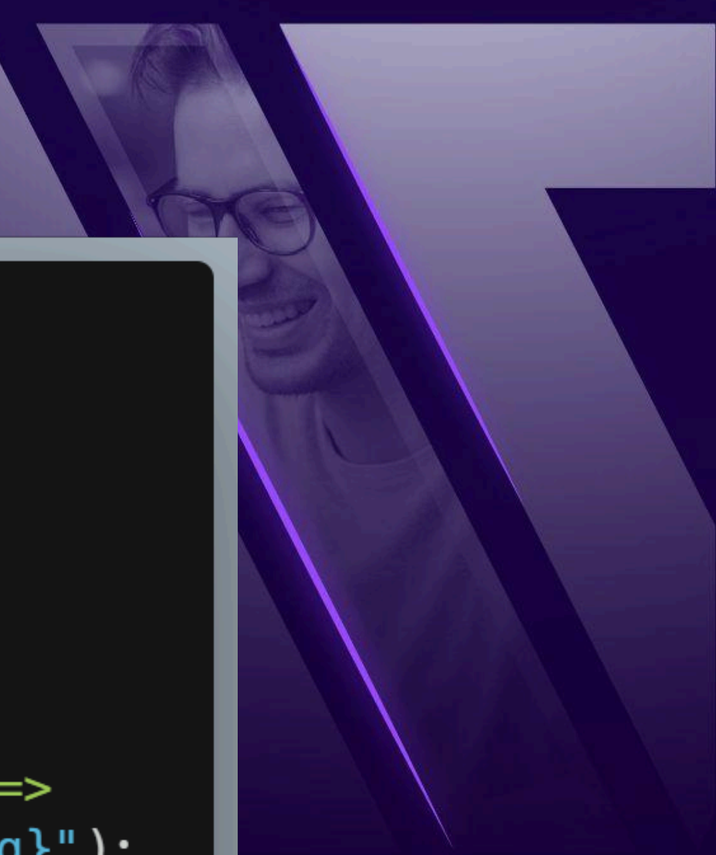
Polymorphism is what allows us to write flexible and extensible code — we can switch out implementations without changing the code that uses them.

In real projects, it keeps our code clean, reduces conditionals, and makes it much easier to scale.

POLYMORPHISM



Polymorphism (Example)



```
public class Notifier
{
    public virtual void Send(string msg) =>
        Console.WriteLine($"Notify: {msg}");

    public void Send(string msg, string user) =>
        Console.WriteLine($"Notify {user}: {msg}");
}

public class EmailNotifier : Notifier
{
    public override void Send(string msg) =>
        Console.WriteLine($"Email: {msg}");
}
```

TYPES OF POLYMORPHISM

Compile-Time Polymorphism (Static Polymorphism)

Achieved by:

- **Method Overloading**
- **Operator Overloading**

✓ Decided at compile time

🔹 Method Overloading Example

```
class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

🔹 Operator Overloading Example

```
class Point
{
    public int X, Y;

    public static Point operator +(Point a, Point b)
    {
        return new Point { X = a.X + b.X, Y = a.Y + b.Y };
    }
}
```

TYPES OF POLYMORPHISM

Runtime Polymorphism (Dynamic Polymorphism)

Achieved by:

- **Method Overriding**
- **Interfaces**
- **Abstract Classes**

✓ Decided at **runtime**

✓ Uses inheritance or interfaces

🔹 Method Overriding Example

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal speaks");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}
```

```
interface IPayment
{
    void Pay();
}

class CardPayment : IPayment
{
    public void Pay()
    {
        Console.WriteLine("Pay by card");
    }
}

class CashPayment : IPayment
{
    public void Pay()
    {
        Console.WriteLine("Pay by cash");
    }
}
```

TYPES OF POLYMORPHISM

Parametric Polymorphism (Generics)

Achieved by:

- Generics

✓ **One code works for many data types**

```
class Box<T>
{
    public T Value { get; set; }
}
Box<int> intBox = new Box<int>();
Box<string> stringBox = new Box<string>();
```

POLYMORPHISM

Polymorphism Best Practices

Program to Interfaces, Not Concrete Classes

👉 Depend on interfaces, not concrete implementations.

❌ Bad Example (Tightly Coupled)

```
class OrderService
{
    private CardPayment payment = new CardPayment();
}
```

Problems:

- Changing **CardPayment requires modifying this class**
- Unit testing becomes difficult
- **High coupling**

✅ Good Example (Decoupled)

```
class OrderService
{
    private readonly IPayment _payment;

    public OrderService(IPayment payment)
    {
        _payment = payment;
    }
}
```

POLYMORPHISM

Replace Long if-else / switch with Polymorphism

👉 Delegate behavior instead of using conditional logic.

❌ Bad Approach

```
if (paymentType == "Card")
    PayByCard();
else if (paymentType == "Bkash")
    PayByBkash();
else if (paymentType == "Cash")
    PayByCash();
```

Problems:

- Adding a new payment method requires **modifying existing code**
- Violates the **Open/Closed Principle (OCP)**

✅ Good Example (Decoupled)

```
interface IPayment
{
    void Pay();
}

class CardPayment : IPayment
{
    public void Pay() => Console.WriteLine("Card payment");
}

class BkashPayment : IPayment
{
    public void Pay() => Console.WriteLine("Bkash payment");
}
```

✓ **New payment → create a new class**

✓ Existing code remains unchanged

POLYMORPHISM

Inject Polymorphic Dependencies Using Dependency Injection (DI)

👉 Use Dependency Injection to swap behavior easily.

➡ Today: CardPayment

➡ Tomorrow: BkashPayment

✓ Runtime behavior swapping

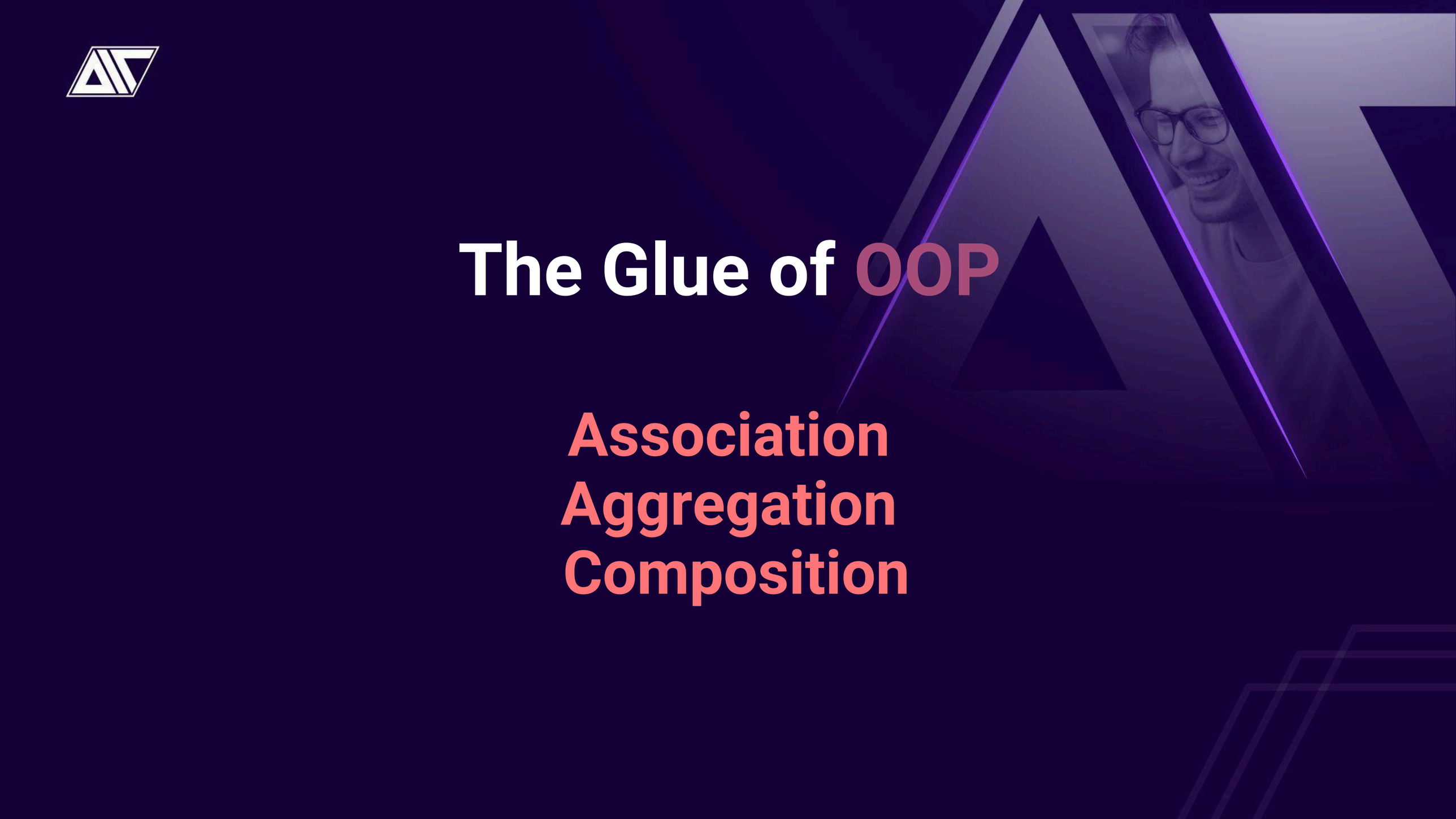
✓ Clean, scalable architecture

Use Polymorphism to Encapsulate Variability

👉 Isolate changing behavior into separate classes.

Example: Pricing Strategy

```
interface IPricingStrategy
{
    decimal Calculate(decimal price);
}

class RegularPrice : IPricingStrategy
{
    public decimal Calculate(decimal price) => price;
}

class DiscountPrice : IPricingStrategy
{
    public decimal Calculate(decimal price) => price * 0.9m;
}
```



The Glue of OOP

Association
Aggregation
Composition

ASSOCIATION

Association is a structural relationship that defines how objects are connected to each other.

It represents a "uses-a" or "has-a" relationship between two or more classes.

It does not imply ownership—just a connection or communication between instances.
Independent Lifecycle: Associated objects can exist **independently** of each other, (**loosest coupling**)

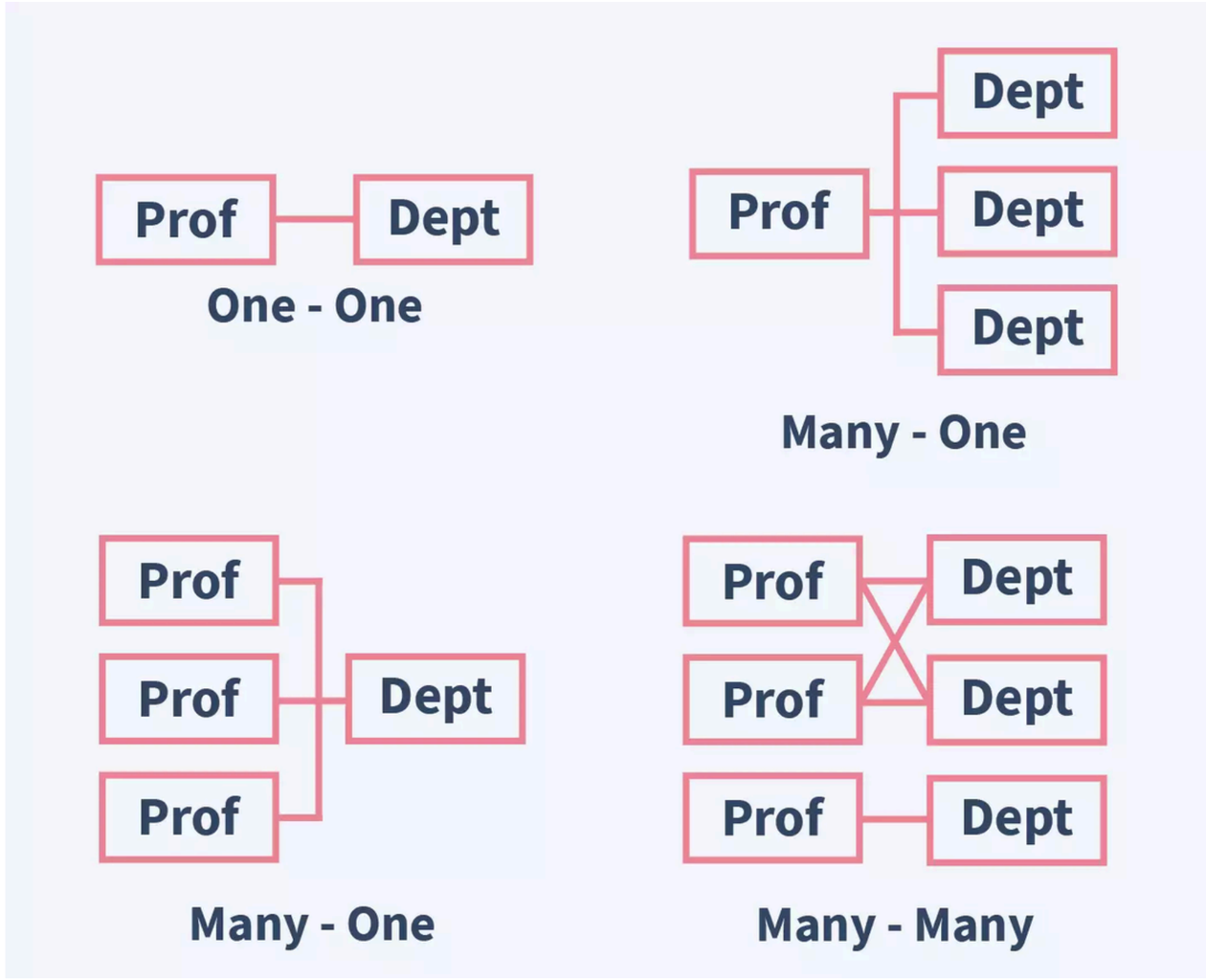
```
class Student
{
    public string Name { get; set; }
}

class Course
{
    public string Title { get; set; }
}

// association
Student student = new Student() { Name = "Naim" };
Course course = new Course() { Title = "OOP" };

// student knows about course
```

✅ Key: Both Student and Course exist independently.

ASSOCIATION

Key Points:

- Weak relationship
- Object lifecycle independent
- One-to-one, one-to-many, many-to-many



ASSOCIATION

Types of Association: By CardinalityWeak relationship

- One-to-One (1:1)
- One-to-Many (1:N)
- Many-to-Many (N:N)

```
// One-to-One
class Customer { public Address ShippingAddress; }
class Address { public Customer Resident; }

// One-to-Many
class Seller { public List<Product> Products; }

// Many-to-Many
class Product { public List<Order> Orders; }
class Order { public List<Product> Products; }
```

See Example 4

ASSOCIATION

Association by Direction

Association can also be classified based on how objects know about each other, i.e., the direction of the relationship.

Unidirectional Association

Theory

- Only one class knows about the other
- Object A can use Object B
- Object B does not know about Object A
- Common in loosely coupled designs

Example in Real Life

- Teacher → Student

```
class Student
{
    public string Name { get; set; }
}

class Teacher
{
    public string Name { get; set; }

    public void Teach(Student student)
    {
        Console.WriteLine($"Teacher {Name} teaches {student.Name}");
    }
}

Student s = new Student { Name = "Ali" };
Teacher t = new Teacher { Name = "Mr. Rahman" };

t.Teach(s); // Unidirectional: Teacher knows Student, Student doesn't know Teacher
```

Teacher knows students, but students don’t know about this teacher

ASSOCIATION

Bidirectional Association

Theory

- Both classes know about each other
- Changes in one object can affect the other
- Often used when mutual awareness is required

Example in Real Life

- Student ↔ Course
- Student knows which courses they take
- Course knows which students are enrolled

```
class Course
{
    public string Title { get; set; }
    public List<Student> Students { get; set; } = new List<Student>();
}

class Student
{
    public string Name { get; set; }
    public List<Course> Courses { get; set; } = new List<Course>();
}

Student s1 = new Student { Name = "Naim" };
Course c1 = new Course { Title = "Math" };

// Both sides know each other
s1.Courses.Add(c1);
c1.Students.Add(s1);

Console.WriteLine($"{s1.Name} enrolled in {s1.Courses[0].Title}");
Console.WriteLine($"{c1.Students[0].Name} enrolled in {c1.Title}");
```

AGGREGATION

Aggregation is a special form of association where a "whole" class has a relationship with its "parts". It represents a **"has-a"** relationship between **two or more classes**.

Aggregation is a “has-a” relationship where:

- One object uses or owns another object
- BUT the child object can exist independently
- This is a weak ownership relationship

👉 If the parent object is destroyed, the **child object still exists.**

Real-life example

- Department has Teachers
- Team has Players
- Library has Books

If the Department is removed, Teachers still exist.

AGGREGATION

Aggregation vs Association (Quick Reminder)

- Association → Just a connection
- Aggregation → Association + ownership meaning

(but still weak)

See Example : 6

COMPOSITION

Composition is a stronger form of aggregation where the "parts" are integral to the existence of the "whole".

Composition is a **strong “has-a”** relationship where:

- One object owns another object
- The child object cannot exist without the parent
- Parent controls the lifecycle of the child

👉 **If the parent is destroyed, the child is destroyed automatically.**

🧠 **Real-Life Examples**

- House has Rooms
- Car has Engine

If House is gone → Rooms are gone

If Car is gone → Engine is gone

See Example : 7

COMPOSITION

🧠 One Golden Rule (Interview 🔥)

Child created inside Parent → Composition
Child passed from outside → Aggregation
Just reference / usage → Association

Feature	Association	Aggregation	Composition
Relationship	Uses-a	Has-a (weak)	Has-a (strong)
Ownership	❌ No	✅ Weak	✅ Strong
Child lifetime depends on parent	❌ No	❌ No	✅ Yes
Child created inside parent	❌ No	❌ No	✅ Yes
Coupling	Loose	Medium	Tight
UML Symbol	Line	Empty Diamond ◇	Filled Diamond ◆

INHERITANCE(IS-A) VS COMPOSITION(HAS-A)

A rule of thumb is, choose inheritance when an object needs all of the behavior of the parent class. If your class only needs part of the functionality of another class use composition to get only the behavior you desire.

- Use inheritance when a class needs all behavior of its parent (is-a).
- Use composition when a class needs only part of another class's behavior (has-a).
- When unsure, prefer composition.
- Inheritance causes tight coupling and is harder to maintain.
- Composition provides better flexibility and maintainability.

🧠 **Final Memory Trick**

- Inheritance → "You ARE me"
- Composition → "You USE me"

🧠 MASTER DECISION TABLE

Situation	Use This
Child is a type of parent	Inheritance
Class owns another class	Composition
Class uses independent object	Aggregation
Classes only communicate	Association

CONCLUSION

- Object-Oriented Programming is more than a coding paradigm — it’s a mindset that promotes clean, scalable, and maintainable software.
- OOP concepts lay the foundation for design patterns and principles.
- Good software architecture isn't just about using OOP — it's about applying OOP through smart design choices.


THANK YOU!