ĐẠI NAM
UNIVERSITY

THIẾT KẾ, LẬP TRÌNH BACKEND

TỔNG QUAN VỀ NGÔN NGỮ C#

NỘI DUNG

1. Giới thiệu tổng quan về .NET & cài đặt môi trường
2. Giới thiệu về C#
3. Các kiểu dữ liệu
4. Toán tử và luồng điều khiển
5. Phương thức và tham số
6. Lớp
7. Kế thừa
8. Giao diện
9. Các kiểu giá trị
10. Xử lý ngoại lệ
11. Lập trình tổng quát
12. Các chủ đề nâng cao

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **.NET là gì?**
  - Là một **nền tảng** (platform) phát triển phần mềm **miễn phí**, **mã nguồn mở** (open-source).
    - Tính "miễn phí" và "mã nguồn mở" làm tăng tính hấp dẫn và cộng đồng mạnh mẽ.
  - Cho phép xây dựng nhiều loại ứng dụng: **Web** (ASP.NET Core), **Mobile** (MAUI, Xamarin), **Desktop** (WinForms, WPF, MAUI), **Cloud**, **Games** (Unity), **IoT**, **AI**,...
  - **Đa nền tảng** (Cross-platform): Chạy được trên **Windows**, **macOS**, và **Linux**.
    - "Đa nền tảng" là một trong những ưu điểm lớn nhất của .NET hiện đại.
  - Được phát triển và bảo trì bởi **Microsoft** **cùng cộng đồng**.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Hành trình phát triển của .NET**
  - **.NET Framework** (Cũ):
    - Phiên bản gốc, ra đời năm **2002**.
    - Chỉ chạy trên Windows.
    - Dùng cho ứng dụng Windows (WinForms, WPF) và web cũ (ASP.NET Web Forms).
    - Không còn được khuyến nghị cho dự án mới.
  - **.NET Core** (Tiền thân hiện đại):
    - Ra mắt khoảng 2016 để hiện đại hóa .NET.
    - Đa nền tảng (Windows, macOS, Linux).
    - Mã nguồn mở, hiệu năng cao, modular.
    - Nền tảng cho ASP.NET Core ban đầu (phiên bản 1.x -> 3.x).

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Hành trình phát triển của .NET**
  - **.NET 5+ (Nền tảng thống nhất - Hiện tại & Tương lai):**
    - Bắt đầu từ .NET 5 (2020), tiếp theo là .NET 6 LTS, .NET 7, .NET 8 LTS , .NET 9, .NET 10
    - Thống nhất .NET Framework và .NET Core thành MỘT nền tảng duy nhất.
    - Không còn chữ "Core" hay "Framework" trong tên gọi.
    - Đây là phiên bản chúng ta sẽ học và sử dụng.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Hành trình phát triển của .NET**
  - **Ưu điểm của .NET thống nhất**
    - Một **.NET** cho tất cả: Cùng bộ thư viện (BCL), cùng runtime (CLR), cùng bộ công cụ (SDK) cho mọi loại ứng dụng.
    - **Đơn giản hóa**: Lập trình viên chỉ cần học một nền tảng chính.
    - **Hiệu năng cao**: Tiếp tục tối ưu hóa tốc độ và sử dụng tài nguyên.
    - **Đa nền tảng thực sự**: Phát triển một lần, chạy nhiều nơi dễ dàng hơn.
    - **Phát triển nhanh hơn**: Chu kỳ phát hành hàng năm, cập nhật tính năng mới liên tục.
    - **Hỗ trợ dài hạn (LTS)**: Các phiên bản như .NET 6, .NET 8 được hỗ trợ lâu dài, ổn định cho sản phẩm.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Các thành phần chính của .NET**
  - **Runtime (CLR - Common Language Runtime):**
    - "Trái tim" của .NET.
    - Thực thi mã code, quản lý bộ nhớ (Garbage Collection - GC), bảo mật,...
  - **Base Class Library (BCL):**
    - Thư viện các lớp (class) dựng sẵn khổng lồ.
    - Cung cấp chức năng thông dụng: xử lý chuỗi, ngày tháng, vào/ra file, mạng, collections,...
  - **SDK (Software Development Kit):**
    - Bộ công cụ để xây dựng ứng dụng.
    - Bao gồm trình biên dịch (compiler), công cụ dòng lệnh dotnet, thư viện chuẩn.
  - **Ngôn ngữ lập trình:**
    - Hỗ trợ chính: C# (chúng ta sẽ học), F#, Visual Basic.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Thực hành:**
  - Kiểm tra SDK: `dotnet --list-sdks`
  - Kiểm tra Runtime: `dotnet --list-runtimes`

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Giới thiệu về C#:**
  - Là một **ngôn ngữ lập trình** (programming language) mạnh mẽ và linh hoạt.
  - Được phát triển bởi **Microsoft**, dẫn đầu bởi Anders Hejlsberg (người cũng tạo ra Turbo Pascal, Delphi).
  - **Hiện đại**: Liên tục được cập nhật và phát triển (phiên bản hiện tại là **C# 14** đi cùng **.NET 10**).
  - **Hướng đối tượng** (Object-Oriented Programming - OOP): Cho phép tổ chức code theo cách mô phỏng thế giới thực, giúp code dễ quản lý, mở rộng và tái sử dụng (sẽ học kỹ hơn ở các module sau).
  - **Type-Safe** (An toàn kiểu dữ liệu): Trình biên dịch kiểm tra kiểu dữ liệu rất chặt chẽ, giúp phát hiện lỗi sớm ngay khi viết code, giảm lỗi khi chương trình chạy.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Những điểm mạnh của C#:**
  - **Đa năng** (Versatile): Nhờ chạy trên .NET, C# có thể dùng để xây dựng hầu hết mọi loại ứng dụng (web, mobile, desktop, game, cloud, AI,...).
  - **Cú pháp (Syntax) rõ ràng, dễ đọc**: Có nhiều nét tương đồng với C, C++, Java, giúp người đã biết các ngôn ngữ này dễ tiếp cận hơn. Cú pháp tương đối trong sáng.
  - **Tích hợp chặt chẽ với .NET**: Tận dụng tối đa sức mạnh của nền tảng .NET:
    - Thư viện nền tảng phong phú (BCL): Cung cấp sẵn vô số chức năng.
    - Runtime (CLR): Quản lý bộ nhớ tự động (Garbage Collection), bảo mật,...
  - **Component-Oriented**: Dễ dàng tạo ra các thành phần (component) phần mềm có thể đóng gói và tái sử dụng.
  - **Cộng đồng lớn và tài liệu tốt**: Rất nhiều tài nguyên học tập, diễn đàn hỗ trợ (Stack Overflow, Microsoft Learn), thư viện mã nguồn mở.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Mối quan hệ giữa C# và .NET:**
  - C# là ngôn ngữ lập trình chính và phổ biến nhất được thiết kế để hoạt động trên nền tảng **.NET**.
  - **Code C# không chạy trực tiếp**: Nó được biên dịch (compile) thành một dạng mã trung gian gọi là Intermediate Language (IL) hoặc Common Intermediate Language (CIL).
  - **.NET Runtime** (CLR) sau đó sẽ lấy mã IL này, biên dịch nó một lần nữa (Just-In-Time compilation - JIT) thành mã máy (machine code) và thực thi chương trình.
  - Lập trình viên C# **sử dụng trực tiếp** các lớp và chức năng từ **Base Class Library (BCL)** của .NET để thực hiện công việc.
  - Liên tưởng qua ví von:
    - .NET giống như một nhà máy hoặc một hệ điều hành ảo (cung cấp môi trường, công cụ, thư viện).
    - C# là ngôn ngữ chính bạn dùng để viết bản thiết kế (source code) cho các sản phẩm làm trong nhà máy đó.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Ứng dụng thực tế của C# với .NET:**
  - **Ứng dụng Web & Web APIs**: Dùng ASP.NET Core. Xây dựng backend cho web, các dịch vụ RESTful API mạnh mẽ. (Trọng tâm chính sau này của khóa học).
  - **Ứng dụng Desktop**:
    - Windows: WinForms, WPF (truyền thống).
    - Đa nền tảng (Windows, macOS): .NET MAUI.
  - **Ứng dụng Mobile**: .NET MAUI (thay thế cho Xamarin) để tạo app cho iOS và Android từ một codebase C#.
  - **Phát triển Game**: C# là ngôn ngữ scripting chính cho Unity Engine, một trong những game engine phổ biến nhất thế giới.
  - **Cloud Computing**: Xây dựng và triển khai ứng dụng trên các nền tảng đám mây như Microsoft Azure.Và nhiều hơn nữa: IoT (Internet of Things), AI & Machine Learning (với ML.NET), Microservices,...

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **ASP.NET Core là gì?:**
  - Là một framework (khung sườn) phát triển ứng dụng web.
  - Miễn phí, mã nguồn mở, được phát triển bởi Microsoft và cộng đồng.
  - Xây dựng trên nền tảng **.NET** hiện đại (.NET 5/6/7/8+), kế thừa các ưu điểm:
    - **Đa nền tảng** (Cross-Platform): Chạy tốt trên Windows, macOS, Linux.
    - **Hiệu năng cực cao** (High-Performance): Tối ưu hóa cho tốc độ và khả năng mở rộng, đặc biệt phù hợp cho **cloud** và **microservices**.
    - **Modular**: Chỉ thêm các thành phần (components) cần thiết cho dự án.
  - **Quan trọng**: Đây là phiên bản thiết kế lại hoàn toàn, kế nhiệm cho ASP.NET Framework cũ (chỉ chạy trên Windows). Không phải là một bản cập nhật đơn thuần.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Mục đích chính của ASP.NET Core là gì?:** ASP.NET Core được thiết kế chủ yếu để xây dựng 3 loại ứng dụng/chức năng web cốt lõi:
  - **Web UIs (Giao diện người dùng Web):**
    - Tạo ra các trang web động, nơi người dùng tương tác trực tiếp qua trình duyệt.
    - Server sẽ tạo (render) mã HTML và gửi về cho trình duyệt hiển thị.
    - Công nghệ phổ biến: Razor Pages, MVC (Model-View-Controller).
    - Ví dụ: Trang tin tức, blog, diễn đàn, trang quản trị.
    - Blazor là một lựa chọn mới hơn để xây dựng UI tương tác phong phú bằng C#.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Mục đích chính của ASP.NET Core là gì?:** ASP.NET Core được thiết kế chủ yếu để xây dựng 3 loại ứng dụng/chức năng web cốt lõi:
  - **Web APIs (Giao diện lập trình ứng dụng Web):**
    - Xây dựng các điểm cuối (endpoints) mà các ứng dụng khác (như app mobile, ứng dụng JavaScript frontend - SPA, hệ thống backend khác) có thể gọi để lấy dữ liệu hoặc thực thi chức năng.
    - Không tạo ra giao diện HTML trực tiếp cho người dùng cuối.
    - Thường tuân theo kiến trúc RESTful và trả về dữ liệu dưới dạng JSON hoặc XML.
    - Ví dụ: API cung cấp danh sách sản phẩm cho app mobile, API xác thực người dùng.

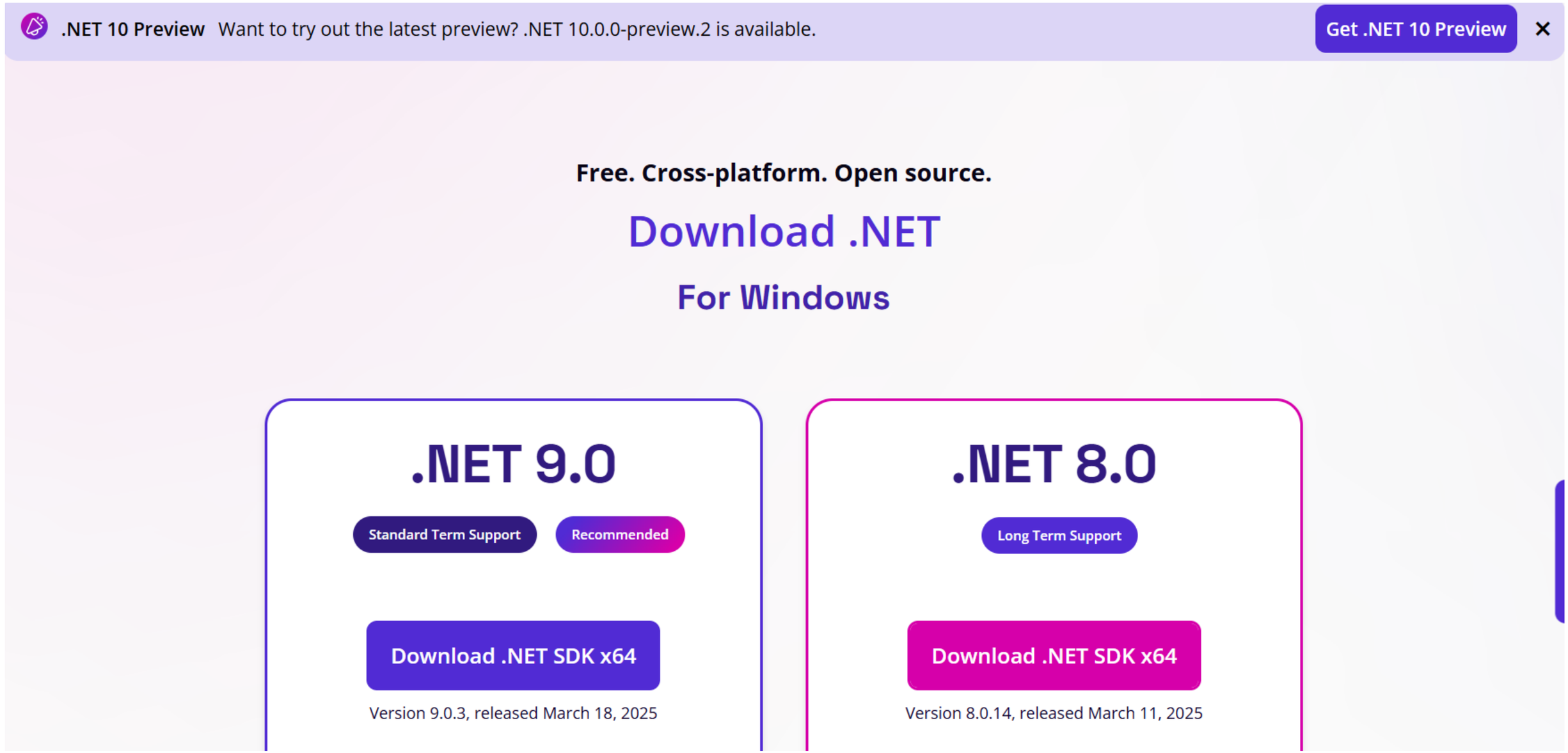
1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Mục đích chính của ASP.NET Core là gì?:** ASP.NET Core được thiết kế chủ yếu để xây dựng 3 loại ứng dụng/chức năng web cốt lõi:
  - **Real-time Web Functionality (Chức năng Web thời gian thực)**:
    - Cho phép giao tiếp hai chiều ngay lập tức giữa server và các client đã kết nối mà không cần client phải liên tục hỏi (polling).
    - Công nghệ chính: SignalR.
    - Ví dụ: Ứng dụng chat, thông báo đẩy (push notifications), bảng điều khiển cập nhật dữ liệu trực tiếp, game online đơn giản.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Cài đặt .NET SDK**
  - **SDK (Software Development Kit - Bộ phát triển phần mềm):**
    - Là bộ công cụ đầy đủ dành cho lập trình viên.Bao gồm mọi thứ cần thiết để: viết code, biên dịch (compile), xây dựng (build), và chạy ứng dụng .NET.
    - Quan trọng: Khi bạn cài SDK, nó đã bao gồm cả Runtime tương ứng.
  - **Runtime (Môi trường chạy):**
    - Chỉ là môi trường cần thiết để chạy các ứng dụng .NET đã được biên dịch sẵn.
    - Người dùng cuối chỉ cần cài Runtime nếu họ muốn chạy ứng dụng .NET của bạn.
  - **Chúng ta cần cài gì? -> SDK, vì chúng ta sẽ viết code!**
    - Chọn phiên bản nào?
    - Phiên bản LTS (Long-Term Support - Hỗ trợ dài hạn) mới nhất.Đảm bảo sự ổn định và nhận cập nhật bảo mật/sửa lỗi trong ít nhất 3 năm. Chứa các tính năng mới nhất của C# và ASP.NET Core mà chúng ta sẽ học hoặc dự án sẽ cần.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

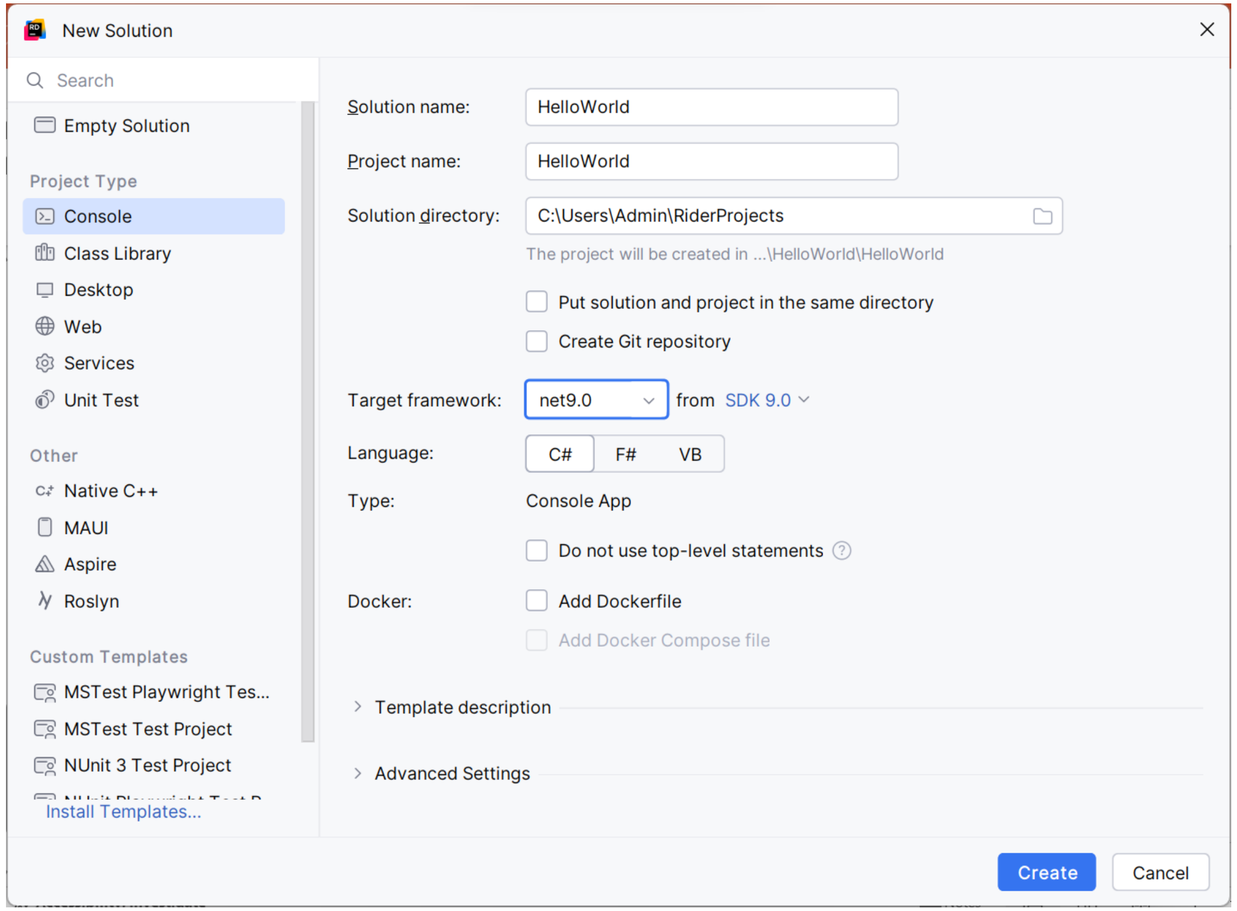


1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Các lựa chọn IDE hàng đầu cho .NET**
  - **Microsoft Visual Studio** (2022): Đặc biệt là phiên bản Community (Miễn phí).
  - **Microsoft Visual Studio Code** (VS Code): Cần cài thêm Extension C# Dev Kit.
  - **JetBrains Rider**: Sản phẩm thương mại (có bản dùng thử/giấy phép đặc biệt).

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Chương trình HelloWorld**
  - **Microsoft Visual Studio** (2022): Đặc biệt là phiên bản Community (Miễn phí).
  - **Microsoft Visual Studio Code** (VS Code): Cần cài thêm Extension C# Dev Kit.
    - `dotnet new cosole`
    - `dotnew run`
  - **JetBrains Rider**: Sản phẩm thương mại



1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Quy trình Build trong .NET diễn ra như thế nào?**
  - **(1) Viết Code (Source Code)**: Bạn viết mã nguồn bằng ngôn ngữ C# trong các file có đuôi .cs (ví dụ: Program.cs).
  - **(2) Biên dịch (Compile)**: Khi bạn nhấn nút "Build" trong IDE hoặc chạy lệnh dotnet build, trình biên dịch C# (Roslyn) sẽ đọc mã nguồn .cs của bạn.
  - **(3) Tạo Mã Trung Gian (Intermediate Language - IL)**: Trình biên dịch không tạo ra mã máy trực tiếp mà tạo ra một loại mã gọi là IL (hoặc CIL). Mã IL này là mã "chung chung", chưa phụ thuộc vào CPU hay Hệ điều hành cụ thể. Mã IL được đóng gói vào các file Assembly (thường có đuôi .dll hoặc .exe). Các file này chứa logic chương trình của bạn.
  - **(4) Lưu trữ Assembly**: Các file Assembly này thường được lưu trong thư mục bin/Debug (khi phát triển) hoặc bin/Release (khi phát hành) của dự án.
  - **(5) Thực thi (Execution)**: Khi bạn chạy ứng dụng (nhấn nút "Run" trong IDE hoặc dùng lệnh dotnet run), .NET Runtime (cụ thể là CLR - Common Language Runtime) được tải lên.

1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

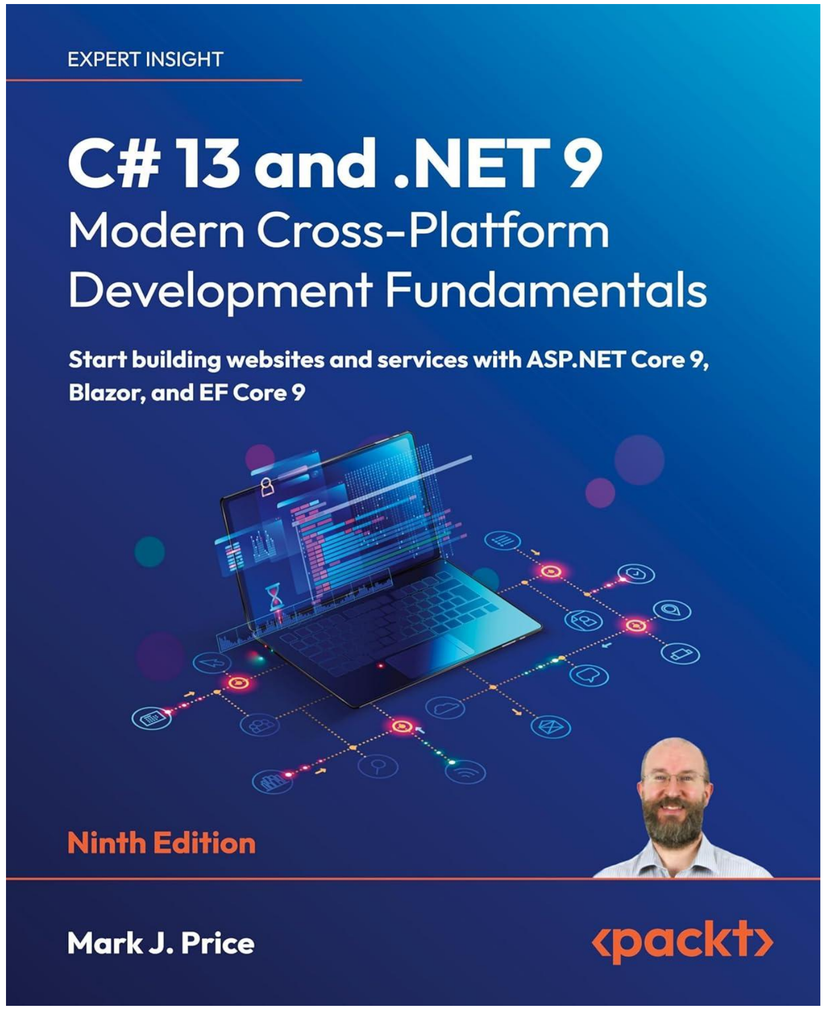
- **Quy trình Build trong .NET diễn ra như thế nào?**
  - **(6) Biên dịch JIT (Just-In-Time)**: CLR sẽ đọc mã IL từ file Assembly. Tại thời điểm cần thực thi một đoạn mã IL nào đó, bộ biên dịch JIT của CLR sẽ dịch đoạn mã IL đó thành Mã Máy (Native Code) - là mã lệnh mà CPU của máy bạn có thể hiểu và chạy trực tiếp.
  - **(7) Chạy trên CPU**: Mã máy được CPU thực thi, và chương trình của bạn hoạt động!
  - *Diễn giải thêm: Bước biên dịch ra IL giúp code C# có thể chạy trên nhiều hệ điều hành (.NET hỗ trợ). Bước JIT giúp tối ưu hóa mã cho CPU cụ thể tại thời điểm chạy.*

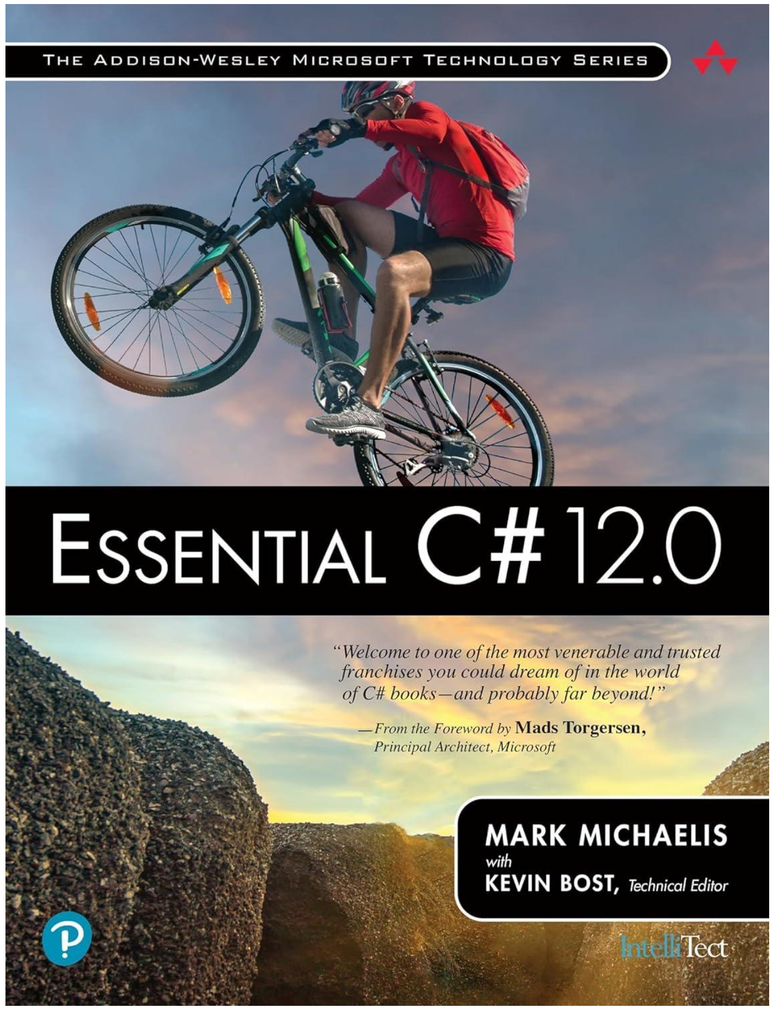
1. GIỚI THIỆU VÀ CÀI ĐẶT MÔI TRƯỜNG

- **Quy trình Build trong .NET diễn ra như thế nào?**
  - **(6) Biên dịch JIT (Just-In-Time)**: CLR sẽ đọc mã IL từ file Assembly. Tại thời điểm cần thực thi một đoạn mã IL nào đó, bộ biên dịch JIT của CLR sẽ dịch đoạn mã IL đó thành Mã Máy (Native Code) - là mã lệnh mà CPU của máy bạn có thể hiểu và chạy trực tiếp.
  - **(7) Chạy trên CPU**: Mã máy được CPU thực thi, và chương trình của bạn hoạt động!
  - *Diễn giải thêm: Bước biên dịch ra IL giúp code C# có thể chạy trên nhiều hệ điều hành (.NET hỗ trợ). Bước JIT giúp tối ưu hóa mã cho CPU cụ thể tại thời điểm chạy.*

2. LẬP TRÌNH C# CĂN BẢN

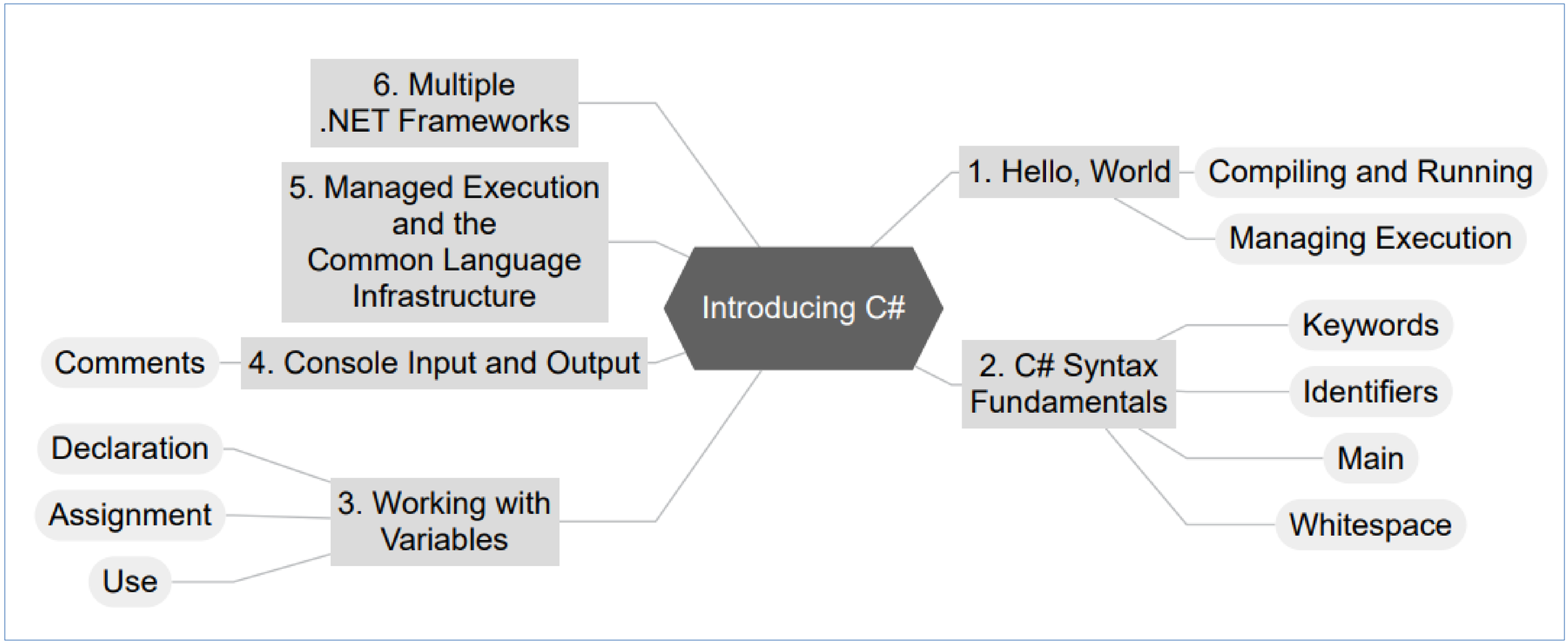
- **Giáo trình/Sách tham khảo: https://essentialcsharp.com/home**





2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 01**



2. LẬP TRÌNH C# CĂN BẢN

- **Cấu trúc của một chương trình C# cơ bản**

```
namespace HelloYou;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Cấu trúc của một chương trình C# cơ bản**
  - **namespace**: Giống như một thư mục ảo để tổ chức code, giúp tránh trùng tên.
  - **class**: Một "bản thiết kế" cho đối tượng, chứa dữ liệu và phương thức. Mọi code thực thi trong C# đều phải nằm trong một class.
  - **Main** method: Đây là điểm khởi đầu của mọi ứng dụng C#. Nó có dạng

static void Main(string[] args)

- **static**: Có nghĩa là bạn có thể gọi phương thức Main mà không cần tạo đối tượng từ lớp chứa nó.
- **void**: Có nghĩa là phương thức Main không trả về giá trị nào.
- **string[] args**: Là một mảng các chuỗi, dùng để nhận các tham số dòng lệnh (sẽ tìm hiểu sau).

2. LẬP TRÌNH C# CĂN BẢN

- **Xuất dữ liệu ra console:** Để hiển thị văn bản hoặc kết quả ra màn hình console (cửa sổ dòng lệnh), chúng ta sử dụng `Console.WriteLine("Nội dung cần in");`
  - **Console**: Là một lớp (class) cung cấp các phương thức để làm việc với cửa sổ console.
  - **WriteLine**: Là một phương thức (method) của lớp Console, dùng để in một chuỗi ký tự ra màn hình và tự động xuống dòng mới.

2. LẬP TRÌNH C# CĂN BẢN

- **Namespaces và using**
  - Lớp Console nằm trong namespace System. Nếu không dùng using, bạn phải viết đầy đủ là `System.Console.WriteLine(...)`.
  - Lệnh using System; ở đầu file cho phép bạn sử dụng các thành phần trong namespace System (như Console) một cách trực tiếp mà không cần tiền tố System.
- **Chú thích (Comments)**: Dùng để giải thích code, trình biên dịch sẽ bỏ qua chúng
  - // Chú thích một dòng: Mọi thứ từ // đến cuối dòng là chú thích.
  - /* Chú thích ... */: Mọi thứ nằm giữa /* và */ (có thể trên nhiều dòng) là chú thích.

2. LẬP TRÌNH C# CĂN BẢN

- **Biên dịch và Thực thi (Khái niệm cơ bản)**
  - Bạn viết code C# vào file **.cs**.
  - Trình biên dịch C# (compiler) đọc file .cs và dịch nó thành mã trung gian gọi là Common Intermediate Language (**CIL**), đóng gói trong một file assembly (**.exe hoặc .dll**).
  - Khi bạn chạy chương trình, Common Language Runtime (CLR) sẽ tải assembly này, sử dụng trình biên dịch **Just-In-Time (JIT)** để dịch mã CIL thành mã máy gốc (native code) mà CPU có thể hiểu và thực thi. CLR cũng quản lý bộ nhớ (Garbage Collection), bảo mật, v.v.
  - **Framework Class Library (FCL)**: Là một thư viện khổng lồ chứa các lớp và phương thức dựng sẵn (như Console) mà CLR cung cấp để bạn sử dụng.

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 01:** Chương trình "Hello, World!" cơ bản.

```
// Sử dụng namespace System để có thể dùng lớp Console
using System;

// Khai báo một namespace tên là HelloWorldApp
namespace HelloWorldApp
{
    // Khai báo một lớp tên là Program
    class Program
    {
        // Phương thức Main - điểm bắt đầu của chương trình
        static void Main(string[] args)
        {
            // In dòng chữ "Hello, World!" ra màn hình console
            Console.WriteLine("Hello, World!");
            // Bạn có thể thêm lệnh này để cửa sổ console không đóng ngay lập tức khi chạy từ
Visual Studio
            // Console.ReadKey();
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Không sử dụng using System;.

```
// Không có 'using System;'

namespace HelloWorldAppNoUsing
{
    class Program
    {
        static void Main(string[] args)
        {
            // Phải viết đầy đủ tên namespace khi gọi WriteLine
            System.Console.WriteLine("Hello without using System!");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 3**: Sử dụng Chú thích

```
using System;

namespace CommentsExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Đây là chú thích một dòng. Lệnh dưới đây sẽ in ra lời chào.
            Console.WriteLine("Xin chào!");

            /* Đây là chú thích nhiều dòng.
               Nó có thể kéo dài trên nhiều dòng khác nhau.
               Lệnh tiếp theo sẽ in ra một thông điệp khác.
            */
            Console.WriteLine("Đây là chương trình C# đầu tiên của tôi.");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: In nhiều dòng

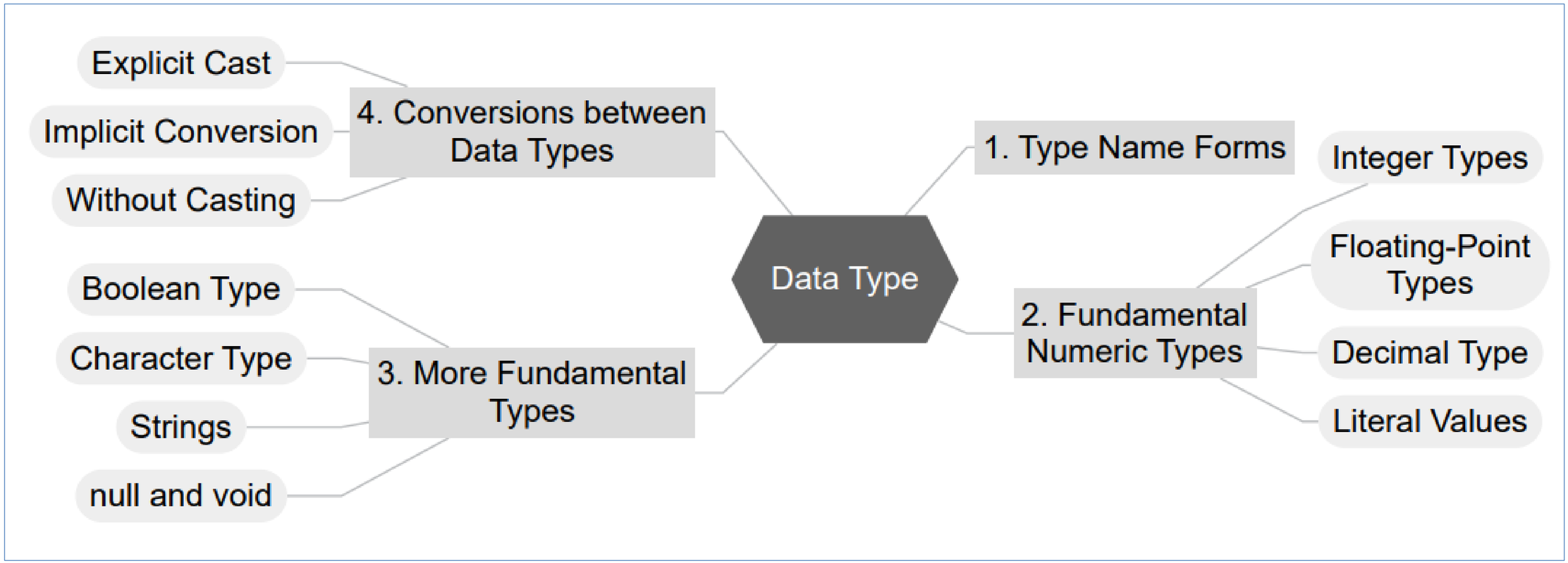
```
using System;

namespace MultipleLines
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Dòng thứ nhất.");
            Console.WriteLine("Dòng thứ hai.");
            Console.WriteLine("Dòng thứ ba.");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Bài tập thực hành**: Xem yêu cầu chi tiết BTTH Chương 01 trong sách BTTH

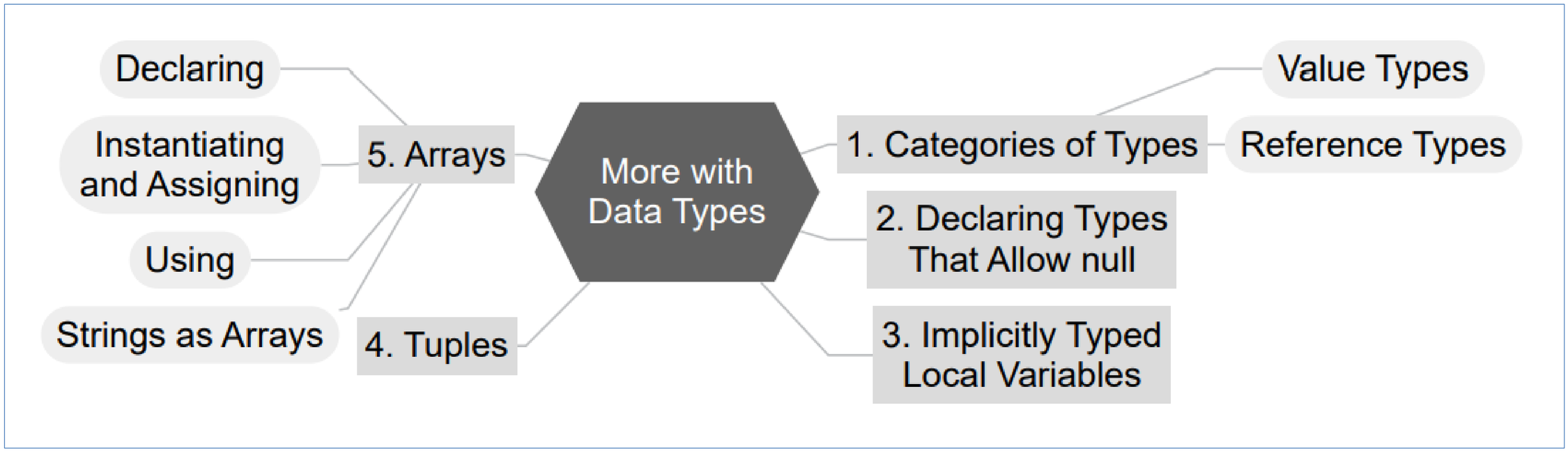
2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 02 + 03**



2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 02**



2. LẬP TRÌNH C# CĂN BẢN

- **Kiểu dữ liệu là gì?** Trong lập trình, mỗi mẩu dữ liệu (như một con số, một dòng chữ) cần được xác định rõ "kiểu" của nó. Kiểu dữ liệu quy định:
  - Loại giá trị mà một biến có thể lưu trữ (ví dụ: số nguyên, số thực, ký tự, chuỗi văn bản).
  - Dung lượng bộ nhớ cần thiết để lưu trữ giá trị đó.
  - Các phép toán hợp lệ có thể thực hiện trên giá trị đó (ví dụ: bạn có thể cộng hai số nguyên, nhưng không thể cộng một số nguyên với một dòng chữ theo nghĩa toán học).

2. LẬP TRÌNH C# CĂN BẢN

- **Các kiểu dữ liệu cơ bản phổ biến:**
  - **Số nguyên**: Dùng để lưu trữ các số không có phần thập phân.
    - **int**: Kiểu phổ biến nhất, đủ cho hầu hết các trường hợp thông thường (khoảng -2 tỷ đến +2 tỷ).
    - **long**: Lưu số nguyên lớn hơn int.
    - **short**: Lưu số nguyên nhỏ hơn int.
    - **byte**: Lưu số nguyên dương rất nhỏ (0 đến 255).
  - **Số thực (dấu phẩy động): Dùng để lưu trữ các số có phần thập phân.**
    - **double**: Kiểu phổ biến nhất cho số thực, độ chính xác khá cao.
    - **float**: Độ chính xác thấp hơn double, tốn ít bộ nhớ hơn.
    - **decimal**: Độ chính xác rất cao, thường dùng trong các ứng dụng tài chính, tiền tệ để tránh sai số làm tròn.

2. LẬP TRÌNH C# CĂN BẢN

- **Các kiểu dữ liệu cơ bản phổ biến:**
  - **Ký tự:**
    - **char**: Lưu một ký tự Unicode duy nhất. Giá trị được đặt trong dấu nháy đơn ('A', '5', '?').
  - **Chuỗi ký tự:**
    - **string**: Lưu một chuỗi các ký tự. Giá trị được đặt trong dấu nháy kép ("Hello World", "Đây là một chuỗi").
  - **Luận lý (Boolean):**
    - **bool**: Chỉ có thể lưu một trong hai giá trị: true hoặc false. Thường dùng trong các biểu thức điều kiện.

2. LẬP TRÌNH C# CĂN BẢN

- **Khai báo biến:**
  - **Cú pháp**: Kiểu_dữ_liệu Tên_biến;
  - **Ví dụ**: int age; (khai báo biến tên age kiểu int), string customerName; (khai báo biến customerName kiểu string).
- **Gán Giá Trị: Sau khi khai báo, bạn có thể gán giá trị cho biến bằng toán tử =.**
  - **Cú pháp**: Tên_biến = Giá_trị;
  - **Ví dụ**: age = 30;, customerName = "Nguyễn Văn A";
  - Có thể khởi tạo giá trị ngay khi khai báo: int score = 100;, bool isActive = true;

2. LẬP TRÌNH C# CĂN BẢN

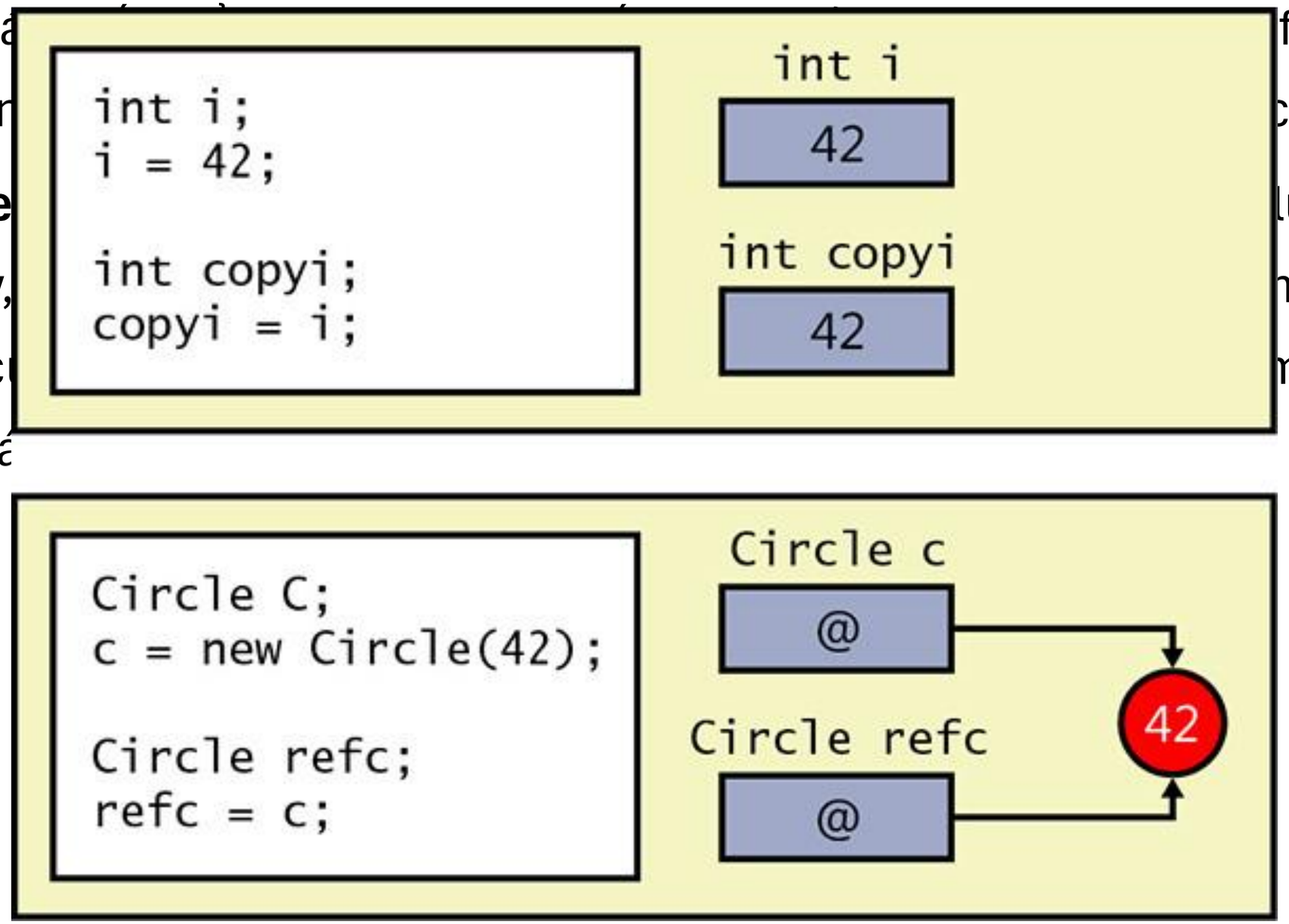
- **Từ khóa var (Implicit Typing):** C# cho phép dùng var thay cho tên kiểu dữ liệu khi khai báo và khởi tạo biến cùng lúc. Trình biên dịch sẽ tự động suy ra kiểu dữ liệu dựa trên giá trị bạn gán.
  - Ví dụ: `var myNumber = 10;` (trình biên dịch hiểu myNumber là int), `var myMessage = "Hi";` (trình biên dịch hiểu myMessage là string).
  - **Lưu ý**: Biến được khai báo bằng var vẫn là biến có kiểu dữ liệu cụ thể (strongly-typed), chỉ là bạn không cần viết tường minh tên kiểu lúc khai báo. Bạn phải khởi tạo giá trị ngay khi dùng var.

2. LẬP TRÌNH C# CĂN BẢN

- **Kiểu Giá Trị (Value Types) vs. Kiểu Tham Chiếu (Reference Types) (Giới thiệu):**
  - **Value Types**: Các biến kiểu này chứa trực tiếp giá trị của dữ liệu (Ví dụ: `int`, `float`, `bool`, `char`, `struct`). Khi gán biến value type này cho biến khác, một bản sao của giá trị sẽ được tạo ra.
  - **Reference Types**: Các biến kiểu này chứa địa chỉ (tham chiếu) đến vùng nhớ lưu trữ dữ liệu thực tế (Ví dụ: `string`, `array`, `class`). Khi gán biến reference type này cho biến khác, chỉ địa chỉ được sao chép, cả hai biến sẽ cùng trỏ đến cùng một đối tượng trong bộ nhớ.
  - (*Chúng ta sẽ tìm hiểu kỹ hơn về sự khác biệt này trong các chương sau, đặc biệt khi làm việc với phương thức và lớp*).

2. LẬP TRÌNH C# CĂN BẢN

- **Kiểu Giá Trị (Value Types) vs. Kiểu Tham Chiếu (Reference Types) (Giới thiệu):**
  - **Value Types**: Ca… float, bool, char, struct). Khi bạn gán biến… c tạo ra.
  - **Reference Type**… lưu trữ dữ liệu thực tế (Ví dụ: string, array,… hỉ địa chỉ được sao chép, cả hai biến sẽ c… m hiểu kỹ hơn về sự khác biệt này trong cá…



2. LẬP TRÌNH C# CĂN BẢN

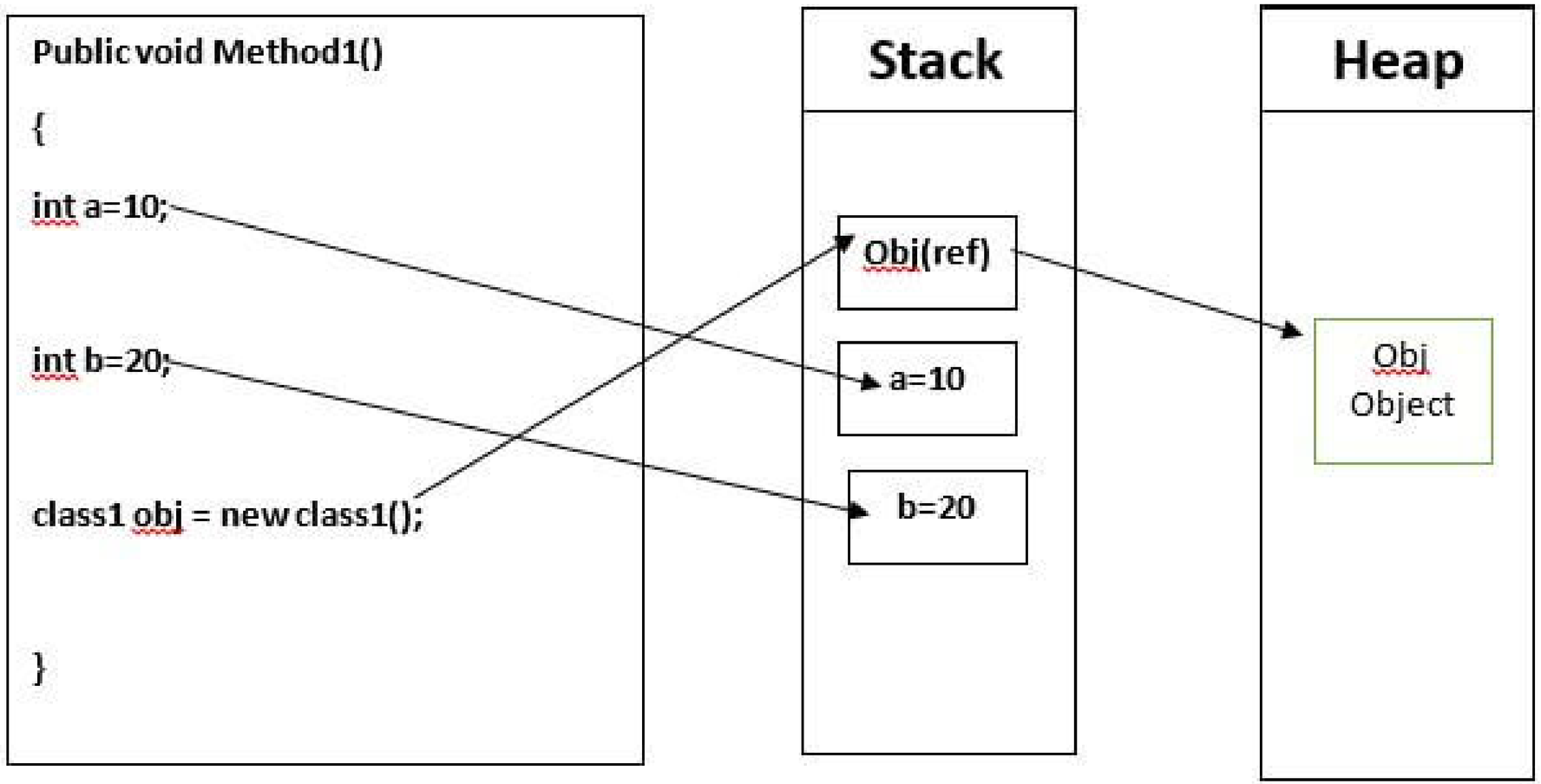
- **Kiểu Giá Trị (Value Types) vs. Kiểu Tham Chiếu (Reference Types) (Giới thiệu):**
  - **Valu**

    Khi l
  - **Refe**

    dụ: s

    cả h

    biệt



char, struct).

ệu thực tế (Ví

ợc sao chép,

n về sự khác

2. LẬP TRÌNH C# CĂN BẢN

- **Nullable Value Types (?)**: Mặc định, các kiểu giá trị (như int, bool) không thể chứa giá trị null (nghĩa là "không có giá trị"). Thêm dấu ? sau tên kiểu cho phép chúng có thể chứa null.
  - Ví dụ: int? productCount = null;, bool? isConfirmed = null;
  - Bạn có thể kiểm tra xem biến nullable có giá trị hay không bằng thuộc tính .HasValue (trả về true/false) hoặc so sánh trực tiếp với null.
- **Tuples**: Là một cách tiện lợi để nhóm nhiều giá trị có thể khác kiểu lại với nhau mà không cần định nghĩa một class hay struct mới.
  - Ví dụ khai báo và khởi tạo: (string Name, int Age) personInfo = ("Alice", 30);
  - Truy cập phần tử: Console.WriteLine(personInfo.Name); // In ra "Alice"

2. LẬP TRÌNH C# CĂN BẢN

- **Mảng (Arrays)**: Dùng để lưu trữ một tập hợp các phần tử cùng kiểu dữ liệu. Các phần tử được lưu trữ liên tiếp nhau trong bộ nhớ và được truy cập thông qua một chỉ số (index) bắt đầu từ 0.
  - **Khai báo**: Kiểu_dữ_liệu[] Tên_mảng; (Ví dụ: `int[] scores;, string[] names;`)
  - **Khởi tạo (cấp phát bộ nhớ)**: Tên_mảng = new Kiểu_dữ_liệu[Số_lượng_phần_tử]; (Ví dụ: `scores = new int[5];` // Tạo mảng chứa 5 số nguyên)
  - **Khởi tạo với giá trị ban đầu**: `int[] numbers = { 10, 20, 30, 40 };` // Tạo mảng 4 số nguyên với giá trị cụ thể.
  - **Truy cập phần tử**: Tên_mảng[Chỉ_số] (Ví dụ: `scores[0] = 95;` // Gán giá trị cho phần tử đầu tiên, `Console.WriteLine(numbers[1]);` // In ra phần tử thứ hai là 20).

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Sử dụng kiểu số (int, double) và phép toán cơ bản

```
using System;
namespace NumericTypesExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Khai báo và khởi tạo biến kiểu int
            int quantity = 5;
            int pricePerUnit = 15000;

            // Tính tổng tiền (int * int = int)
            int totalPrice = quantity * pricePerUnit;
            Console.WriteLine("Số lượng: " + quantity);
            Console.WriteLine("Đơn giá: " + pricePerUnit);
            Console.WriteLine("Tổng tiền (int): " + totalPrice);

            // Khai báo và khởi tạo biến kiểu double
            double length = 10.5;
            double width = 4.2;

            // Tính diện tích (double * double = double)
            double area = length * width;
            Console.WriteLine("Chiều dài: " + length);
            Console.WriteLine("Chiều rộng: " + width);
            Console.WriteLine("Diện tích (double): " + area);

            // Khai báo biến decimal cho tính toán tài chính
            decimal accountBalance = 1234567.89m; // Lưu ý hậu tố 'm' cho decimal
            Console.WriteLine("Số dư tài khoản (decimal): " + accountBalance);
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Sử dụng char, string, bool và nối chuỗi

```
using System;

namespace OtherTypesExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Kiểu char (ký tự đơn)
            char grade = 'A';
            Console.WriteLine("Điểm: " + grade);

            // Kiểu string (chuỗi ký tự)
            string firstName = "John";
            string lastName = "Doe";
            string fullName = firstName + " " + lastName; // Nối chuỗi bằng dấu '+'
            Console.WriteLine("Tên đầy đủ: " + fullName);

            // Kiểu bool (luận lý)
            bool isAvailable = true;
            bool isExpired = false;
            Console.WriteLine("Còn hàng: " + isAvailable);
            Console.WriteLine("Đã hết hạn: " + isExpired);
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 3**: Khai báo, gán giá trị và sử dụng var

```
using System;

namespace VarExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Khai báo tường minh
            int userAge;
            string userName;

            // Gán giá trị sau
            userAge = 25;
            userName = "Alice";
            Console.WriteLine("Tên (khai báo tường minh): " + userName);
            Console.WriteLine("Tuổi (khai báo tường minh): " + userAge);

            // Sử dụng var (phải khởi tạo ngay)
            var productCode = 1001; // Trình biên dịch suy ra kiểu int
            var productName = "Laptop"; // Trình biên dịch suy ra kiểu string
            var productPrice = 1200.50; // Trình biên dịch suy ra kiểu double

            Console.WriteLine("Mã SP (dùng var): " + productCode);
            Console.WriteLine("Tên SP (dùng var): " + productName);
            Console.WriteLine("Giá SP (dùng var): " + productPrice);

            // Lấy kiểu dữ liệu thực tế của biến khai báo bằng var
            Console.WriteLine("Kiểu của productCode: " + productCode.GetType());
            Console.WriteLine("Kiểu của productName: " + productName.GetType());
            Console.WriteLine("Kiểu của productPrice: " + productPrice.GetType());
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Sử dụng Nullable Value Types (int?)

```
using System;

namespace NullableExample
{
    class Program
    {
        static void Main(string[] args)
        {
            int? numberOfOrders = null; // Có thể chưa có đơn hàng nào

            // Kiểm tra xem có giá trị hay không trước khi sử dụng
            if (numberOfOrders.HasValue)
            {
                Console.WriteLine("Số đơn hàng: " + numberOfOrders.Value);
            }
            else
            {
                Console.WriteLine("Chưa có đơn hàng nào.");
            }

            numberOfOrders = 15; // Gán giá trị
             if (numberOfOrders.HasValue)
            {
                Console.WriteLine("Số đơn hàng hiện tại: " + numberOfOrders.Value);
            }
            else
            {
                Console.WriteLine("Chưa có đơn hàng nào.");
            }

            // Cách kiểm tra khác
            if(numberOfOrders == null)
            {
                 Console.WriteLine("Biến đang là null");
            }
            else
            {
                 Console.WriteLine("Biến không phải là null, giá trị: " + numberOfOrders);
            }
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 5**: Sử dụng Tuples đơn giản

```
using System;

namespace TupleExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Tạo Tuple để lưu tọa độ (x, y)
            (int x, int y) point = (10, 20);

            // Truy cập các phần tử bằng tên
            Console.WriteLine("Tọa độ X: " + point.x);
            Console.WriteLine("Tọa độ Y: " + point.y);

            // Tạo Tuple với kiểu dữ liệu khác nhau
            (string product, double price, int quantity) item = ("Sách C#", 25.5, 5);
            Console.WriteLine("Sản phẩm: " + item.product);
            Console.WriteLine("Giá: " + item.price);
            Console.WriteLine("Số lượng: " + item.quantity);
        }
    }
}
```

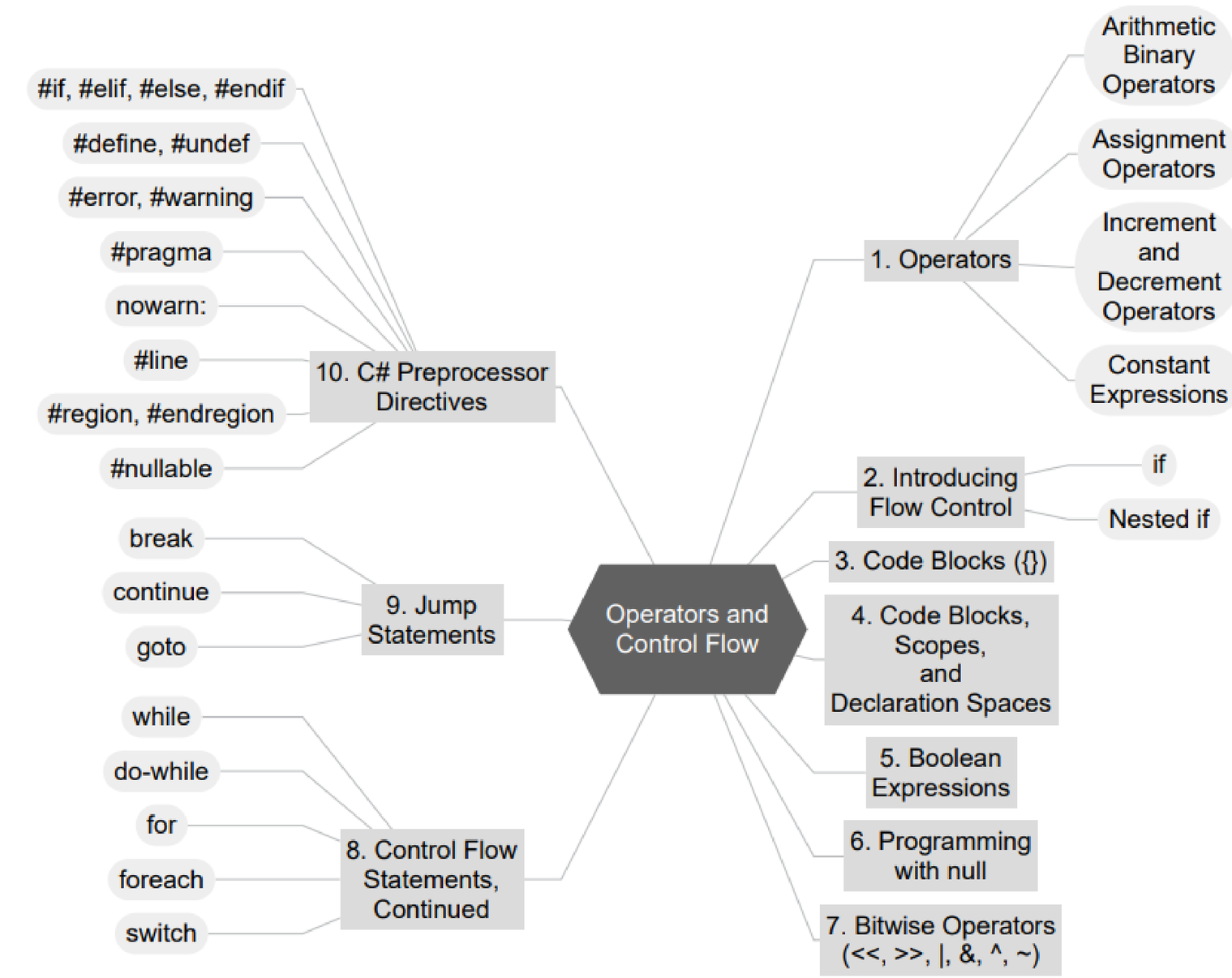
2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 6**: Sử dụng Mảng số nguyên (int[])

```
using System;

namespace ArrayExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Khai báo và khởi tạo mảng điểm số với giá trị cụ thể
            int[] scores = { 85, 92, 78, 90, 88 };

            // Truy cập và in phần tử đầu tiên (chỉ số 0)
            Console.WriteLine("Điểm số đầu tiên: " + scores[0]); // Output: 85

            // Truy cập và in phần tử thứ ba (chỉ số 2)
            Console.WriteLine("Điểm số thứ ba: " + scores[2]); // Output: 78

            // Thay đổi giá trị của phần tử thứ hai (chỉ số 1)
            scores[1] = 95;
            Console.WriteLine("Điểm số thứ hai sau khi thay đổi: " + scores[1]); // Output: 95

            // Khai báo mảng chuỗi tên các ngày trong tuần (khởi tạo sau)
            string[] daysOfWeek = new string[7];
            daysOfWeek[0] = "Thứ Hai";
            daysOfWeek[1] = "Thứ Ba";
            // ... gán cho các ngày còn lại ...
            daysOfWeek[6] = "Chủ Nhật";

            Console.WriteLine("Ngày đầu tuần: " + daysOfWeek[0]);
            Console.WriteLine("Ngày cuối tuần: " + daysOfWeek[6]);

            // Lấy số lượng phần tử của mảng
            Console.WriteLine("Mảng scores có " + scores.Length + " phần tử.");
            Console.WriteLine("Mảng daysOfWeek có " + daysOfWeek.Length + " phần tử.");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 4**



2. LẬP TRÌNH C# CĂN BẢN

- **Toán Tử (Operators)**: Là các ký hiệu đặc biệt dùng để thực hiện các phép toán trên một hoặc nhiều giá trị (toán hạng - operands)
  - Toán tử Số học:
    - + (cộng), - (trừ), * (nhân), / (chia).
    - Lưu ý: phép chia số nguyên (int / int) sẽ cho kết quả là phần nguyên.% (Modulo - chia lấy dư): Trả về phần dư của phép chia số nguyên (Ví dụ: 10 % 3 là 1).
  - Toán tử Gán:
    - = (Gán đơn giản): Gán giá trị bên phải cho biến bên trái.
    - Toán tử Gán phức hợp: Là cách viết tắt kết hợp phép toán với gán (+=, -=, *=, /=, %=). Ví dụ: x += 5; tương đương với x = x + 5;.

2. LẬP TRÌNH C# CĂN BẢN

- **Toán Tử (Operators)**: Là các ký hiệu đặc biệt dùng để thực hiện các phép toán trên một hoặc nhiều giá trị (toán hạng - operands)
  - Toán tử Tăng/Giảm:
    - ++ (Tăng 1), -- (Giảm 1).
    - Tiền tố (Prefix): ++x, --x. Tăng/giảm giá trị của x trước, sau đó trả về giá trị mới.
    - Hậu tố (Postfix): x++, x--. Trả về giá trị hiện tại của x trước, sau đó mới tăng/giảm giá trị của x.
  - Toán tử So sánh: Dùng để so sánh hai giá trị, kết quả luôn là true hoặc false (kiểu bool).
    - == (Bằng), != (Không bằng)> (Lớn hơn), < (Nhỏ hơn)>= (Lớn hơn hoặc bằng), <= (Nhỏ hơn hoặc bằng)

2. LẬP TRÌNH C# CĂN BẢN

- **Toán Tử (Operators)**: Là các ký hiệu đặc biệt dùng để thực hiện các phép toán trên một hoặc nhiều giá trị (toán hạng - operands)
  - Toán tử Logic: Dùng để kết hợp các biểu thức boolean.
    - ! (Logical NOT - Phủ định): Đảo ngược giá trị boolean (!true là false).
    - && (Logical AND - Và): Trả về true nếu cả hai toán hạng đều true.
    - || (Logical OR - Hoặc): Trả về true nếu ít nhất một toán hạng là true.
  - Toán tử Điều kiện (Ba ngôi - Ternary): Cách viết gọn của if-else.
    - Cú pháp: biểu_thức_điều_kiện ? giá_trị_nếu_đúng : giá_trị_nếu_sai
  - Độ ưu tiên Toán tử: Các toán tử có độ ưu tiên khác nhau (nhân/chia ưu tiên hơn cộng/trừ). Dùng dấu ngoặc đơn () để thay đổi thứ tự ưu tiên nếu cần.

2. LẬP TRÌNH C# CĂN BẢN

- **Khối Lệnh ({ }) và Phạm Vi (Scope):**
  - Cặp dấu ngoặc nhọn {} dùng để nhóm một hoặc nhiều câu lệnh thành một khối lệnh duy nhất.
  - Phạm vi (Scope): Biến được khai báo bên trong một khối lệnh (ví dụ: bên trong if, for, while) chỉ có thể được truy cập từ bên trong khối lệnh đó hoặc các khối lệnh con lồng bên trong nó. Không thể truy cập biến đó từ bên ngoài khối lệnh chứa nó.
- **Biểu Thức Boolean**: Là các biểu thức sử dụng toán tử so sánh và logic, kết quả cuối cùng là true hoặc false. Đây là nền tảng cho các câu lệnh điều khiển luồng.

2. LẬP TRÌNH C# CĂN BẢN

- **Câu Lệnh Điều Khiển Luồng (Control Flow Statements)**: Cho phép bạn điều khiển thứ tự thực thi các câu lệnh trong chương trình, thay vì chỉ chạy tuần tự từ trên xuống dưới.
  - **Lệnh Rẽ nhánh (Selection Statements)**: Quyết định khối lệnh nào sẽ được thực thi.
    - if (condition): Thực thi khối lệnh theo sau nếu condition là true.
    - if (condition) { ... } else { ... }: Nếu condition là true, thực thi khối lệnh trong if. Ngược lại (false), thực thi khối lệnh trong else.
    - if (condition1) { ... } else if (condition2) { ... } else { ... }: Kiểm tra condition1 trước, nếu đúng thì thực thi khối lệnh tương ứng và kết thúc. Nếu sai, kiểm tra condition2, ... Cuối cùng, nếu tất cả các điều kiện trên đều sai, thực thi khối lệnh trong else (nếu có).
    - switch (expression): So sánh giá trị của expression với các giá trị hằng trong các case. Nếu khớp case nào, thực thi các lệnh từ case đó cho đến khi gặp lệnh break. Lệnh break là bắt buộc để thoát khỏi switch. default: là một case tùy chọn, sẽ được thực thi nếu không có case nào khác khớp.

2. LẬP TRÌNH C# CĂN BẢN

- **Câu Lệnh Điều Khiển Luồng (Control Flow Statements)**: Cho phép bạn điều khiển thứ tự thực thi các câu lệnh trong chương trình, thay vì chỉ chạy tuần tự từ trên xuống dưới.
  - **Lệnh Lặp (Iteration Statements / Loops): Lặp lại việc thực thi một khối lệnh nhiều lần.**
    - **for (khởi_tạo; điều_kiện_lặp; bước_lặp) { ... }**: Thường dùng khi biết trước số lần lặp.
      - khởi_tạo: Thực thi một lần duy nhất khi bắt đầu vòng lặp (thường để khởi tạo biến đếm).
      - điều_kiện_lặp: Kiểm tra trước mỗi lần lặp. Nếu true, thực thi khối lệnh. Nếu false, thoát vòng lặp.
      - bước_lặp: Thực thi sau mỗi lần lặp (thường để cập nhật biến đếm).

2. LẬP TRÌNH C# CĂN BẢN

- **Câu Lệnh Điều Khiển Luồng (Control Flow Statements)**: Cho phép bạn điều khiển thứ tự thực thi các câu lệnh trong chương trình, thay vì chỉ chạy tuần tự từ trên xuống dưới.
  - **Lệnh Lặp (Iteration Statements / Loops): Lặp lại việc thực thi một khối lệnh nhiều lần.**
    - while (điều_kiện_lặp) { ... }: Lặp lại khối lệnh chừng nào điều_kiện_lặp còn true. Kiểm tra điều kiện trước mỗi lần lặp. Có thể không chạy lần nào nếu điều kiện ban đầu là false.
    - do { ... } while (điều_kiện_lặp);: Giống while, nhưng kiểm tra điều kiện sau khi thực thi khối lệnh. Do đó, khối lệnh luôn được thực thi ít nhất một lần.
    - foreach (Kiểu_dữ_liệu tên_biến in tập_hợp) { ... }: Cách thuận tiện để lặp qua từng phần tử của một tập hợp (như mảng đã học ở Chương 2). Trong mỗi lần lặp, tên_biến sẽ giữ giá trị của phần tử hiện tại.

2. LẬP TRÌNH C# CĂN BẢN

- **Câu lệnh điều khiển luồng (Control Flow Statements)**: Cho phép bạn điều khiển thứ tự thực thi các câu lệnh trong chương trình, thay vì chỉ chạy tuần tự từ trên xuống dưới.
  - **Lệnh Nhảy (Jump Statements): Thay đổi luồng thực thi một cách đột ngột.**
    - **break;**: Thoát ngay lập tức khỏi vòng lặp (for, while, do, foreach) hoặc cấu trúc switch gần nhất.
    - **continue;**: Bỏ qua phần còn lại của lần lặp hiện tại và chuyển ngay sang lần lặp tiếp theo của vòng lặp (for, while, do, foreach).
    - **return;**: (Sẽ học kỹ hơn ở Chương 4) Thoát khỏi phương thức hiện tại.
    - **goto label;**: (Ít dùng, nên tránh) Nhảy đến một nhãn (label:) trong code.
- **Chỉ thị tiền xử lý (Preprocessor Directives)**: Các lệnh bắt đầu bằng # được xử lý trước khi mã nguồn được biên dịch thực sự. Thường dùng cho việc biên dịch có điều kiện (conditional compilation). Ví dụ: #define DEBUG, #if DEBUG ... #endif.

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Toán tử số học và gán phức hợp

```
using System;

namespace ArithmeticOperators
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10;
            int b = 3;
            int sum = a + b;
            int difference = a - b;
            int product = a * b;
            int quotient = a / b; // Chia số nguyên, kết quả là 3
            int remainder = a % b; // Chia lấy dư, kết quả là 1
            double quotientDouble = (double)a / b; // Ép kiểu để chia số thực, kết quả là 3.33...

            Console.WriteLine($"a = {a}, b = {b}");
            Console.WriteLine($"Tổng: {sum}"); // 13
            Console.WriteLine($"Hiệu: {difference}"); // 7
            Console.WriteLine($"Tích: {product}"); // 30
            Console.WriteLine($"Thương (int): {quotient}"); // 3
            Console.WriteLine($"Phần dư: {remainder}"); // 1
            Console.WriteLine($"Thương (double): {quotientDouble}"); // 3.33...

            int score = 100;
            score += 50; // score = score + 50;
            Console.WriteLine($"Điểm sau khi cộng: {score}"); // 150
            score /= 3; // score = score / 3;
            Console.WriteLine($"Điểm sau khi chia: {score}"); // 50
        }
    }
}
```

- **Ví dụ 2**: Toán tử tăng/giảm (Prefix vs Postfix)

```
using System;

namespace IncrementDecrement
{
    class Program
    {
        static void Main(string[] args)
        {
            int count = 5;
            Console.WriteLine($"Count ban đầu: {count}"); // 5

            // Postfix: gán giá trị cũ (5) cho currentCount, sau đó tăng count lên 6
            int currentCountPost = count++;
            Console.WriteLine($"Giá trị gán (Postfix): {currentCountPost}"); // 5
            Console.WriteLine($"Count sau Postfix: {count}"); // 6

            // Reset count
            count = 5;
            Console.WriteLine($"Count reset: {count}"); // 5

            // Prefix: tăng count lên 6 trước, sau đó gán giá trị mới (6) cho currentCountPre
            int currentCountPre = ++count;
            Console.WriteLine($"Giá trị gán (Prefix): {currentCountPre}"); // 6
            Console.WriteLine($"Count sau Prefix: {count}"); // 6
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 3**: Toán tử so sánh và logic

```
using System;

namespace ComparisonLogical
{
    class Program
    {
        static void Main(string[] args)
        {
            int age = 25;
            double gpa = 3.5;
            bool hasScholarship = false;
            bool isAdult = age >= 18;
            Console.WriteLine($"Là người lớn: {isAdult}"); // True

            bool isExcellent = gpa >= 3.6;
            Console.WriteLine($"Học lực xuất sắc: {isExcellent}"); // False

            bool canGraduate = isAdult && (gpa >= 2.0);
            Console.WriteLine($"Đủ điều kiện tốt nghiệp: {canGraduate}"); // True

            bool specialConsideration = isExcellent || hasScholarship;
            Console.WriteLine($"Cân nhắc đặc biệt: {specialConsideration}"); // False

            Console.WriteLine($"Phủ định của isAdult: {!isAdult}"); // False
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Toán tử điều kiện (ba ngôi)

```
using System;

namespace TernaryOperator
{
    class Program
    {
        static void Main(string[] args)
        {
            int temperature = 15;
            string weather = temperature >= 25 ? "Nóng" : "Mát hoặc Lạnh";
            Console.WriteLine($"Thời tiết: {weather}"); // Mát hoặc Lạnh
            int score = 75;
            string result = score >= 50 ? "Đạt" : "Không đạt";
            Console.WriteLine($"Kết quả: {result}"); // Đạt
        }
    }
}
```

- **Ví dụ 5**: Câu lệnh if-else if-else và phạm vi biến

```
using System;

namespace IfElseScope
{
    class Program
    {
        static void Main(string[] args)
        {
            int points = 82;
            string grade; // Khai báo grade ở phạm vi ngoài if-else

            if (points >= 90)
            {
                grade = "A";
                string message = "Xuất sắc!"; // message chỉ tồn tại trong khối if này
                Console.WriteLine(message);
            }
            else if (points >= 80)
            {
                grade = "B";
                // Console.WriteLine(message); // Lỗi! Không thể truy cập message ở đây
            }
            else if (points >= 70)
            {
                grade = "C";
            }
            else
            {
                grade = "D";
                bool needsImprovement = true; // Biến này chỉ tồn tại trong khối else
                Console.WriteLine($"Cần cải thiện: {needsImprovement}");
            }

            Console.WriteLine($"Điểm: {points}, Xếp loại: {grade}");
            // Console.WriteLine(needsImprovement); // Lỗi! Không thể truy cập ở đây
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 6**: Câu lệnh switch-case

```
using System;

namespace SwitchCase
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Nhập một ngày trong tuần (1-7): ");
            // Đọc input từ người dùng (sẽ học kỹ hơn sau)
            // Tạm thời gán cứng giá trị để minh họa
            int day = 3;
            string dayName;

            switch (day)
            {
                case 1:
                    dayName = "Thứ Hai";
                    break; // Thoát khỏi switch
                case 2:
                    dayName = "Thứ Ba";
                    break;
                case 3:
                    dayName = "Thứ Tư";
                    break;
                case 4:
                    dayName = "Thứ Năm";
                    break;
                case 5:
                    dayName = "Thứ Sáu";
                    break;
                case 6:
                    dayName = "Thứ Bảy";
                    break;
                case 7:
                    dayName = "Chủ Nhật";
                    break;
                default: // Nếu không khớp case nào ở trên
                    dayName = "Ngày không hợp lệ";
                    break;
            }
            Console.WriteLine($"Hôm nay là: {dayName}");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 7**: Vòng lặp for

```
using System;

namespace ForLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Đếm từ 1 đến 5:");
            // i = 1: Khởi tạo
            // i <= 5: Điều kiện lặp
            // i++: Bước lặp (tăng i sau mỗi lần lặp)
            for (int i = 1; i <= 5; i++)
            {
                Console.WriteLine(i);
            }

            Console.WriteLine("\nTính tổng các số chẵn từ 2 đến 10:");
            int sumEven = 0;
            for (int j = 2; j <= 10; j += 2) // Bước lặp là j = j + 2
            {
                sumEven += j;
            }
            Console.WriteLine($"Tổng các số chẵn: {sumEven}"); // 2 + 4 + 6 + 8 + 10 = 30
        }
    }
}
```

- **Ví dụ 8**: Vòng lặp while

```
using System;

namespace WhileLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            int countdown = 5;
            Console.WriteLine("Đếm ngược từ 5 về 1:");
            while (countdown >= 1) // Kiểm tra điều kiện trước
            {
                Console.WriteLine(countdown);
                countdown--; // Giảm biến đếm
            }
            Console.WriteLine("Bắt đầu!");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 9**: Vòng lặp do .. while

```
using System;

namespace DoWhileLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            int number;
            // Vòng lặp này đảm bảo chạy ít nhất 1 lần để lấy input
            do
            {
                Console.WriteLine("Nhập một số dương (nhập số âm hoặc 0 để thoát): ");
                // Tạm gán giá trị để minh họa, phần đọc input sẽ học sau
                // Giả sử lần 1 người dùng nhập -5
                number = -5; // Lần đầu gán -5 để minh họa thoát ngay
                // number = 5; // Thử gán số dương để xem lặp lại

                if(number > 0) {
                    Console.WriteLine($"Bạn đã nhập số dương: {number}");
                }

            } while (number > 0); // Kiểm tra điều kiện sau

            Console.WriteLine("Đã thoát khỏi vòng lặp do-while.");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 10**: Vòng lặp foreach với mảng

```
using System;

namespace ForEachLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] fruits = { "Táo", "Chuối", "Cam", "Xoài" };

            Console.WriteLine("Danh sách các loại trái cây:");
            foreach (string fruit in fruits) // Lặp qua từng 'fruit' trong mảng 'fruits'
            {
                Console.WriteLine("- " + fruit);
            }

            int[] numbers = { 10, 20, 30, 40, 50 };
            int sum = 0;
            foreach(int num in numbers)
            {
                sum += num;
            }
            Console.WriteLine($"\nTổng các số trong mảng: {sum}"); // 150
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 11**: break và continue trong vòng lặp mảng

```
using System;

namespace BreakContinue
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tìm số 7 và dừng lại (dùng break):");
            int[] data = { 2, 5, 1, 7, 4, 9 };
            foreach (int item in data)
            {
                Console.WriteLine($"Đang kiểm tra: {item}");
                if (item == 7)
                {
                    Console.WriteLine("Đã tìm thấy số 7!");
                    break; // Thoát khỏi vòng lặp foreach ngay lập tức
                }
            }

            Console.WriteLine("\nIn ra các số lẻ (bỏ qua số chẵn dùng continue):");
            for (int i = 1; i <= 10; i++)
            {
                if (i % 2 == 0) // Nếu i là số chẵn
                {
                    continue; // Bỏ qua phần còn lại của lần lặp này, đi đến lần lặp tiếp theo
                }
                // Lệnh này chỉ chạy khi i là số lẻ
                Console.Write(i + " ");
            }
            Console.WriteLine(); // Xuống dòng mới
        }
    }
}
```

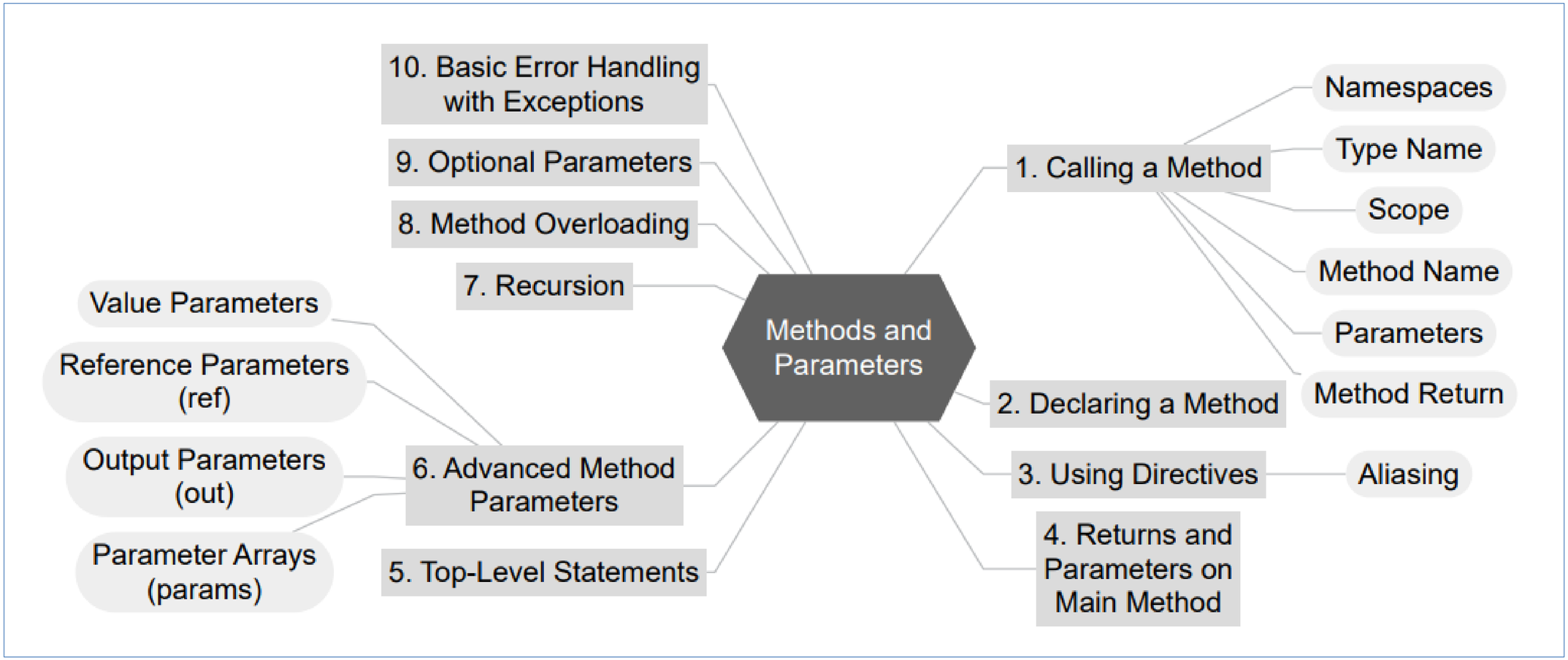
2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 12**: Chỉ thị tiền xử lý #define / #if

```
#define EXPERIMENTAL_FEATURE // Định nghĩa một biểu tượng tiền xử lý

using System;

namespace PreprocessorDirectives
{
    class Program
    {
        static void Main(string[] args)
        {
             Console.WriteLine("Tính năng cơ bản luôn chạy.");
            #if EXPERIMENTAL_FEATURE
                Console.WriteLine("Tính năng thử nghiệm đang được bật!");
                // Code dành riêng cho tính năng thử nghiệm sẽ nằm ở đây
                // Trình biên dịch chỉ biên dịch phần này nếu EXPERIMENTAL_FEATURE được #define
            #else
                Console.WriteLine("Tính năng thử nghiệm đang tắt.");
            #endif

            #if DEBUG // DEBUG thường được tự động định nghĩa khi biên dịch ở chế độ Debug
                 Console.WriteLine("Chế độ Debug đang bật.");
            #endif
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung chương 5**



2. LẬP TRÌNH C# CĂN BẢN

- **Phương thức (Method) là gì?**
  - Là một khối các câu lệnh được nhóm lại với nhau dưới một cái tên duy nhất để thực hiện một công việc cụ thể.
  - Lợi ích:Tổ chức code: Chia chương trình lớn thành các phần nhỏ hơn, dễ quản lý hơn.
  - Tái sử dụng: Viết một lần, gọi nhiều lần từ nhiều nơi khác nhau.
  - Module hóa: Giúp chương trình dễ đọc, dễ hiểu, dễ bảo trì và nâng cấp.

2. LẬP TRÌNH C# CĂN BẢN

- **Khai báo Phương thức:**
  - **Cú pháp chung:** [modifiers] Kiểu_trả_về Tên_Phương_thức([Danh_sách_tham_số]) { // Thân phương thức: các câu lệnh }
  - **modifiers**: Các từ khóa bổ trợ (sẽ tìm hiểu sâu hơn sau).
    - public: (Sẽ học ở chương Class) Cho phép truy cập từ bất kỳ đâu.
    - private: (Sẽ học ở chương Class) Chỉ cho phép truy cập bên trong lớp chứa nó.
    - static: Phương thức thuộc về chính lớp đó, không cần tạo đối tượng để gọi. Có thể gọi trực tiếp bằng TênLớp.TênPhươngThức(). Phương thức Main là static. Trong giai đoạn này, chúng ta sẽ chủ yếu viết các phương thức static trong lớp Program.

2. LẬP TRÌNH C# CĂN BẢN

- **Khai báo Phương thức:**
  - **Kiểu_trả_về**: Kiểu dữ liệu của giá trị mà phương thức sẽ trả về sau khi thực hiện xong.
    - **void**: Nếu phương thức không trả về giá trị nào.
    - **Kiểu cụ thể** (int, string, bool, double, mảng, ...): Nếu phương thức trả về một giá trị thuộc kiểu đó. Phải có lệnh return giá_trị; trong thân phương thức.
  - **Tên_Phương_thức**: Đặt tên theo quy tắc PascalCase (viết hoa chữ cái đầu mỗi từ, ví dụ: CalculateSum, PrintUserInfo). Tên nên gợi tả chức năng của phương thức.
  - **Danh_sách_tham_số (Parameters)**: (Tùy chọn) Danh sách các biến đầu vào mà phương thức cần để thực hiện công việc. Các tham số được ngăn cách bởi dấu phẩy, mỗi tham số có dạng Kiểu_dữ_liệu tên_tham_số. Ví dụ: (int number1, int number2), (string name).

2. LẬP TRÌNH C# CĂN BẢN

- **Gọi Phương thức (Calling a Method):**
  - **Cú pháp**: TênPhươngThức(Đối_số_1, Đối_số_2, ...); (Nếu gọi phương thức trong cùng lớp)
  - **Hoặc**: TênLớp.TênPhươngThức(Đối_số_1, ...); (Nếu gọi phương thức static từ lớp khác hoặc để rõ ràng).
  - **Đối_số (Arguments)**: Là các giá trị cụ thể bạn truyền vào cho các tham_số của phương thức khi gọi nó. Số lượng và kiểu dữ liệu của đối số phải khớp với danh sách tham số.

- **Tham số (Parameters) và Cách truyền:**
  - **Truyền Tham trị (Pass by Value - Mặc định cho Value Types)**: Khi bạn truyền một biến kiểu giá trị (int, double, bool, char, struct...) vào phương thức, một bản sao của giá trị đó được tạo ra và truyền vào. Mọi thay đổi trên tham số bên trong phương thức không ảnh hưởng đến biến gốc bên ngoài.
  - **Truyền Tham chiếu (Pass by Reference)**: Cho phép phương thức truy cập và thay đổi trực tiếp biến gốc bên ngoài. Dùng các từ khóa:
    - **ref**: Biến gốc phải được khởi tạo giá trị trước khi truyền vào phương thức. Phương thức có thể đọc và thay đổi giá trị biến gốc.
    - **out**: Biến gốc không cần phải khởi tạo trước khi truyền vào. Phương thức bắt buộc phải gán một giá trị cho tham số out trước khi kết thúc. Dùng khi muốn phương thức trả về nhiều hơn một giá trị.
    - **in**: (Ít dùng hơn, thường cho struct lớn) Truyền tham chiếu để tăng hiệu suất (tránh sao chép), nhưng đảm bảo phương thức không được phép thay đổi giá trị biến gốc.

2. LẬP TRÌNH C# CĂN BẢN

- **Tham số (Parameters) và Cách truyền:**
  - **Tham số params**: Cho phép truyền một số lượng đối số không xác định (nhưng cùng kiểu) vào phương thức. Bên trong phương thức, các đối số này được gom thành một mảng. Từ khóa params phải đi với tham số cuối cùng trong danh sách.
- **Tham số và Kiểu trả về của Main:**
  - static void Main(string[] args): args là một mảng string, chứa các đối số được truyền vào từ dòng lệnh khi chạy chương trình.
  - static int Main(...): Main cũng có thể được khai báo để trả về một int. Giá trị này là mã thoát (exit code) của chương trình, thường 0 nghĩa là thành công, các giá trị khác 0 báo hiệu lỗi.

- **Đệ quy (Recursion)**: Là hiện tượng một phương thức gọi lại chính nó. Cần có một điều kiện dừng (base case) để tránh vòng lặp vô hạn và lỗi tràn bộ nhớ stack (StackOverflowException).
- **Nạp chồng Phương thức (Method Overloading)**: Cho phép định nghĩa nhiều phương thức có cùng tên trong cùng một lớp, nhưng chúng phải khác nhau về danh sách tham số (khác về số lượng tham số, hoặc kiểu dữ liệu của tham số, hoặc cả hai). Trình biên dịch sẽ tự động chọn phiên bản phương thức phù hợp dựa trên các đối số bạn truyền vào khi gọi.
- **Tham số Tùy chọn (Optional Parameters)**: Bạn có thể gán giá trị mặc định cho một hoặc nhiều tham số ngay trong phần khai báo phương thức. Nếu người gọi không cung cấp đối số cho tham số đó, giá trị mặc định sẽ được sử dụng. Tham số tùy chọn phải được đặt sau tất cả các tham số bắt buộc.
  - Ví dụ: void PrintMessage(string message, bool isImportant = false) { ... }

2. LẬP TRÌNH C# CĂN BẢN

- **Đối số Được đặt tên (Named Arguments)**: Khi gọi phương thức, bạn có thể chỉ định rõ tên của tham số mà bạn muốn gán giá trị, thay vì dựa vào thứ tự.
  - Cú pháp: TênPhươngThức(tênThamSố1: giáTrị1, tênThamSố2: giáTrị2);
  - Lợi ích: Giúp code rõ ràng hơn, đặc biệt khi dùng với tham số tùy chọn; cho phép truyền đối số không theo thứ tự khai báo.

2. LẬP TRÌNH C# CĂN BẢN

- **Xử lý Ngoại lệ Cơ bản (Basic Exception Handling)**: Lỗi xảy ra trong quá trình chạy chương trình được gọi là ngoại lệ (exception). Nếu không xử lý, ngoại lệ sẽ làm chương trình bị dừng đột ngột (crash).
  - **Khối try-catch: Dùng để "bắt" và xử lý các ngoại lệ dự kiến.**
    - try { // Đặt code có khả năng gây lỗi vào đây }: Chương trình thử thực thi code trong khối try.
    - catch (LoạiNgoạiLệ ex) { // Code xử lý lỗi được đặt ở đây }: Nếu có ngoại lệ thuộc LoạiNgoạiLệ (hoặc kiểu con của nó) xảy ra trong try, chương trình sẽ nhảy vào khối catch tương ứng. Biến ex chứa thông tin về lỗi. Bạn có thể có nhiều khối catch cho các loại ngoại lệ khác nhau.
    - catch { // Xử lý mọi loại ngoại lệ khác }: Khối catch không chỉ định loại ngoại lệ sẽ bắt tất cả các loại lỗi chưa được bắt bởi các khối catch cụ thể trước đó (nên dùng hạn chế).
    - finally { // Code trong này luôn chạy }: (Tùy chọn) Khối lệnh này luôn được thực thi sau khi khối try (và catch, nếu có) kết thúc, bất kể có ngoại lệ xảy ra hay không. Thường dùng để giải phóng tài nguyên (đóng file, kết nối mạng...).

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Phương thức static void đơn giản

```
using System;

namespace SimpleMethod
{
    class Program
    {
        // Khai báo phương thức static void
        static void PrintWelcomeMessage()
        {
            Console.WriteLine("**************************");
            Console.WriteLine("Chào mừng đến với Chương 4!");
            Console.WriteLine("**************************");
        }

        static void Main(string[] args)
        {
            // Gọi phương thức
            PrintWelcomeMessage();

            Console.WriteLine("\nGọi lại lần nữa:");
            PrintWelcomeMessage();
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Phương thức static có tham số và trả về giá trị

```
using System;

namespace MethodWithReturn
{
    class Program
    {
        // Phương thức nhận 2 int, trả về tổng là int
        static int AddNumbers(int number1, int number2)
        {
            int sum = number1 + number2;
            return sum; // Trả về kết quả
        }

        static void Main(string[] args)
        {
            int a = 15;
            int b = 7;

            // Gọi phương thức và lưu kết quả trả về vào biến result
            int result = AddNumbers(a, b);
            Console.WriteLine($"{a} + {b} = {result}"); // 15 + 7 = 22

            // Có thể dùng trực tiếp kết quả trả về
            Console.WriteLine($"Tổng của 100 và 200 là: {AddNumbers(100, 200)}"); // 300
        }
    }
}
```

- **Ví dụ 3**: Tham số ref (Truyền tham chiếu)

```
using System;

namespace RefParameter
{
    class Program
    {
        // Phương thức hoán đổi giá trị 2 biến int dùng ref
        static void Swap(ref int x, ref int y)
        {
            Console.WriteLine($" -> Bên trong Swap TRƯỚC khi hoán đổi: x={x}, y={y}");
            int temp = x;
            x = y;
            y = temp;
            Console.WriteLine($" -> Bên trong Swap SAU khi hoán đổi: x={x}, y={y}");
        }

        static void Main(string[] args)
        {
            int num1 = 10;
            int num2 = 20;

            Console.WriteLine($"TRƯỚC khi gọi Swap: num1={num1}, num2={num2}"); // 10, 20

            // Khi gọi phương thức có ref, phải dùng từ khóa ref trước đối số
            Swap(ref num1, ref num2);

            Console.WriteLine($"SAU khi gọi Swap: num1={num1}, num2={num2}"); // 20, 10 (Giá trị gốc đã thay đổi)
        }
    }
}
```

- **Ví dụ 4**: Tham số out (Lấy kết quả đầu ra)

```
using System;

namespace OutParameter
{
    class Program
    {
        // Phương thức cố gắng chia a cho b
        // Trả về true nếu thành công, false nếu b là 0
        // Kết quả phép chia (nếu thành công) được trả về qua tham số out 'result'
        static bool TryDivide(double a, double b, out double result)
        {
            if (b == 0)
            {
                result = 0; // Phải gán giá trị cho tham số out trước khi return
                return false; // Chia thất bại
            }
            else
            {
                result = a / b; // Gán kết quả
                return true; // Chia thành công
            }
        }

        // Ví dụ sử dụng int.TryParse (là phương thức dựng sẵn dùng out)
        static void ParseInput(string input)
        {
            Console.WriteLine($"\nĐang thử phân tích chuỗi: \"{input}\"");
            // int.TryParse trả về bool, kết quả parse (nếu thành công) qua tham số out
            if (int.TryParse(input, out int parsedNumber))
            {
                Console.WriteLine($" -> Phân tích thành công: {parsedNumber}");
            }
            else
            {
                // parsedNumber sẽ là 0 nếu TryParse thất bại
                Console.WriteLine(" -> Phân tích thất bại.");
            }
        }

    }
```

```
    static void Main(string[] args)
        {
            double numA = 10.0;
            double numB = 2.0;
            double divisionResult; // Không cần khởi tạo vì sẽ nhận giá trị từ out

            // Khi gọi phương thức có out, phải dùng từ khóa out trước đối số
            if (TryDivide(numA, numB, out divisionResult))
            {
                Console.WriteLine($"{numA} / {numB} = {divisionResult}"); // 10.0 / 2.0 = 5
            }
            else
            {
                Console.WriteLine($"Không thể chia {numA} cho {numB}");
            }

            numB = 0.0;
             if (TryDivide(numA, numB, out divisionResult))
            {
                Console.WriteLine($"{numA} / {numB} = {divisionResult}");
            }
            else
            {
                Console.WriteLine($"Không thể chia {numA} cho {numB}"); // In ra dòng này
            }

            // Dùng TryParse
            ParseInput("123");
            ParseInput("abc");
            ParseInput("-50");
        }
    }
```

- **Ví dụ 5**: Tham số params (Số lượng đối số thay đổi)

```
using System;

namespace ParamsParameter
{
    class Program
    {
        // Phương thức tính tổng của một dãy số int dùng params
        // 'numbers' sẽ là một mảng int bên trong phương thức
        static int SumAll(params int[] numbers)
        {
            Console.WriteLine($"\nĐang tính tổng của {numbers.Length} số...");
            int total = 0;
            foreach (int num in numbers)
            {
                total += num;
            }
            return total;
        }

        static void Main(string[] args)
        {
            // Gọi với nhiều đối số riêng lẻ
            int sum1 = SumAll(1, 2, 3);
            Console.WriteLine($"Tổng 1, 2, 3 là: {sum1}"); // 6

            // Gọi với nhiều đối số hơn
            int sum2 = SumAll(10, 20, 30, 40, 50);
            Console.WriteLine($"Tổng 10->50 là: {sum2}"); // 150

            // Gọi mà không có đối số nào (numbers sẽ là mảng rỗng)
            int sum3 = SumAll();
            Console.WriteLine($"Tổng khi không có số nào: {sum3}"); // 0

            // Cũng có thể truyền một mảng int đã có sẵn
            int[] myNumbers = { 5, 5, 5, 5 };
            int sum4 = SumAll(myNumbers);
             Console.WriteLine($"Tổng của mảng myNumbers: {sum4}"); // 20
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 6**: Đệ quy (Tính giai thừa)

```
using System;

namespace RecursionExample
{
    class Program
    {
        // Phương thức đệ quy tính N! (N giai thừa)
        // N! = N * (N-1) * (N-2) * ... * 1
        // 0! = 1 (Đây là trường hợp cơ sở)
        static long Factorial(int n)
        {
            Console.WriteLine($" -> Đang tính Factorial({n})");
        // Điều kiện dừng (base case)
        if (n < 0)
        {
             Console.WriteLine(" (!) Không tính giai thừa số âm.");
             return -1; // Hoặc throw exception
        }
        else if (n == 0)
        {
             Console.WriteLine(" <- Đạt trường hợp cơ sở Factorial(0), trả về 1");
             return 1;
        }
        else
        {
            // Bước đệ quy: gọi lại chính nó với giá trị nhỏ hơn
            long result = n * Factorial(n - 1);
            Console.WriteLine($" <- Tính xong Factorial({n}), kết quả: {result}");
            return result;
        }
    }
```

```
    static void Main(string[] args)
        {
            int number = 5;
            long fact = Factorial(number);
            if(fact >= 0)
            {
                Console.WriteLine($"\n{number}! = {fact}"); // 5! = 120
            }

            // Thử với số âm
            Factorial(-3);
        }
    }
}
```

- **Ví dụ 7**: Nạp chồng phương thức (Method Overloading)

```
using System;

namespace OverloadingExample
{
    class Program
    {
        // Phiên bản Add cho 2 số int
        static int Add(int a, int b)
        {
            Console.WriteLine("Đang gọi Add(int, int)");
            return a + b;
        }

// Phiên bản Add cho 3 số int
static int Add(int a, int b, int c)
{
    Console.WriteLine("Đang gọi Add(int, int, int)");
    return a + b + c;
}

// Phiên bản Add cho 2 số double
static double Add(double a, double b)
{
    Console.WriteLine("Đang gọi Add(double, double)");
    return a + b;
}

// // Lỗi! Không thể nạp chồng chỉ dựa trên kiểu trả về
// static double Add(int a, int b)
// {
//     Console.WriteLine("Phiên bản này gây lỗi");
//     return (double)(a + b);
// }
```

```
static void Main(string[] args)
        {
            Console.WriteLine($"Tổng 2 int: {Add(5, 10)}");          // Gọi Add(int,
int)
            Console.WriteLine($"Tổng 3 int: {Add(5, 10, 15)}");      // Gọi Add(int,
int, int)
            Console.WriteLine($"Tổng 2 double: {Add(3.5, 2.1)}");    // Gọi
Add(double, double)

            // Trình biên dịch tự ép kiểu nếu có thể và không bị nhập nhằng
            Console.WriteLine($"Tổng int và double: {Add(5, 2.5)}"); // Gọi
Add(double, double) vì 5 có thể ép kiểu thành double
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 8**: Tham số tùy chọn và Đối số được đặt tên

```
using System;

namespace OptionalNamedArgs
{
    class Program
    {
        // power là tham số tùy chọn, mặc định là 1
        // prefix cũng tùy chọn, mặc định là "Kết quả:"
        static void CalculateAndPrint(int number, string prefix = "Kết quả:", int power =
1)
        {
            long result = number; // Dùng long phòng trường hợp lũy thừa lớn
            if(power > 1)
            {
                // Tính lũy thừa đơn giản (chưa tối ưu)
                for(int i=1; i < power; i++){
                    result *= number;
                }
            } else if (power == 0) {
                result = 1;
            } else if (power < 0) {
                Console.WriteLine(" (!) Không hỗ trợ lũy thừa âm trong ví dụ này.");
                result = -1; // Giá trị báo lỗi
            }

             if(result != -1)
                Console.WriteLine($"{prefix} {number}^{power} = {result}");
        }
```

```
        static void Main(string[] args)
        {
            // 1. Chỉ cung cấp tham số bắt buộc (number)
            CalculateAndPrint(10); // prefix="Kết quả:", power=1 => Kết quả: 10^1 =
10

            // 2. Cung cấp number và prefix (theo thứ tự)
            CalculateAndPrint(5, "Giá trị:"); // power=1 => Giá trị: 5^1 = 5

            // 3. Cung cấp cả 3 tham số (theo thứ tự)
            CalculateAndPrint(3, "3 mũ 4:", 4); // => 3 mũ 4: 3^4 = 81

            // 4. Dùng đối số được đặt tên để chỉ cung cấp number và power (bỏ qua
prefix)
            CalculateAndPrint(number: 2, power: 10); // prefix dùng mặc định => Kết
quả: 2^10 = 1024

            // 5. Dùng đối số được đặt tên không theo thứ tự
            CalculateAndPrint(power: 3, number: 7, prefix: "Lập phương:"); // =>
Lập phương: 7^3 = 343
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 9**: Tham số string[] args của Main

```
using System;

namespace MainArgsExample
{
    class Program
    {
        // Để chạy ví dụ này, bạn cần biên dịch ra file .exe
        // Sau đó mở command prompt (cmd) và chạy: ten_file.exe arg1 "argument 2" arg3
        static void Main(string[] args) // args sẽ chứa các tham số dòng lệnh
        {
            Console.WriteLine($"Chương trình nhận được {args.Length} tham số dòng lệnh.");

            if (args.Length > 0)
            {
                Console.WriteLine("Các tham số đó là:");
                int index = 0;
                foreach (string arg in args)
                {
                    Console.WriteLine($"  args[{index}]: {arg}");
                    index++;
                }
            }
            else
            {
                Console.WriteLine("Không có tham số nào được truyền vào.");
                Console.WriteLine("Thử chạy lại với tham số, ví dụ: MyProgram.exe hello world 123");
            }
        }
    }
}
```

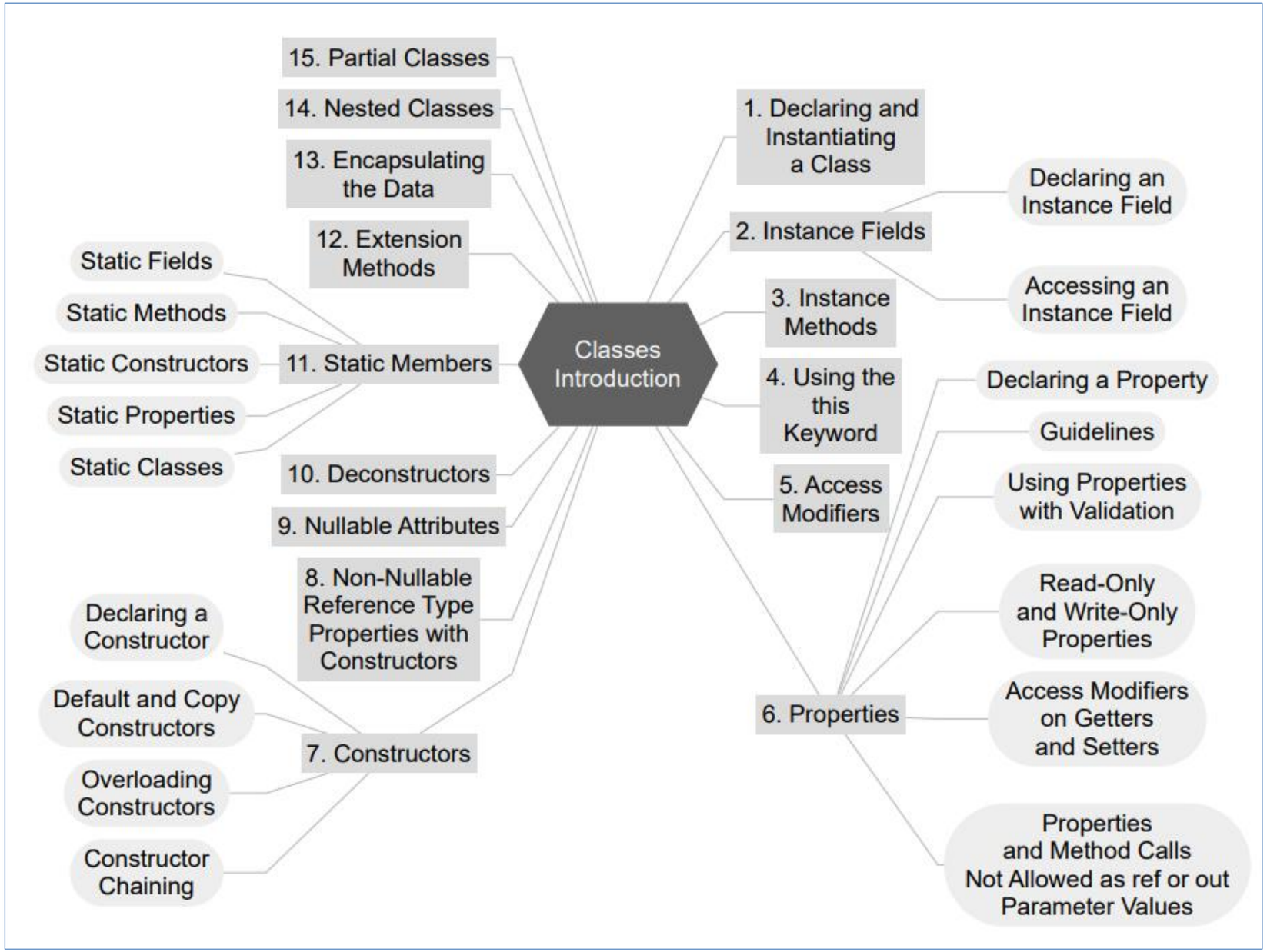
2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 10**: Xử lý ngoại lệ cơ bản try-catch

```
using System;

namespace TryCatchExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Nhập tuổi của bạn: ");
            // Console.ReadLine() đọc input từ người dùng dưới dạng string
            // Tạm gán cứng để minh họa:
            string inputAge = "двадцать"; // Tiếng Nga nghĩa là 20 -> sẽ gây lỗi FormatException
            // string inputAge = "30"; // Thử với input đúng

            try
            {
                // int.Parse cố gắng chuyển đổi string thành int
                // Nếu input không phải là số hợp lệ, nó sẽ ném ra FormatException
                int age = int.Parse(inputAge);

                Console.WriteLine($"Tuổi của bạn là: {age}");
                // Các xử lý khác nếu Parse thành công...
            }
            catch (FormatException fe) // Bắt cụ thể lỗi FormatException
            {
                Console.WriteLine("Lỗi! Dữ liệu nhập vào không phải là một số nguyên hợp lệ.");
                Console.WriteLine($"Chi tiết lỗi (dành cho dev): {fe.Message}");
            }
            catch (OverflowException oe) // Bắt lỗi nếu số quá lớn/nhỏ cho int
            {
                Console.WriteLine("Lỗi! Số bạn nhập quá lớn hoặc quá nhỏ.");
                Console.WriteLine($"Chi tiết lỗi (dành cho dev): {oe.Message}");
            }
            catch (Exception ex) // Bắt mọi loại lỗi khác chưa được xử lý ở trên
            {
                Console.WriteLine("Đã có lỗi không xác định xảy ra!");
                Console.WriteLine($"Chi tiết lỗi (dành cho dev): {ex.Message}");
            }
            finally
            {
                Console.WriteLine("\nKhối finally luôn được thực thi, dù có lỗi hay không.");
            }

            Console.WriteLine("Chương trình tiếp tục chạy sau khối try-catch...");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 6**



2. LẬP TRÌNH C# CĂN BẢN

- **Lập trình Hướng đối tượng (OOP - Object-Oriented Programming)**: Là một phương pháp lập trình lấy "đối tượng" làm trung tâm. Mỗi đối tượng trong thế giới thực (như Sinh viên, Xe hơi, Tài khoản ngân hàng) có thể được mô hình hóa trong code.
  - **Đối tượng (Object)**: Một thực thể cụ thể có:
    - **Trạng thái (State)**: Các dữ liệu, đặc điểm mô tả đối tượng (ví dụ: tên, tuổi, màu sắc, số dư). Thường được lưu trữ trong các fields hoặc properties.
    - **Hành vi (Behavior)**: Các hành động, thao tác mà đối tượng có thể thực hiện (ví dụ: Sinh viên có thể đăng ký môn học, Xe hơi có thể chạy, Tài khoản có thể nạp/rút tiền). Thường được định nghĩa bằng các methods.
  - **Lớp (Class)**: Là một "bản thiết kế" hoặc "khuôn mẫu" định nghĩa cấu trúc (trạng thái) và hành vi chung cho một loại đối tượng. Từ một lớp, bạn có thể tạo ra nhiều đối tượng (thể hiện - instance) cụ thể. Ví dụ: từ lớp SinhVien, bạn tạo ra các đối tượng sinh viên cụ thể như sv1, sv2

2. LẬP TRÌNH C# CĂN BẢN

- **Khai báo Lớp**:
  - Cú pháp: [access_modifier] class TenLop { // Định nghĩa các thành viên (members) ở đây }
  - access_modifier: (Ví dụ: public, internal - mặc định) Kiểm soát xem lớp này có thể được sử dụng từ đâu.
  - Thường dùng public cho các lớp chính.TenLop: Đặt tên theo quy tắc PascalCase.
- **Tạo Đối tượng (Instantiating a Class):**
  - Sử dụng từ khóa new và gọi constructor của lớp.
  - Cú pháp: TenLop tenBienDoiTuong = new TenLop();tenBienDoiTuong: Biến tham chiếu, trỏ đến đối tượng được tạo ra trong bộ nhớ (Heap).

- **Thành viên của Lớp (Class Members)**:
  - Fields (Trường): Là các biến được khai báo trực tiếp bên trong lớp, dùng để lưu trữ trạng thái (dữ liệu) của đối tượng. Theo nguyên tắc đóng gói, fields thường được khai báo là private.
    - Ví dụ: private string _name;, private int _age; (Thường có dấu _ ở đầu để phân biệt với property hoặc parameter).
  - Methods (Phương thức): Định nghĩa hành vi của đối tượng. Là các hàm được khai báo bên trong lớp (không có static). Chúng có thể truy cập và thao tác trên các field của chính đối tượng đó.
    - Ví dụ: public void DisplayInfo() { ... }, public bool Withdraw(decimal amount) { ... }
    - Gọi phương thức thông qua đối tượng: tenBienDoiTuong.TenPhuongThuc();

2. LẬP TRÌNH C# CĂN BẢN

- **Từ khóa this**: Bên trong một phương thức hoặc constructor không phải static của lớp, this là một tham chiếu đến chính đối tượng hiện tại đang thực thi phương thức đó.
  - Công dụng:
    - Phân biệt giữa field của lớp và tham số/biến cục bộ có cùng tên: this._name = name;
    - Gọi một constructor khác từ bên trong một constructor (constructor chaining): public Student(string id) : this(id, "Unknown") { ... } (Gọi constructor nhận id và name).

- **Access Modifiers (Bổ trợ Truy cập)**: Kiểm soát mức độ truy cập đến lớp và các thành viên của nó từ các phần khác của chương trình.
  - public: Truy cập được từ mọi nơi. Dùng cho các thành viên mà bạn muốn thế giới bên ngoài tương tác (methods, properties, constructors).
  - private: Chỉ truy cập được từ bên trong chính lớp đó. Dùng để che giấu dữ liệu (fields) và các chi tiết cài đặt bên trong (helper methods). Đây là mức truy cập mặc định cho thành viên của lớp.
  - protected: (Sẽ học ở Chương 6 - Kế thừa) Giống private, nhưng cho phép lớp con truy cập.
  - internal: Truy cập được từ mọi nơi bên trong cùng một assembly (project). Đây là mức truy cập mặc định cho lớp (khi không chỉ định). (Còn protected internal, private protected - ít dùng hơn).
  - Nguyên tắc Đóng gói (Encapsulation): Là một trong các trụ cột của OOP. Khuyến khích che giấu trạng thái bên trong của đối tượng (private fields) và chỉ cung cấp các cách thức truy cập có kiểm soát thông qua public methods và properties. Lợi ích: bảo vệ dữ liệu, dễ thay đổi cài đặt bên trong mà không ảnh hưởng code bên ngoài.

2. LẬP TRÌNH C# CĂN BẢN

- **Properties (Thuộc tính)**: Cách thức chuẩn và linh hoạt để truy cập dữ liệu của đối tượng (thay vì truy cập trực tiếp field). Cung cấp cơ chế kiểm soát việc đọc (get) và ghi (set) dữ liệu.
  - Cú pháp đầy đủ (Full Property):

```
private KiểuDữLiệu _backingField; // Field ẩn private
public KiểuDữLiệu TenThuocTinh // Property public
{
    get { /* Logic để trả về giá trị, thường là return _backingField; */ }
    set { /* Logic để gán giá trị mới, có thể validation. Dùng từ khóa 'value' */
          // Ví dụ: if (value > 0) _backingField = value;
        }
}
```

  - Thuộc tính Tự động (Auto-Implemented Property): Cách viết gọn khi không cần logic tùy chỉnh trong get hoặc set. Trình biên dịch sẽ tự động tạo ra backing field ẩn.

```
public KiểuDữLiệu TenThuocTinh { get; set; }
// Chỉ đọc: public KiểuDữLiệu TenThuocTinh { get; }
// Chỉ gán lúc khởi tạo: public KiểuDữLiệu TenThuocTinh { get; init; }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Constructors (Phương thức khởi tạo)**: Phương thức đặc biệt được gọi tự động khi bạn dùng new để tạo đối tượng. Mục đích chính là khởi tạo trạng thái ban đầu cho đối tượng.
  - Đặc điểm:
    - Tên trùng với tên lớp.
    - Không có kiểu trả về (kể cả void).
    - Có thể có tham số hoặc không.
    - Có thể được nạp chồng (overload).
  - Default Constructor: Nếu bạn không tự định nghĩa bất kỳ constructor nào, trình biên dịch sẽ tự động tạo một constructor không tham số và không làm gì cả. Nếu bạn định nghĩa bất kỳ constructor nào (dù có tham số hay không), constructor mặc định này sẽ không được tạo tự động nữa.
  - Constructor Chaining: Dùng this(danh_sách_đối_số) để gọi một constructor khác trong cùng lớp. Phải là câu lệnh đầu tiên trong thân constructor.

2. LẬP TRÌNH C# CĂN BẢN

- **Static Members (Thành viên tĩnh)**: Thuộc về lớp chứ không thuộc về một đối tượng cụ thể nào. Được chia sẻ chung giữa tất cả các đối tượng của lớp đó.
  - Static Fields: Biến tĩnh, lưu trữ dữ liệu chung cho cả lớp. Ví dụ: bộ đếm số lượng đối tượng đã tạo.
  - Static Methods: Phương thức tĩnh, được gọi trực tiếp thông qua tên lớp: TenLop.PhuongThucTinh(). Không thể truy cập các thành viên không phải static (instance members, bao gồm cả this) một cách trực tiếp. Main là một static method. Thường dùng cho các hàm tiện ích không phụ thuộc vào trạng thái đối tượng cụ thể.
  - Static Constructor: Một constructor đặc biệt, không có tham số, không có access modifier. Được CLR gọi tự động một lần duy nhất trước khi đối tượng đầu tiên của lớp được tạo hoặc trước khi bất kỳ thành viên tĩnh nào được truy cập. Dùng để khởi tạo các static field phức tạp.
  - Static Class: Lớp được đánh dấu là static. Lớp này chỉ có thể chứa các thành viên tĩnh và không thể được dùng để tạo đối tượng bằng new. Ví dụ: System.Console, System.Math.

- **Extension Methods (Phương thức mở rộng)**: Cho phép bạn "thêm" phương thức mới vào các kiểu dữ liệu đã có (kể cả kiểu do .NET định nghĩa như string, int) mà không cần sửa đổi mã nguồn gốc của chúng.
  - Cách khai báo:
    - Phải là phương thức static.
    - Phải nằm trong một lớp static.
    - Tham số đầu tiên phải có từ khóa this theo sau là kiểu dữ liệu bạn muốn mở rộng (this string str, this int number).
  - Cách gọi: Gọi như thể nó là một phương thức bình thường của đối tượng thuộc kiểu được mở rộng: myString.MyExtensionMethod();

2. LẬP TRÌNH C# CĂN BẢN

- **Nested Classes (Lớp lồng nhau)**: Khai báo một lớp bên trong phần thân của một lớp khác. Thường dùng khi lớp bên trong chỉ có ý nghĩa hoặc chỉ được sử dụng trong ngữ cảnh của lớp bên ngoài.
- **Partial Classes (Lớp cục bộ)**: Cho phép định nghĩa một lớp được chia thành nhiều file mã nguồn (.cs). Tất cả các phần phải được đánh dấu bằng từ khóa partial. Trình biên dịch sẽ ghép chúng lại thành một lớp duy nhất. Hữu ích cho các lớp lớn hoặc khi làm việc với code được sinh tự động (ví dụ: Windows Forms, WPF).

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Class đơn giản, Constructor, this

```
using System;

namespace SimpleClassExample
{
    // Khai báo lớp SinhVien
    public class Student
    {
        // Fields (private) - Lưu trữ trạng thái
        private string _studentId;
        private string _name;
        private int _birthYear;

        // Constructor - Phương thức khởi tạo khi tạo đối tượng 'new Student(...)'
// Dùng để nhận giá trị ban đầu và gán cho fields
public Student(string studentId, string name, int birthYear)
{
    Console.WriteLine($" -> Constructor Student({studentId}, {name}, {birthYear}) đang chạy...");
    // Dùng 'this' để phân biệt field của lớp (_studentId) và tham số (studentId)
    this._studentId = studentId;
    this._name = name;
    this._birthYear = birthYear;
}

// Instance Method (public) - Định nghĩa hành vi
public void DisplayInfo()
{
    Console.WriteLine($"--- Thông tin sinh viên ---");
    Console.WriteLine($"ID: {this._studentId}"); // Dùng 'this' ở đây là tùy chọn, vì không bị trùng tên
    Console.WriteLine($"Tên: {_name}");          // Có thể viết gọn hơn nếu không trùng tên
    Console.WriteLine($"Năm sinh: {_birthYear}");
    Console.WriteLine($"Tuổi (ước tính): {CalculateAge()}"); // Gọi một phương thức khác trong lớp
    Console.WriteLine($"-------------------------");
}
```

- **Ví dụ 1**: Class đơn giản, Constructor, this

```
// Một phương thức private chỉ dùng nội bộ trong lớp (ví dụ)
        private int CalculateAge() {
            // Lấy năm hiện tại (sẽ học kỹ hơn sau, tạm dùng cách đơn giản)
            // Cần cập nhật năm theo thời gian thực tế
            int currentYear = 2025; // <<<< CẬP NHẬT NĂM HIỆN TẠI NẾU CẦN
            return currentYear - this._birthYear;
        }
    } // Kết thúc lớp Student

    // Lớp Program chứa Main (client sử dụng lớp Student)
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tạo đối tượng sinh viên thứ nhất...");
            // Tạo đối tượng (instance) từ lớp Student bằng 'new' và gọi constructor
            Student student1 = new Student("SV001", "Nguyễn Văn An", 2005);

            Console.WriteLine("\nTạo đối tượng sinh viên thứ hai...");
            Student student2 = new Student("SV002", "Trần Thị Bình", 2004);

            Console.WriteLine("\nHiển thị thông tin sinh viên:");
            // Gọi phương thức DisplayInfo() thông qua đối tượng
            student1.DisplayInfo();
            student2.DisplayInfo();

             // // Lỗi! Không thể truy cập field private từ bên ngoài lớp
             // Console.WriteLine(student1._name);
        }
    }
}
```

- **Ví dụ 2**: Properties (Full & Auto-Implemented), Validation

```
using System;

namespace PropertiesExample
{
    public class Product
    {
        // --- Fields ---
        private string _productId; // Chỉ đọc sau khi tạo
        private string _productName;
        private decimal _unitPrice; // Giá phải > 0

        // --- Properties ---

        // 1. Property chỉ đọc (chỉ có 'get') cho ProductId
        public string ProductId
        {
            get { return _productId; }
            // Không có 'set' -> không thể thay đổi từ bên ngoài sau khi được gán ở constructor
        }

        // 2. Property đầy đủ cho ProductName (ví dụ: kiểm tra không rỗng)
        public string ProductName
        {
            get { return _productName; }
            set
            {
                // 'value' là từ khóa chứa giá trị đang được gán vào property
                if (!string.IsNullOrWhiteSpace(value)) // Kiểm tra nếu chuỗi không rỗng hoặc chỉ chứa khoảng trắng
                {
                    _productName = value;
                }
                else
                {
                    Console.WriteLine("(!) Lỗi: Tên sản phẩm không được để trống.");
                    // Có thể ném Exception ở đây (sẽ học sau)
                }
            }
        }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Properties (Full & Auto-Implemented), Validation

```
// 3. Property đầy đủ cho UnitPrice với validation
        public decimal UnitPrice
        {
            get { return _unitPrice; }
            set
            {
                if (value > 0)
                {
                    _unitPrice = value;
                }
                else
                {
                    Console.WriteLine("(!) Lỗi: Đơn giá phải lớn hơn 0.");
                }
            }
        }

        // 4. Auto-Implemented Property cho mô tả (không cần logic đặc biệt)
        public string Description { get; set; }

        // --- Constructor ---
        public Product(string productId, string initialName, decimal initialPrice)
        {
            this._productId = productId; // Gán trực tiếp field vì Property chỉ có get
            this.ProductName = initialName; // Gán qua Property để chạy validation (nếu có)
            this.UnitPrice = initialPrice;   // Gán qua Property để chạy validation
            this.Description = "Chưa có mô tả"; // Gán giá trị mặc định cho auto-property
        }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Properties (Full & Auto-Implemented), Validation

```
// --- Method ---
        public void Display() {
             Console.WriteLine($"ID: {ProductId}, Tên: {ProductName}, Giá: {UnitPrice:C}, Mô tả: {Description}");
             // :C là định dạng tiền tệ
        }
    } // Kết thúc lớp Product

    class Program
    {
        static void Main(string[] args)
        {
            Product laptop = new Product("LAP001", "Laptop XPS", 25000000m);
            laptop.Description = "Laptop mỏng nhẹ cấu hình cao";
            laptop.Display();

            Console.WriteLine("\nThử thay đổi giá trị:");
            laptop.ProductName = "   "; // Thử gán tên trống -> Báo lỗi
            laptop.UnitPrice = -100;   // Thử gán giá âm -> Báo lỗi
            // laptop.ProductId = "LAP002"; // Lỗi! Property này chỉ đọc (chỉ có get)

             Console.WriteLine("\nThông tin sau khi thử thay đổi:");
            laptop.Display(); // Tên và giá vẫn như cũ do validation thất bại

            laptop.ProductName = "Laptop Gaming ABC";
            laptop.UnitPrice = 30000000m;
             Console.WriteLine("\nThông tin sau khi thay đổi hợp lệ:");
             laptop.Display();

        }
    }
}
```

- **Ví dụ 3**: Constructor Overloading & Chaining (this)

```
using System;

namespace ConstructorOverloading
{
    public class Book
    {
        public string ISBN { get; } // Auto-property chỉ đọc
        public string Title { get; set; }
        public string Author { get; set; }
        public int PublicationYear { get; set; }

        // Constructor 1: Chỉ cần ISBN (Tên/Tác giả là "Unknown")
public Book(string isbn) : this(isbn, "Unknown Title", "Unknown Author") // Gọi Constructor 2
{
     Console.WriteLine(" -> Chạy Constructor(string isbn)");
     // Không cần làm gì thêm ở đây vì đã gọi constructor kia
}

// Constructor 2: Cần ISBN, Title, Author (Năm XB mặc định là năm hiện tại?)
 public Book(string isbn, string title, string author) : this(isbn, title, author, 2025) // Gọi Constructor 3
 {
      Console.WriteLine(" -> Chạy Constructor(string, string, string)");
 }
```

- **Ví dụ 3**: Constructor Overloading & Chaining (this)

```
// Constructor 3: Đầy đủ thông tin
        public Book(string isbn, string title, string author, int publicationYear)
        {
             Console.WriteLine(" -> Chạy Constructor(string, string, string, int) - Constructor chính");
            this.ISBN = isbn; // Gán vào property
            this.Title = title;
            this.Author = author;
            this.PublicationYear = publicationYear;
        }
         public void Display() {
             Console.WriteLine($"ISBN: {ISBN}, Title: {Title}, Author: {Author}, Year: {PublicationYear}");
         }
    }


class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tạo sách 1 (chỉ có ISBN):");
            Book book1 = new Book("978-1");
            book1.Display();

             Console.WriteLine("\nTạo sách 2 (ISBN, Title, Author):");
             Book book2 = new Book("978-2", "Lập trình C# Nâng cao", "Giáo sư X");
             book2.Display();

            Console.WriteLine("\nTạo sách 3 (Đầy đủ):");
            Book book3 = new Book("978-3", "Design Patterns", "Gang of Four", 1994);
            book3.Display();
        }
    }
}
```

- **Ví dụ 4**: Static Members & Static Class

```
using System;

namespace StaticMembersExample
{
    // --- Lớp với thành viên static ---
    public class Counter
    {
        // Static field - Dùng chung cho tất cả các đối tượng Counter (nếu có)
        // Hoặc dùng mà không cần tạo đối tượng
        private static int _totalCount = 0;

        // Instance field (ví dụ thêm vào)
public string Name { get; set; }

// Static constructor - Chạy 1 lần duy nhất
static Counter()
{
    Console.WriteLine("(!) Static constructor của Counter đang chạy...");
    _totalCount = 0; // Khởi tạo giá trị ban đầu cho static field
}

// Instance constructor
public Counter(string name)
{
    Console.WriteLine($" -> Instance constructor Counter({name}) đang chạy...");
    this.Name = name;
    _totalCount++; // Tăng biến đếm static mỗi khi tạo đối tượng mới
    Console.WriteLine($" -> _totalCount hiện tại: {_totalCount}");
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Static Members & Static Class

```
// Static method - Gọi trực tiếp từ lớp
public static int GetTotalCount()
{
     // Static method không thể truy cập instance member 'Name' trực tiếp
     // Console.WriteLine(this.Name); // Lỗi!
    return _totalCount;
}

// Instance method
 public void PrintCurrentCount() {
      Console.WriteLine($"Đối tượng '{this.Name}' thấy _totalCount = {Counter._totalCount}"); // Truy cập static field từ instance method
     }
}

// --- Static class ---
// Lớp tiện ích chỉ chứa thành viên static, không thể tạo đối tượng 'new'
public static class MathHelper
{
    public static double PI = 3.14159; // Static field

    // Static method
    public static double CalculateCircleArea(double radius)
    {
        if (radius < 0) return 0;
        // Math.Pow cũng là static method từ static class Math
        return PI * Math.Pow(radius, 2);
    }
}
```

- **Ví dụ 4**: Static Members & Static Class

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Bắt đầu chương trình...");
            // Truy cập static method/field mà không cần tạo đối tượng
            Console.WriteLine($"Số bộ đếm ban đầu: {Counter.GetTotalCount()}"); // Static constructor chạy ngầm ở đây
            Console.WriteLine($"Số PI từ MathHelper: {MathHelper.PI}");
            Console.WriteLine($"Diện tích hình tròn bán kính 5: {MathHelper.CalculateCircleArea(5)}");

            Console.WriteLine("\nTạo đối tượng Counter:");
            Counter c1 = new Counter("Bộ đếm A");
            Counter c2 = new Counter("Bộ đếm B");

            Console.WriteLine($"\nSố bộ đếm sau khi tạo 2 đối tượng: {Counter.GetTotalCount()}"); // Giá trị là 2

             c1.PrintCurrentCount(); // Đối tượng A thấy giá trị static là 2
             c2.PrintCurrentCount(); // Đối tượng B cũng thấy giá trị static là 2

            // MathHelper mh = new MathHelper(); // Lỗi! Không thể tạo đối tượng từ static class
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 5**: Extension Method

```
using System;
// Extension method thường được đặt trong namespace riêng hoặc chung
namespace ExtensionMethodExample
{
    // Lớp static chứa extension method
    public static class StringExtensions
    {
        // Phương thức mở rộng cho kiểu 'string'
        // 'this string str' cho biết phương thức này mở rộng cho kiểu string
        // 'str' sẽ là chuỗi mà phương thức được gọi trên đó
        public static int WordCount(this string str)
        {
            if (string.IsNullOrWhiteSpace(str))
            {
                return 0;
            }
            // Tách chuỗi thành các từ dựa trên khoảng trắng, loại bỏ các phần tử rỗng
            string[] words = str.Split(new char[] { ' ', '.', ',', '!', '?' },
                                       StringSplitOptions.RemoveEmptyEntries);
            return words.Length;
        }
         // Một extension method khác ví dụ
         public static string Reverse(this string str)
        {
             if (string.IsNullOrEmpty(str)) return str;
             char[] charArray = str.ToCharArray();
             Array.Reverse(charArray);
             return new string(charArray);
        }
} // Kết thúc lớp static
```

- **Ví dụ 5**: Extension Method

```
class Program
    {
        static void Main(string[] args)
        {
            string mySentence = "Đây là một câu ví dụ để đếm từ.";

            // Gọi extension method như thể nó là method của string
            int count = mySentence.WordCount();
            Console.WriteLine($"Câu: \"{mySentence}\"");
            Console.WriteLine($"Số từ: {count}"); // Output: 8

            string original = "Hello";
            string reversed = original.Reverse(); // Gọi extension method Reverse
            Console.WriteLine($"Chuỗi gốc: {original}");
            Console.WriteLine($"Chuỗi đảo ngược: {reversed}"); // Output: olleH
        }
    }
}
```

- **Ví dụ 6**: Nested Class & Partial Class

```
// ---- File 1: Student_Main.cs (Phần chính của lớp Student) ----
using System;
namespace NestedPartialExample
{
    // Đánh dấu lớp Student là partial
    public partial class Student
    {
        public string StudentId { get; set; }
        public string Name { get; set; }

        // Danh sách các môn học đã đăng ký (dùng lớp lồng nhau)
        private Enrollment[] _enrollments = new Enrollment[10]; // Giả sử tối đa 10 môn
        private int _enrollmentCount = 0;

        public Student(string id, string name) {
            this.StudentId = id;
            this.Name = name;
        }

        // Lớp Enrollment được định nghĩa lồng bên trong lớp Student
        // Có thể là public hoặc private tùy nhu cầu truy cập từ bên ngoài Student
        public class Enrollment
        {
            public string CourseCode { get; set; }
            public string CourseName { get; set; }
            public double? Grade { get; set; } // Điểm có thể là null nếu chưa có

            public Enrollment(string code, string name) {
                this.CourseCode = code;
                this.CourseName = name;
                this.Grade = null;
            }

            public void PrintEnrollmentInfo() {
                string gradeStr = Grade.HasValue ? Grade.Value.ToString() : "Chưa
có điểm";
                Console.WriteLine($"   - Môn: {CourseName} ({CourseCode}), Điểm:
{gradeStr}");
            }
        } // Kết thúc nested class Enrollment
    } // Kết thúc phần 1 của partial class Student
}
```

- **Ví dụ 6**: Nested Class & Partial Class

```
// ---- File 2: Student_EnrollmentMethods.cs (Phần xử lý đăng ký của Student) ----
namespace NestedPartialExample
{
     // Phần còn lại của lớp Student cũng phải đánh dấu partial
    public partial class Student
    {
         // Phương thức thuộc lớp Student để thêm đăng ký môn học
        public bool AddEnrollment(string courseCode, string courseName)
        {
            if (_enrollmentCount < _enrollments.Length)
            {
                // Tạo đối tượng từ lớp lồng nhau Enrollment
                Enrollment newEnrollment = new Enrollment(courseCode, courseName);
                _enrollments[_enrollmentCount] = newEnrollment;
                _enrollmentCount++;
                Console.WriteLine($" -> {Name} đã đăng ký môn {courseName}");
                return true;
            }
            else
            {
                Console.WriteLine($"(!) Lỗi: {Name} đã đăng ký tối đa số môn.");
                return false;
            }
        }
```

```
        public void DisplayAllEnrollments()
        {
            Console.WriteLine($"\nDanh sách môn học của {Name} ({StudentId}):");
            if (_enrollmentCount == 0) {
                Console.WriteLine("  (Chưa đăng ký môn nào)");
                return;
            }
            for (int i = 0; i < _enrollmentCount; i++)
            {
                // Gọi phương thức của đối tượng thuộc lớp lồng nhau
                _enrollments[i].PrintEnrollmentInfo();
            }
        }
    } // Kết thúc phần 2 của partial class Student
}
```

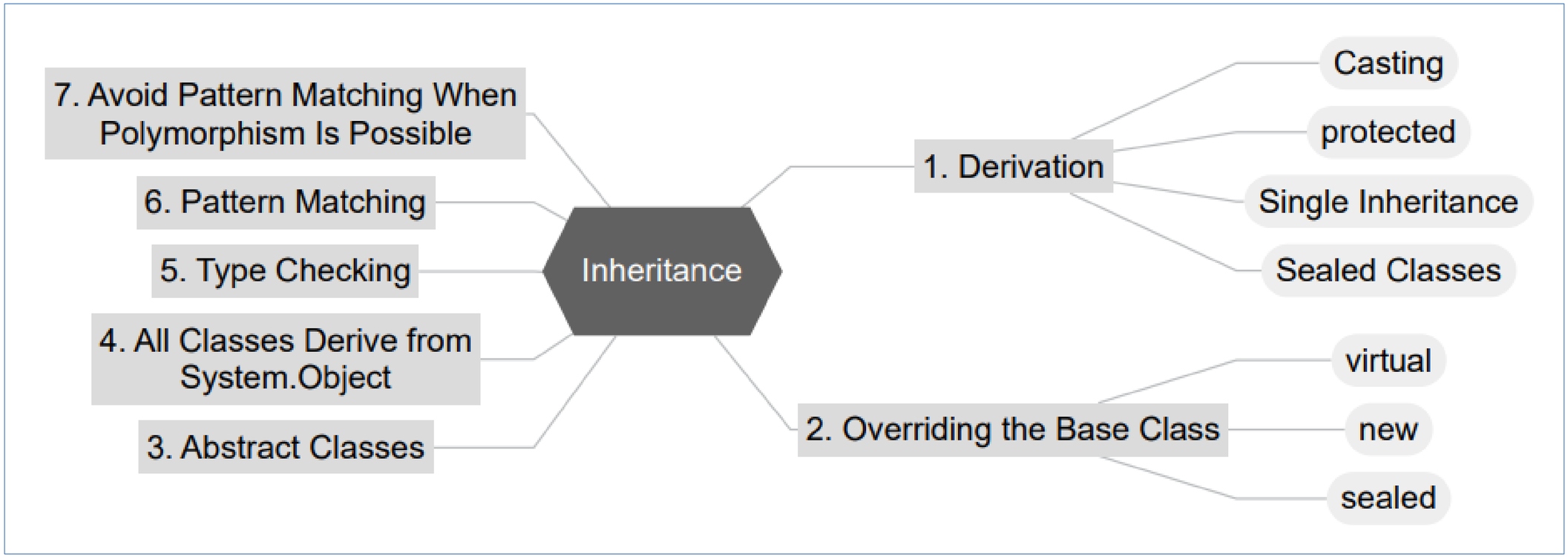
2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 6**: Nested Class & Partial Class

```
// ---- File 3: Program.cs (Sử dụng lớp Student) ----
namespace NestedPartialExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Student sv1 = new Student("S101", "Mai");

            // Thêm đăng ký môn học (gọi phương thức từ phần 2 của partial class)
            sv1.AddEnrollment("CS101", "Nhập môn Lập trình");
            sv1.AddEnrollment("MA101", "Toán cao cấp A1");

            // Hiển thị danh sách môn học
            sv1.DisplayAllEnrollments();

            // // Tạo đối tượng Enrollment trực tiếp (nếu Enrollment là public)
            // Student.Enrollment enrollment = new Student.Enrollment("PE101", "Giáo dục thể chất");
            // enrollment.PrintEnrollmentInfo();
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 7**



- **Kế thừa (Inheritance / Derivation)**:
  - Là cơ chế cho phép một lớp mới (gọi là lớp con - derived class, child class, subclass) được tạo ra dựa trên một lớp đã có (gọi là lớp cha - base class, parent class, superclass).
  - Lớp con sẽ thừa hưởng các thành viên (fields, methods, properties) không phải private của lớp cha.
  - Mối quan hệ "is-a": Kế thừa tạo ra mối quan hệ "là một loại của". Ví dụ: nếu lớp Dog kế thừa từ lớp Animal, thì một Dog là một loại Animal.
  - Cú pháp: Dùng dấu hai chấm : sau tên lớp con và theo sau là tên lớp cha.

```
class Animal { /* ... members ... */ }
class Dog : Animal { /* ... members của Dog + thừa hưởng từ Animal ... */
}
```

- **Kế thừa (Inheritance / Derivation)**:
  - Lợi ích:
    - Tái sử dụng mã (Code Reusability): Tránh viết lại code đã có ở lớp cha.
    - Tổ chức và phân cấp: Xây dựng cấu trúc lớp rõ ràng, dễ quản lý.
    - Đa hình (Polymorphism): (Sẽ thấy rõ hơn ở chương sau) Khả năng đối tượng thể hiện nhiều hình thái, cho phép đối xử với đối tượng lớp con như đối tượng lớp cha.
  - Truy cập thành viên lớp cha: Lớp con có thể truy cập các thành viên public và protected của lớp cha. Thành viên private của lớp cha thì không thể truy cập trực tiếp từ lớp con.
  - Từ khóa protected: Giống private, nhưng cho phép các lớp con (derived classes) truy cập được thành viên đó.

- **Gọi Constructor của lớp Cha (base)**:
  - Khi một đối tượng lớp con được tạo, constructor của lớp cha luôn được gọi trước constructor của lớp con.
  - Mặc định: Nếu bạn không chỉ định gì, constructor không tham số của lớp cha sẽ được gọi ngầm trước khi thân constructor lớp con thực thi. Nếu lớp cha không có constructor không tham số (hoặc nó là private), bạn sẽ gặp lỗi biên dịch.
  - Chỉ định tường minh: Để gọi một constructor có tham số cụ thể của lớp cha, bạn dùng từ khóa base ngay sau khai báo tham số của constructor lớp con và trước dấu {.

```
class Derived : Base
{
    public Derived(int derivedParam, string baseParam) : base(baseParam)
// Gọi constructor của Base nhận string
    {
        // Code của constructor Derived
    }
}
```

- **Ghi đè thành viên lớp Cha (Overriding)**:
  - Cho phép lớp con cung cấp một cài đặt (implementation) riêng, cụ thể hơn cho một phương thức hoặc thuộc tính đã được định nghĩa ở lớp cha.
    - **Bước 1 (Lớp Cha)**: Đánh dấu thành viên (method/property) mà bạn muốn cho phép ghi đè bằng từ khóa virtual.

```
class Animal {
    public virtual void MakeSound() { Console.WriteLine("Animal makes a sound"); }
}
```

    - **Bước 2 (Lớp Con)**: Sử dụng từ khóa override trước khai báo thành viên ở lớp con để cung cấp cài đặt mới. Signature (tên, kiểu trả về, tham số) phải khớp chính xác với thành viên virtual ở lớp cha.

```
class Dog : Animal {
    public override void MakeSound() { Console.WriteLine("Woof! Woof!"); }
}
```

  - **Gọi cài đặt gốc của lớp cha**: Bên trong phương thức override của lớp con, bạn có thể gọi phiên bản gốc của phương thức ở lớp cha bằng base.TenPhuongThuc().

```
public override void MakeSound() {
    base.MakeSound(); // Gọi "Animal makes a sound" trước
    Console.WriteLine("Specifically, a dog goes Woof!");
}
```

- **Lớp và Thành viên Trừu tượng (Abstract Classes and Members):**
  - **abstract class**: Là lớp được đánh dấu bằng abstract. Lớp này không thể được dùng để tạo đối tượng trực tiếp (new AbstractClass() sẽ lỗi). Nó chỉ đóng vai trò làm lớp cha cho các lớp khác kế thừa. Lớp trừu tượng có thể chứa cả thành viên trừu tượng và thành viên thông thường (có cài đặt).
  - **abstract member**: Là phương thức hoặc thuộc tính được đánh dấu abstract và không có phần thân cài đặt (chỉ có khai báo kết thúc bằng dấu ;). Thành viên abstract chỉ có thể tồn tại bên trong một abstract class. Lớp con kế thừa từ lớp trừu tượng bắt buộc phải override và cung cấp cài đặt cho tất cả các thành viên abstract mà nó kế thừa. Thành viên abstract mặc nhiên là virtual.

```
abstract class Shape {
    public abstract double CalculateArea(); // Phải được override bởi lớp con
    public void Display() { Console.WriteLine("This is a shape."); } // Phương thức thường
}
class Circle : Shape {
    public double Radius { get; set; }
    public override double CalculateArea() { return Math.PI * Radius * Radius; } // Bắt buộc override
}
```

Mục đích: Định nghĩa một "bộ khung" hoặc "hợp đồng" chung mà các lớp con phải tuân theo, đảm bảo chúng có các hành vi cốt lõi cần thiết.

- **Che giấu Thành viên (**Member Hiding using **new)**:
  - Nếu lớp con định nghĩa một thành viên có cùng tên với một thành viên không phải virtual ở lớp cha, bạn nên dùng từ khóa **new** để chỉ rõ rằng bạn đang cố ý tạo một thành viên mới độc lập, che giấu đi thành viên của lớp cha.
  - Khác biệt với Overriding: Hiding không liên quan đến virtual/override. Việc phương thức nào được gọi sẽ phụ thuộc vào kiểu của biến tham chiếu tại thời điểm biên dịch, không phải kiểu của đối tượng thực tế tại thời điểm chạy (như overriding).
  - Nên tránh: Thường gây khó hiểu, nên ưu tiên dùng overriding nếu có thể thiết kế lại lớp cha với virtual.

```
class Parent { public void ShowInfo() { Console.WriteLine("Parent Info"); } }
class Child : Parent { public new void ShowInfo() { Console.WriteLine("Child Info"); }
}
// Parent p = new Child(); p.ShowInfo(); // In ra "Parent Info"
// Child c = new Child(); c.ShowInfo(); // In ra "Child Info"
```

- **Từ khóa sealed**:
  - sealed class TenLop: Ngăn chặn bất kỳ lớp nào khác kế thừa từ TenLop. Ví dụ: lớp string của .NET là sealed.
  - sealed override ReturnType MemberName(...): Đặt trên một thành viên đã được override ở lớp con để ngăn chặn các lớp con tiếp theo (cháu của lớp cha ban đầu) ghi đè lên thành viên đó nữa.

- **Kế thừa từ System.Object**:
  - Mọi lớp trong C#, dù bạn có khai báo kế thừa hay không, đều ngầm định kế thừa từ lớp cơ sở **System.Object**.
  - **System.Object** cung cấp các phương thức nền tảng quan trọng:
    - **ToString()**: Trả về biểu diễn chuỗi của đối tượng (mặc định thường là tên lớp). Nên override để cung cấp thông tin hữu ích hơn.
    - **Equals(object obj)**: So sánh xem đối tượng hiện tại có bằng đối tượng khác không (mặc định thường so sánh tham chiếu). Nên override để so sánh dựa trên giá trị trạng thái (fields/properties).
    - **GetHashCode()**: Trả về một mã hash (số nguyên) đại diện cho đối tượng. Nếu bạn override Equals, bạn bắt buộc phải override GetHashCode sao cho các đối tượng bằng nhau (Equals trả về true) phải có cùng GetHashCode.
    - **GetType()**: Trả về đối tượng Type chứa thông tin về kiểu của đối tượng hiện tại.

- **Kiểm tra và Ép kiểu (Type Checking and Casting)**:
  - Khi làm việc với kế thừa, bạn thường có biến tham chiếu kiểu lớp cha nhưng nó lại trỏ đến đối tượng của lớp con. Để sử dụng các thành viên đặc trưng của lớp con, bạn cần kiểm tra và ép kiểu.
  - **Toán tử is**: Kiểm tra xem một đối tượng có thể được coi là một kiểu cụ thể hay không (là kiểu đó, hoặc kế thừa từ kiểu đó, hoặc thực thi interface đó - sẽ học sau). Trả về true hoặc false. An toàn, không gây lỗi.

```
if (myAnimal is Dog) { Console.WriteLine("It's a dog!"); }
```

  - **Toán tử as**: Cố gắng ép kiểu đối tượng sang kiểu chỉ định. Nếu thành công, trả về tham chiếu đã được ép kiểu. Nếu không thành công (đối tượng không tương thích), trả về null. An toàn, không ném ra exception. Thường dùng khi bạn không chắc chắn và muốn xử lý trường hợp ép kiểu thất bại.C#

```
Dog dogRef = myAnimal as Dog;
if (dogRef != null) { // Kiểm tra null sau khi dùng 'as'
    dogRef.Bark();
}
```

- **Kiểm tra và Ép kiểu (Type Checking and Casting):**
  - **Ép kiểu tường minh (TypeName)object**: Ép kiểu trực tiếp. Nếu đối tượng không tương thích với TypeName tại thời điểm chạy, nó sẽ ném ra một InvalidCastException, có thể làm crash chương trình nếu không được bắt (try-catch). Chỉ nên dùng khi bạn chắc chắn về kiểu của đối tượng (ví dụ, sau khi đã kiểm tra bằng is).

```
if (myAnimal is Dog) {
    Dog specificDog = (Dog)myAnimal; // An toàn
vì đã kiểm tra 'is'
    specificDog.Bark();
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Kế thừa cơ bản, gọi base() constructor

```
using System;

namespace BasicInheritance
{
    // Lớp Cha (Base Class)
    public class Shape
    {
        public string Color { get; set; }

        // Constructor lớp cha
        public Shape(string color)
        {
            Console.WriteLine(" -> Constructor Shape(string) chạy...");
            this.Color = color;
        }

         // Constructor không tham số (ví dụ)
         public Shape() : this("Trắng") { // Gọi constructor kia với màu mặc định
             Console.WriteLine(" -> Constructor Shape() không tham số chạy...");
         }


        public void DisplayColor()
        {
            Console.WriteLine($"Màu sắc: {this.Color}");
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Kế thừa cơ bản, gọi base() constructor

```
public class Rectangle : Shape
    {
        public double Width { get; set; }
        public double Height { get; set; }

        // Constructor lớp con, gọi constructor có tham số của lớp cha Shape
        public Rectangle(string color, double width, double height) : base(color) // Gọi Shape(color)
        {
            Console.WriteLine("   -> Constructor Rectangle(string, double, double) chạy...");
            this.Width = width;
            this.Height = height;
        }

        // Constructor khác của Rectangle, gọi constructor không tham số của Shape (ngầm định hoặc tường minh)
        public Rectangle(double width, double height) // : base() // Không cần viết base() nếu gọi constructor không tham số của cha
        {
             Console.WriteLine("   -> Constructor Rectangle(double, double) chạy...");
             this.Width = width;
             this.Height = height;
             // this.Color sẽ là "Trắng" do constructor Shape() không tham số gán
        }


        public double CalculateArea()
        {
            return Width * Height;
        }

        public void DisplayRectangleInfo() {
            DisplayColor(); // Gọi phương thức kế thừa từ Shape
            Console.WriteLine($"Chiều rộng: {Width}, Chiều cao: {Height}, Diện tích: {CalculateArea()}");
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Kế thừa cơ bản, gọi base() constructor

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Tạo Hình chữ nhật 1:");
            Rectangle rect1 = new Rectangle("Xanh dương", 10.5, 5.2);
            rect1.DisplayRectangleInfo();

            Console.WriteLine("\nTạo Hình chữ nhật 2 (không truyền màu):");
            Rectangle rect2 = new Rectangle(8, 4);
            rect2.DisplayRectangleInfo();
        }
    }
}
```

- **Ví dụ 2**: Ghi đè phương thức (virtual/override), gọi base.

```
using System;

namespace MethodOverriding
{
    public class Animal
    {
        public string Name { get; set; }

        public Animal(string name) { this.Name = name; }

        // Phương thức virtual - cho phép lớp con ghi đè
        public virtual void MakeSound()
        {
            Console.Write($"{Name} tạo ra tiếng kêu chung chung.");
        }

        public void Eat() {
             Console.WriteLine($"{Name} đang ăn.");
        }
    }
```

- **Ví dụ 2**: Ghi đè phương thức (virtual/override), gọi base.

```
public class Dog : Animal
{
    public Dog(string name) : base(name) { }

    // Ghi đè phương thức MakeSound
    public override void MakeSound()
    {
        // base.MakeSound(); // Tùy chọn: Gọi cài đặt gốc của lớp cha nếu muốn
        Console.Write($"{Name} kêu 'Gâu Gâu!'"); // Cài đặt riêng của Dog
    }
    // Phương thức riêng của Dog
    public void WagTail() {
         Console.WriteLine($"{Name} đang vẫy đuôi.");
    }
}

public class Cat : Animal
    {
        public Cat(string name) : base(name) { }

        // Ghi đè phương thức MakeSound
        public override void MakeSound()
        {
            Console.Write($"{Name} kêu 'Meo Meo!'"); // Cài đặt riêng của Cat
        }
         // Phương thức riêng của Cat
        public void Purr() {
            Console.WriteLine($"{Name} đang rên gừ gừ.");
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Ghi đè phương thức (virtual/override), gọi base.

```
class Program
    {
        static void Main(string[] args)
        {
            Animal genericAnimal = new Animal("Con vật");
            Dog myDog = new Dog("Buddy");
            Cat myCat = new Cat("Whiskers");

            Console.Write("genericAnimal: "); genericAnimal.MakeSound(); Console.WriteLine(); // Gọi bản gốc
            Console.Write("myDog: "); myDog.MakeSound(); Console.WriteLine();            // Gọi bản override của Dog
            Console.Write("myCat: "); myCat.MakeSound(); Console.WriteLine();            // Gọi bản override của Cat

            Console.WriteLine("\n--- Đa hình (Polymorphism) ---");
            // Biến kiểu lớp cha có thể tham chiếu đến đối tượng lớp con
            Animal animalRef1 = myDog;
            Animal animalRef2 = myCat;

            // Khi gọi phương thức virtual/override qua tham chiếu lớp cha,
            // phiên bản override của đối tượng *thực tế* sẽ được gọi.
            Console.Write("animalRef1 (trỏ tới Dog): "); animalRef1.MakeSound(); Console.WriteLine(); // Gọi Dog.MakeSound()
            Console.Write("animalRef2 (trỏ tới Cat): "); animalRef2.MakeSound(); Console.WriteLine(); // Gọi Cat.MakeSound()

            // animalRef1.WagTail(); // Lỗi! Không gọi được phương thức riêng của Dog qua tham chiếu Animal
        }
```

- **Ví dụ 3**: Lớp trừu tượng và thành viên trừu tượng.

```
using System;

namespace AbstractClassExample
{
    // Lớp trừu tượng - không thể tạo đối tượng trực tiếp
    public abstract class Vehicle
    {
        public int Year { get; protected set; } // Chỉ lớp con có thể set
        public string Model { get; protected set; }

        protected Vehicle(string model, int year) {
            this.Model = model;
            this.Year = year;
        }

        // Phương thức trừu tượng - không có cài đặt, lớp con PHẢI override
        public abstract void StartEngine();
        public abstract void StopEngine();

        // Phương thức thường - có cài đặt, lớp con có thể kế thừa hoặc override (nếu là virtual)
        public virtual void DisplayInfo() {
            Console.WriteLine($"Model: {Model}, Year: {Year}");
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 3**: Lớp trừu tượng và thành viên trừu tượng.

```
// Lớp con kế thừa từ lớp trừu tượng
    public class Car : Vehicle
    {
        public int NumberOfDoors { get; set; }

        public Car(string model, int year, int doors) : base(model, year) {
            this.NumberOfDoors = doors;
        }

        // BẮT BUỘC phải override các phương thức abstract từ lớp cha
        public override void StartEngine()
        {
            Console.WriteLine($"Xe hơi {Model} nổ máy 'Brum Brum'.");
        }

        public override void StopEngine()
        {
            Console.WriteLine($"Xe hơi {Model} tắt máy.");
        }

        // Tùy chọn override phương thức virtual
        public override void DisplayInfo()
        {
            base.DisplayInfo(); // Gọi cài đặt gốc của Vehicle
            Console.WriteLine($"Số cửa: {NumberOfDoors}");
        }
}
```

- **Ví dụ 3**: Lớp trừu tượng và thành viên trừu tượng.

```
public class Motorcycle : Vehicle
    {
        public bool HasSidecar { get; set; }
        public Motorcycle(string model, int year, bool hasSidecar) : base(model, year) {
            this.HasSidecar = hasSidecar;
        }
        public override void StartEngine() {
            Console.WriteLine($"Xe máy {Model} nổ máy 'Reng Reng'.");
         }
        public override void StopEngine() {
            Console.WriteLine($"Xe máy {Model} tắt máy.");
        }
        public override void DisplayInfo() {
            base.DisplayInfo();
            Console.WriteLine($"Có thùng bên cạnh: {(HasSidecar ? "Có" : "Không")}");
        }
}
```

- **Ví dụ 3**: Lớp trừu tượng và thành viên trừu tượng.

```
class Program
    {
        static void Main(string[] args)
        {
            // Vehicle v = new Vehicle("Some model", 2020); // Lỗi! Không thể tạo đối tượng từ lớp abstract

            Car myCar = new Car("Toyota Camry", 2022, 4);
            Motorcycle myBike = new Motorcycle("Honda CB500", 2021, false);

            myCar.DisplayInfo();
            myCar.StartEngine();
            myCar.StopEngine();

            Console.WriteLine();

            myBike.DisplayInfo();
            myBike.StartEngine();
            myBike.StopEngine();

            Console.WriteLine("\n--- Đa hình với Abstract Class ---");
            Vehicle vehicleRef1 = myCar;
            Vehicle vehicleRef2 = myBike;

            vehicleRef1.StartEngine(); // Gọi Car.StartEngine()
            vehicleRef2.StartEngine(); // Gọi Motorcycle.StartEngine()

        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Type Checking (is) và Casting (as, ())

```
using System;
// Sử dụng lại các lớp Animal, Dog, Cat từ ví dụ 2
using MethodOverriding;

namespace TypeCastingExample
{
    class Program
    {
        static void Main(string[] args)
        {
            // Tạo một mảng chứa các đối tượng lớp con, lưu dưới dạng tham chiếu lớp cha
            Animal[] animals = new Animal[4];
animals[0] = new Dog("Buddy");
animals[1] = new Cat("Whiskers");
animals[2] = new Dog("Lucy");
animals[3] = new Animal("Generic"); // Đối tượng lớp cha

Console.WriteLine("Duyệt qua vườn thú:");
foreach (Animal currentAnimal in animals)
{
    Console.Write($" - Gặp {currentAnimal.Name}. Tiếng kêu: ");
    currentAnimal.MakeSound(); // Đa hình: Gọi đúng phiên bản override
    Console.WriteLine();

    // Kiểm tra kiểu và gọi phương thức riêng của lớp con

    // 1. Dùng 'is' và ép kiểu tường minh '()'
    if (currentAnimal is Dog)
    {
        Console.Write("   Đây là chó! ");
        Dog specificDog = (Dog)currentAnimal; // Ép kiểu an toàn vì đã check 'is'
        specificDog.WagTail();
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Type Checking (is) và Casting (as, ())

```
// 2. Dùng 'as' và kiểm tra null
                Cat specificCat = currentAnimal as Cat; // Thử ép kiểu sang Cat
                if (specificCat != null) // Nếu ép kiểu thành công (không phải null)
                {
                     Console.Write("   Đây là mèo! ");
                    specificCat.Purr();
                }

                Console.WriteLine("   ------");
            }
             // 3. Thử ép kiểu không an toàn (sẽ gây lỗi)
            Console.WriteLine("\nThử ép kiểu không an toàn:");
            Animal anotherAnimal = new Animal("Mystery");
            try {
                 // Ép kiểu Animal thành Dog khi nó không phải Dog -> InvalidCastException
                 Dog forcedDog = (Dog)anotherAnimal;
                 forcedDog.WagTail(); // Dòng này sẽ không chạy
            }
            catch (InvalidCastException ex) {
                 Console.WriteLine($"(!) Lỗi ép kiểu: {ex.Message}");
            }
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 5**: Hiding (new) và sealed

```
using System;

namespace HidingAndSealed
{
    public class Gadget
    {
        public string Name { get; set; } = "Generic Gadget";

        // Phương thức không phải virtual
        public void Display() {
            Console.WriteLine($"Gadget Display: {Name}");
        }
         // Phương thức virtual
        public virtual void Activate() {
            Console.WriteLine("Gadget activating...");
        }
    }

    public class Smartphone : Gadget
    {
        // Che giấu (hide) phương thức Display của Gadget
        public new void Display() {
            base.Display(); // Vẫn có thể gọi bản gốc của cha nếu muốn
            Console.WriteLine($"Smartphone Display: Showing fancy UI for {Name}");
        }

         // Override phương thức virtual Activate
         // Đánh dấu là sealed để lớp con của Smartphone (nếu có) không override được nữa
        public sealed override void Activate() {
            Console.WriteLine($"Smartphone ({Name}) activating with touch screen...");
        }
    }
```

- **Ví dụ 5**: Hiding (new) và sealed

```
// Lớp này sealed, không thể kế thừa từ nó
    public sealed class FeaturePhone : Gadget
    {
         public override void Activate() {
              Console.WriteLine("Feature Phone activating with keypad...");
         }
    }

    // // Lỗi! Không thể kế thừa từ sealed class 'FeaturePhone'
    // public class BasicPhone : FeaturePhone { }
```

```
// // Lỗi! Không thể override sealed method 'Activate' từ Smartphone
// public class GamingPhone : Smartphone {
//     public override void Activate() { }
// }
```

- **Ví dụ 5**: Hiding (new) và sealed

```
class Program
    {
        static void Main(string[] args)
        {
            Gadget g1 = new Gadget();
            Smartphone s1 = new Smartphone { Name = "Galaxy S30" };
            Gadget g2 = s1; // Tham chiếu Gadget trỏ tới đối tượng Smartphone

            Console.WriteLine("--- Gọi Display ---");
            g1.Display(); // Gọi Gadget.Display
            s1.Display(); // Gọi Smartphone.Display (bản new)
            g2.Display(); // Gọi Gadget.Display (vì Display không virtual, gọi theo kiểu tham chiếu)

            Console.WriteLine("\n--- Gọi Activate ---");
            g1.Activate(); // Gọi Gadget.Activate
            s1.Activate(); // Gọi Smartphone.Activate (bản override)
            g2.Activate(); // Gọi Smartphone.Activate (vì Activate là virtual, gọi theo kiểu đối tượng thực tế)
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

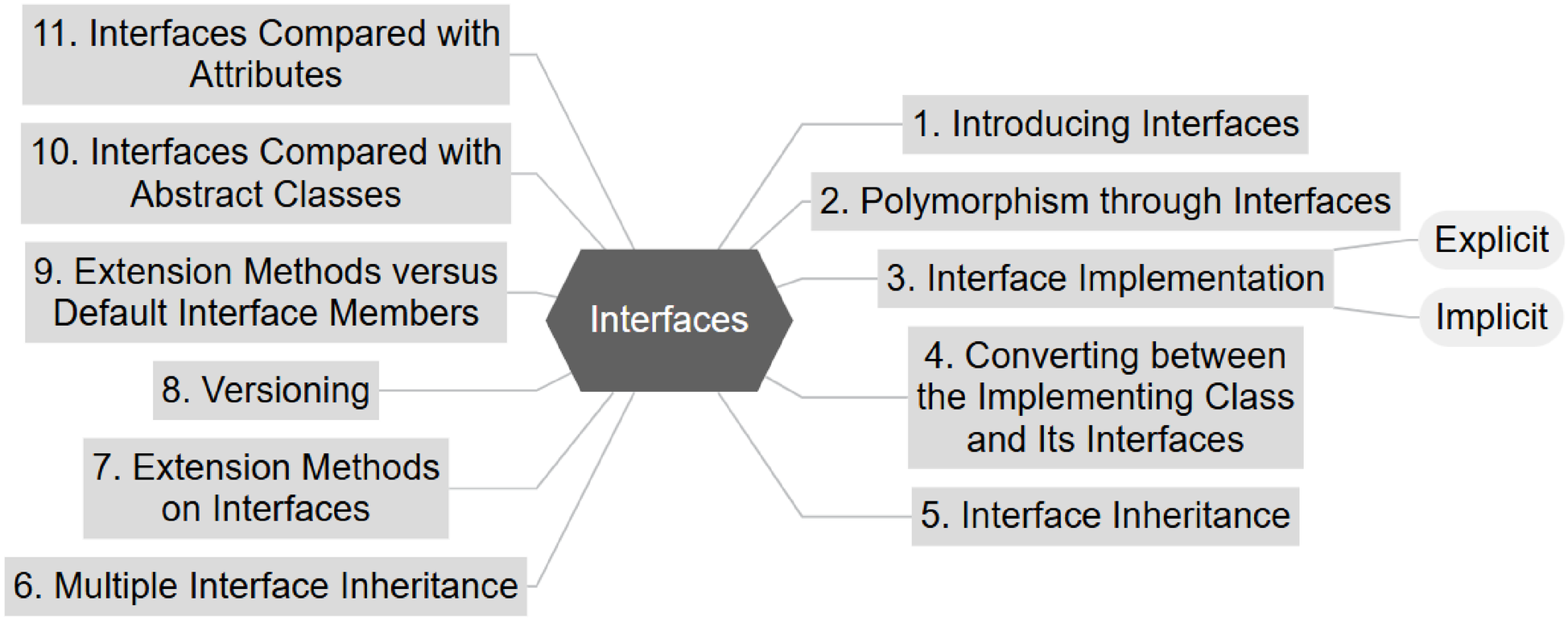
- **Ví dụ 6**: Override ToString()

```
using System;

namespace OverrideToString
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point(int x, int y) { X = x; Y = y; }

        // Override phương thức ToString() kế thừa từ System.Object
        public override string ToString()
        {
            // Trả về biểu diễn chuỗi mong muốn thay vì tên lớp mặc định
            return $"({X}, {Y})";
        }
    }
```

- **Ví dụ 6**: Override ToString()

```
class Program
    {
        static void Main(string[] args)
        {
            Point p1 = new Point(10, 20);
            Point p2 = new Point(-5, 8);

            // Khi dùng Console.WriteLine hoặc nối chuỗi, .ToString() được gọi ngầm
            Console.WriteLine("Điểm p1: " + p1); // Output: Điểm p1: (10, 20)
            Console.WriteLine($"Thông tin điểm p2: {p2}"); // Output: Thông tin điểm p2: (-5, 8)

            // So sánh với hành vi mặc định nếu không override ToString()
             object obj = new object();
             Console.WriteLine($"Object mặc định: {obj.ToString()}"); // Output: Object mặc định: System.Object
        }
    }
}
```

- **Nội dung Chương 8**



2. LẬP TRÌNH C# CĂN BẢN

- **Interface (Giao diện) là gì?**
  - Là một "bản hợp đồng" (contract) hoàn toàn trừu tượng, định nghĩa một tập hợp các thành viên công khai (public) mà một lớp (class) hoặc cấu trúc (struct) phải cung cấp cài đặt (implement) nếu nó muốn "tuân thủ" hợp đồng đó.
  - Chỉ định "Cái gì" (What), không phải "Cách làm" (How): Interface chỉ định những gì một đối tượng có khả năng làm (ví dụ: có khả năng ghi log, có khả năng kết nối), chứ không quy định cách thức thực hiện những hành động đó.

2. LẬP TRÌNH C# CĂN BẢN

- **Interface (Giao diện) là gì?**
  - Thành viên của Interface (Trước C# 8.0): Chỉ chứa khai báo (signatures) của:
    - Methods (Phương thức)
    - Properties (Thuộc tính)
    - Events (Sự kiện - sẽ học sau)
    - Indexers (Bộ chỉ mục - sẽ học sau)
    - Quan trọng: Các thành viên này không có phần thân cài đặt ({...}) và mặc định chúng là public và abstract. Bạn không cần (và không được) viết public hay abstract trước chúng.

- **Interface (Giao diện) là gì?**
  - Mối quan hệ "**Can-do**": Nếu một lớp cài đặt interface IFlyable, nó có nghĩa là lớp đó có khả năng bay (can fly). Khác với kế thừa ("is-a").
  - Cú pháp Khai báo: Tên interface thường bắt đầu bằng chữ I.

```
interface ILogger
{
    void Log(string message); // Khai báo phương thức
    string LogLevel { get; set; } // Khai báo property
}
```

- **Interface (Giao diện) là gì?**
  - **Lợi ích**:
    - Trừu tượng hóa (Abstraction): Che giấu chi tiết cài đặt, chỉ tập trung vào chức năng.
    - Kết nối lỏng lẻo (Loose Coupling): Các thành phần phụ thuộc vào interface thay vì lớp cụ thể, giúp dễ dàng thay thế cài đặt mà không ảnh hưởng nhiều đến hệ thống.
    - Đa hình (Polymorphism): Cho phép đối xử với các đối tượng từ các lớp khác nhau (miễn là chúng cài đặt cùng interface) một cách thống nhất thông qua biến kiểu interface.

- **Cài đặt Interface (Implementing an Interface)**
  - Một lớp (hoặc struct) có thể khai báo rằng nó sẽ cài đặt một hoặc nhiều interface bằng cách dùng dấu : theo sau tên lớp (hoặc tên lớp cha nếu có kế thừa) và danh sách các interface (cách nhau bởi dấu phẩy).
  - Yêu cầu: Lớp đó bắt buộc phải cung cấp phần cài đặt (implementation) public cho tất cả các thành viên được định nghĩa trong các interface mà nó khai báo cài đặt. Signature (tên, kiểu trả về, tham số) phải khớp chính xác.
  - Kế thừa đơn, Cài đặt đa interface: Một lớp chỉ có thể kế thừa từ một lớp cha, nhưng có thể cài đặt nhiều interface.

2. LẬP TRÌNH C# CĂN BẢN

- **Cài đặt Interface (Implementing an Interface)**

```
class FileLogger : ILogger // Cài đặt 1 interface
{
    public string LogLevel { get; set; } = "Info"; // Cài đặt property

    public void Log(string message) // Cài đặt method (phải là public)
    {
        // Giả lập ghi ra file
        Console.WriteLine($"FILE LOG ({LogLevel}): {message}");
    }
}

class NetworkDevice : Device, IConnectable, ISendable // Kế thừa lớp Device, cài đặt 2
interface
{
    // ... phải cài đặt tất cả thành viên của IConnectable và ISendable ...
}
```

- **Đa hình qua Interface**
  - Một biến có kiểu là interface có thể tham chiếu (trỏ) đến một đối tượng của bất kỳ lớp nào cài đặt interface đó.
  - Khi gọi phương thức thông qua biến interface, phiên bản cài đặt của lớp đối tượng thực tế sẽ được thực thi.

```
ILogger logger; // Biến kiểu interface

logger = new FileLogger(); // Trỏ tới đối tượng FileLogger
logger.Log("File logger message."); // Gọi FileLogger.Log()

logger = new ConsoleLogger(); // Trỏ tới đối tượng ConsoleLogger (giả sử có lớp
này)
logger.Log("Console logger message."); // Gọi ConsoleLogger.Log()
```

- **Kế thừa Interface (Interface Inheritance)**
  - Một interface có thể kế thừa từ một hoặc nhiều interface khác.
  - Lớp nào cài đặt interface con thì cũng phải cài đặt tất cả thành viên của các interface cha của nó.

```
interface IReadable { string Read(); }
interface IWritable { void Write(string data); }
interface IReadWrite : IReadable, IWritable { } // Kế thừa cả hai

class StreamHandler : IReadWrite // Phải cài đặt Read() và Write()
{
    public string Read() { /*...*/ return ""; }
    public void Write(string data) { /*...*/ }
}
```

- **Cài đặt Tường minh (Explicit Interface Implementation)**
  - Trường hợp sử dụng:
    - Khi một lớp cài đặt hai (hoặc nhiều) interface có thành viên trùng tên và signature. Cần phân biệt rõ cài đặt nào dành cho interface nào.
    - Khi muốn "ẩn" một thành viên của interface khỏi API công khai của lớp. Thành viên này sẽ không truy cập được trực tiếp qua biến kiểu lớp, mà chỉ truy cập được qua biến kiểu interface tương ứng.
  - Cú pháp: Không dùng access modifier (public), tên thành viên phải bao gồm đầy đủ tên interface và dấu chấm.

2. LẬP TRÌNH C# CĂN BẢN

- **Cài đặt Tường minh (Explicit Interface Implementation)**

```
interface IAction1 { void Execute(); }
interface IAction2 { void Execute(); }

class TaskRunner : IAction1, IAction2
{
    // Cài đặt tường minh cho IAction1.Execute
    void IAction1.Execute() { Console.WriteLine("Executing Action 1"); }

    // Cài đặt tường minh cho IAction2.Execute
    void IAction2.Execute() { Console.WriteLine("Executing Action 2"); }

    // Một phương thức Execute public thông thường của lớp (tùy chọn)
    public void Execute() { Console.WriteLine("Executing TaskRunner default action");}
}
// Cách gọi:
// TaskRunner runner = new TaskRunner();
// runner.Execute(); // Gọi bản public của TaskRunner
// IAction1 act1 = runner; act1.Execute(); // Gọi bản của IAction1
// IAction2 act2 = runner; act2.Execute(); // Gọi bản của IAction2
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Interface cơ bản, Cài đặt, Đa hình

```
using System;

namespace BasicInterface
{
    // 1. Định nghĩa Interface
    public interface IMovable
    {
        void Move(); // Phương thức cần cài đặt
        int CurrentSpeed { get; } // Property chỉ đọc cần cài đặt
    }

// 2. Lớp thứ nhất cài đặt Interface
public class Car : IMovable
{
    private int _speed = 0;
    public string Model { get; set; }

    public Car(string model) { Model = model; }

    // Cài đặt thành viên interface (phải public)
    public int CurrentSpeed => _speed; // Dùng biểu thức lambda cho property chỉ đọc

    public void Move()
    {
        _speed = 60;
        Console.WriteLine($"Xe hơi {Model} đang di chuyển với tốc độ {_speed} km/h.");
    }

    public void Stop() {
         _speed = 0;
         Console.WriteLine($"Xe hơi {Model} đã dừng.");
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1**: Interface cơ bản, Cài đặt, Đa hình

```
// 3. Lớp thứ hai cài đặt Interface
    public class Person : IMovable
    {
        private int _speed = 0;
        public string Name { get; set; }

        public Person(string name) { Name = name; }

        // Cài đặt thành viên interface
        public int CurrentSpeed => _speed;

        public void Move()
        {
            _speed = 5;
            Console.WriteLine($"{Name} đang đi bộ với tốc độ {_speed} km/h.");
        }
        public void Run() {
             _speed = 15;
             Console.WriteLine($"{Name} đang chạy với tốc độ {_speed} km/h.");
        }
}
```

- **Ví dụ 1**: Interface cơ bản, Cài đặt, Đa hình

```
class Program
    {
        // Phương thức nhận tham số kiểu Interface
        static void MakeItMove(IMovable thing)
        {
             Console.Write("Yêu cầu di chuyển: ");
            thing.Move(); // Gọi phương thức Move() bất kể đối tượng là Car hay Person
             Console.WriteLine($"   -> Tốc độ hiện tại: {thing.CurrentSpeed}");
        }

        static void Main(string[] args)
        {
            Car myCar = new Car("Vinfast VF8");
            Person me = new Person("Giang");

            // Đa hình: Biến kiểu interface tham chiếu đến đối tượng lớp cài đặt
            IMovable mover1 = myCar;
            IMovable mover2 = me;

            Console.WriteLine("--- Sử dụng biến kiểu interface ---");
            mover1.Move(); // Gọi Car.Move()
            mover2.Move(); // Gọi Person.Move()

            Console.WriteLine("\n--- Sử dụng phương thức nhận interface ---");
            MakeItMove(myCar);
            MakeItMove(me);

            // Không thể gọi phương thức riêng của lớp qua tham chiếu interface
            // mover1.Stop(); // Lỗi! Stop() không có trong IMovable
            // mover2.Run(); // Lỗi! Run() không có trong IMovable
        }
    }
}
```

- **Ví dụ 2**: Cài đặt nhiều Interface, Kế thừa Interface

```
using System;

namespace MultipleInterfaceInheritance
{
    // Interface cơ bản
    public interface IDataSource
    {
        string GetData();
    }

    // Interface kế thừa từ IDataSource và thêm phương thức mới
public interface ITransactionalDataSource : IDataSource
{
    void BeginTransaction();
    void Commit();
    void Rollback();
}

// Một interface khác không liên quan
public interface ILoggable
{
     void LogMessage(string message);
}
```

- **Ví dụ 2**: Cài đặt nhiều Interface, Kế thừa Interface

```
// Lớp cài đặt interface kế thừa (ITransactionalDataSource)  // và một interface khác (ILoggable)
    public class DatabaseService : ITransactionalDataSource, ILoggable
    {
        private bool _inTransaction = false;

         public void LogMessage(string message) {
              Console.WriteLine($"DATABASE LOG: {message}");
         }

        // Cài đặt thành viên của IDataSource (kế thừa bởi ITransactionalDataSource)
        public string GetData()
        {
            LogMessage("Đang đọc dữ liệu từ database...");
            return "Dữ liệu từ Database";
        }

        // Cài đặt các thành viên của ITransactionalDataSource
        public void BeginTransaction()
        {
            _inTransaction = true;
             LogMessage("Bắt đầu Transaction.");
            Console.WriteLine("Database transaction started.");
        }
public void Commit()
        {
            if (_inTransaction)
            {
                 LogMessage("Commit Transaction.");
                Console.WriteLine("Database transaction committed.");
                _inTransaction = false;
            }
        }

        public void Rollback()
        {
            if (_inTransaction)
            {
                 LogMessage("Rollback Transaction.");
                Console.WriteLine("Database transaction rolled back.");
                _inTransaction = false;
            }
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 2**: Cài đặt nhiều Interface, Kế thừa Interface

```
class Program
    {
        static void ProcessData(ITransactionalDataSource transactionalDb) {
            Console.WriteLine("\n--- Bắt đầu xử lý dữ liệu ---");
             transactionalDb.BeginTransaction();
             try {
                  string data = transactionalDb.GetData(); // Gọi phương thức từ IDataSource (cha)
                  Console.WriteLine($"Đã nhận: {data}");
                  // ... xử lý dữ liệu ...
                  Console.WriteLine("Xử lý thành công!");
                  transactionalDb.Commit();
             }
             catch (Exception ex) {
                  Console.WriteLine($"Lỗi xử lý: {ex.Message}");
                   transactionalDb.Rollback();
             }
             Console.WriteLine("--- Kết thúc xử lý dữ liệu ---");
    }

    static void LogSomething(ILoggable logger, string info) {
         logger.LogMessage($"Thông tin cần log: {info}");
    }
```

- **Ví dụ 2**: Cài đặt nhiều Interface, Kế thừa Interface

```
static void Main(string[] args)
        {
            DatabaseService dbService = new DatabaseService();

            // Sử dụng qua interface kế thừa
            ProcessData(dbService);

             // Sử dụng qua interface khác mà lớp cài đặt
             LogSomething(dbService, "Hoàn tất tác vụ chính.");

             // Sử dụng qua interface cha
             IDataSource basicSource = dbService;
             Console.WriteLine($"\nĐọc dữ liệu cơ bản: {basicSource.GetData()}");
             // basicSource.Commit(); // Lỗi! Commit() không có trong IDataSource

        }
    }
}
```

- **Ví dụ 3**: Cài đặt Tường minh (Explicit Implementation)

```
using System;

namespace ExplicitImplementation
{
    interface IEnglishGreeting
    {
        void SayHello();
    }

    interface IFrenchGreeting
    {
        void SayHello(); // Cùng tên và signature với interface trên
    }
```

- **Ví dụ 3**: Cài đặt Tường minh (Explicit Implementation)

```
// Lớp cài đặt cả hai interface
    public class MultilingualPerson : IEnglishGreeting, IFrenchGreeting
    {
        // Cài đặt TƯỜNG MINH cho IEnglishGreeting
        // Không có 'public', chỉ truy cập được qua tham chiếu IEnglishGreeting
        void IEnglishGreeting.SayHello()
        {
            Console.WriteLine("Hello!");
        }

        // Cài đặt TƯỜNG MINH cho IFrenchGreeting
        void IFrenchGreeting.SayHello()
        {
            Console.WriteLine("Bonjour!");
        }

        // Có thể có một phương thức SayHello public riêng của lớp (tùy chọn)
        public void SayHello()
        {
            Console.WriteLine("Xin chào! (Vietnamese as default)");
        }
}
```

- **Ví dụ 3**: Cài đặt Tường minh (Explicit Implementation)

```
class Program
    {
        static void Main(string[] args)
        {
            MultilingualPerson person = new MultilingualPerson();

            // Gọi phương thức public mặc định của lớp
            Console.Write("Gọi trực tiếp từ đối tượng: ");
            person.SayHello(); // Output: Xin chào!

            // Để gọi các bản cài đặt tường minh, cần ép kiểu sang interface tương ứng
            Console.Write("Gọi qua tham chiếu IEnglishGreeting: ");
            IEnglishGreeting englishSpeaker = person;
            englishSpeaker.SayHello(); // Output: Hello!

            Console.Write("Gọi qua tham chiếu IFrenchGreeting: ");
            IFrenchGreeting frenchSpeaker = person;
            frenchSpeaker.SayHello(); // Output: Bonjour!
        }
    }
}
```

- **Ví dụ 4**: Kiểm tra kiểu, Ép kiểu, Extension Method với Interface

```
using System;
using System.Collections.Generic; // Dùng cho List<T>

namespace InterfaceCastingExtension
{
    // Định nghĩa các interface và class logger như Ví dụ 1
     public interface ILogger { void Log(string message); }
     public class ConsoleLogger : ILogger { public void Log(string m) => Console.WriteLine($"CONSOLE: {m}"); }
     public class FileLogger : ILogger { public void Log(string m) => Console.WriteLine($"FILE: {m}"); }
     // Thêm một lớp khác không liên quan
     public class Calculator {}

// Lớp static chứa extension method cho ILogger
public static class LoggerExtensions
{
    // Thêm phương thức LogError vào *bất kỳ* lớp nào cài đặt ILogger
    public static void LogError(this ILogger logger, string errorMessage, Exception ex = null)
    {
        string fullMessage = $"ERROR: {errorMessage}";
        if (ex != null)
        {
            fullMessage += $"\nException: {ex.Message}";
        }
        // Gọi phương thức Log() gốc của interface
        logger.Log(fullMessage);
    }
}
```

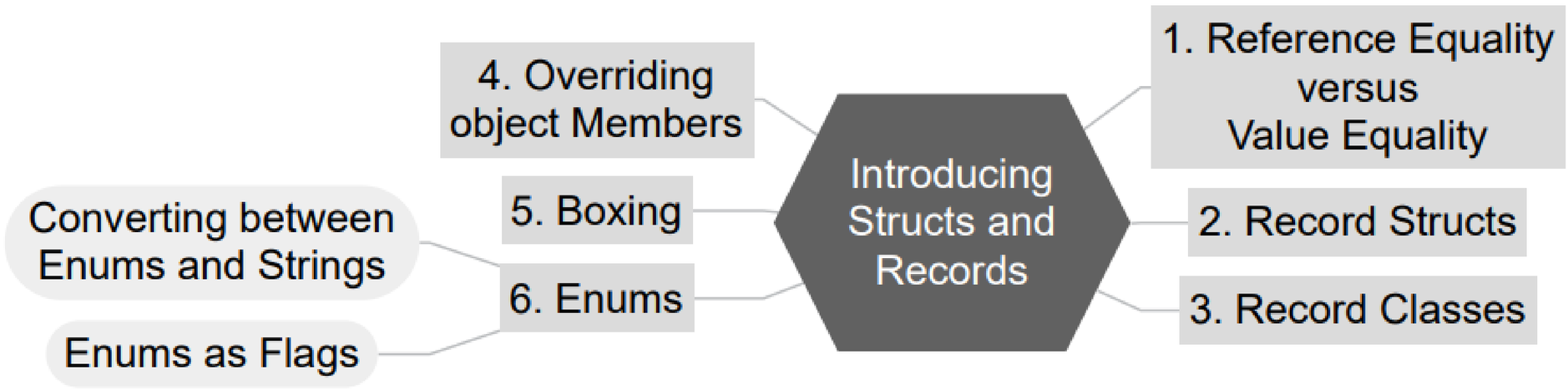
2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4**: Kiểm tra kiểu, Ép kiểu, Extension Method với Interface

```
class Program
    {
        static void Main(string[] args)
        {
            List<object> items = new List<object>();
            items.Add(new ConsoleLogger());
            items.Add(new FileLogger());
            items.Add(new Calculator());
            items.Add(new ConsoleLogger());
            items.Add("Một chuỗi bất kỳ");

            Console.WriteLine("Xử lý các item, chỉ log nếu là ILogger:");
            foreach (object item in items)
            {
                // 1. Dùng 'is' để kiểm tra
                if (item is ILogger)
                {
                    Console.Write("   - Item này là ILogger. ");
                    // Ép kiểu tường minh (an toàn sau khi check 'is')
                    ILogger loggerInstance = (ILogger)item;
                    loggerInstance.Log("Logging thông thường.");

                     // Gọi Extension Method
                    loggerInstance.LogError("Đã có lỗi xảy ra (ví dụ).");
                }
                // 2. Dùng 'as' để thử ép kiểu
                ILogger loggerAs = item as ILogger;
                if (loggerAs != null) // Kiểm tra null sau khi dùng 'as'
                {
                    Console.WriteLine("   - (Kiểm tra bằng 'as'): Có thể log.");
                    // loggerAs.LogError("Lỗi khác", new InvalidOperationException("Demo"));
                }
                else
                {
                    Console.WriteLine($"   - Item '{item.GetType().Name}' không phải là ILogger.");
                }
                 Console.WriteLine("   ---------");
            }
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 9**



2. LẬP TRÌNH C# CĂN BẢN

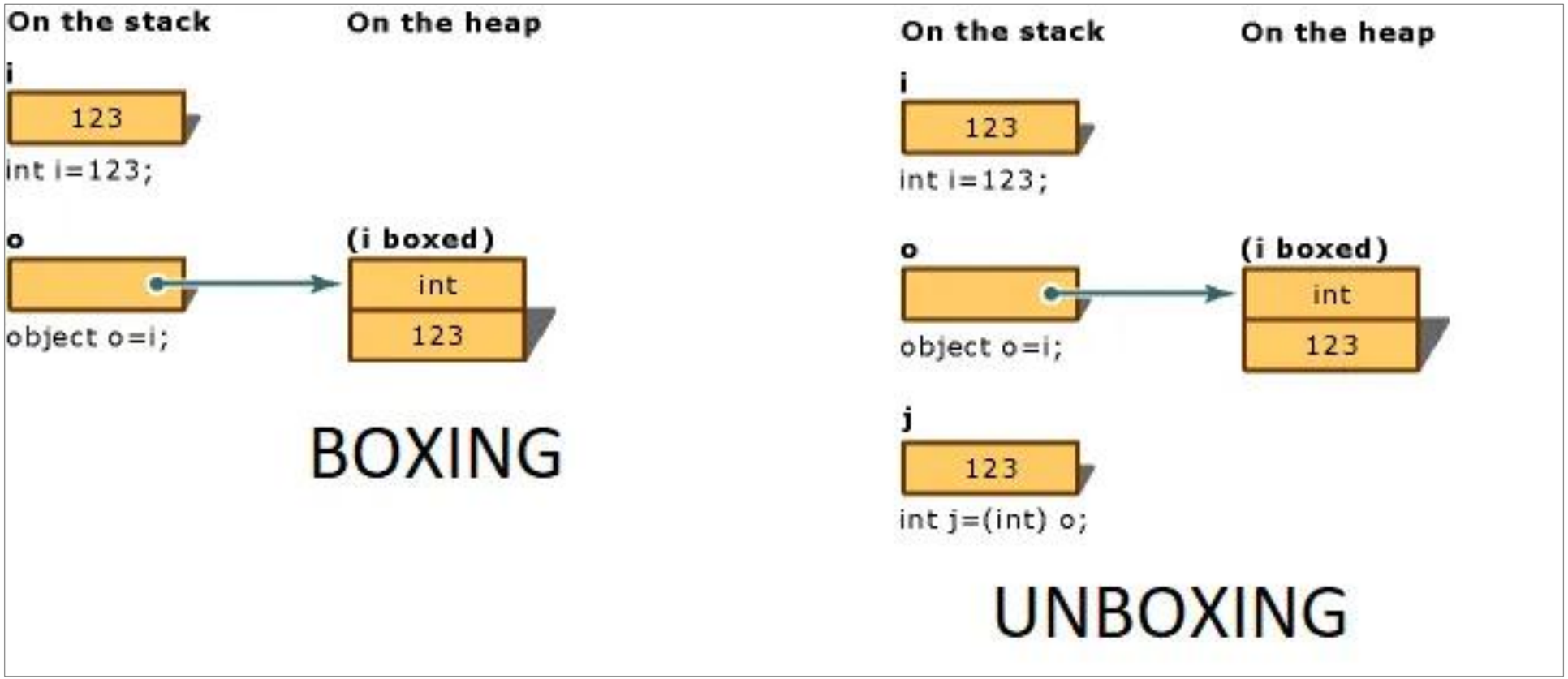
- **Ôn tập kiểu Giá trị (Value Type) vs. kiểu Tham chiếu (Reference Type)**
  - **Kiểu giá trị:**
    - Bao gồm các kiểu dữ liệu cơ bản (int, bool, double, char, ...), struct, enum.
    - Kế thừa ngầm từ **System.ValueType** (mà System.ValueType lại kế thừa từ **System.Object**).
    - Biến kiểu value type chứa trực tiếp giá trị của dữ liệu.
    - Thường được lưu trữ trên **stack** (vùng nhớ nhanh, quản lý tự động) nếu là biến cục bộ hoặc tham số phương thức.
    - Khi gán (**=**), giá trị được sao chép sang biến mới. Hai biến hoạt động độc lập.
    - Khi truyền vào phương thức (mặc định là **pass-by-value**), giá trị được sao chép vào tham số của phương thức. Thay đổi tham số không ảnh hưởng biến gốc.

- **Ôn tập kiểu Giá trị (Value Type) vs. kiểu Tham chiếu (Reference Type)**
  - **Kiểu tham chiếu:**
    - Bao gồm class, interface, delegate, string, array.
    - Kế thừa từ **System.Object**.
    - Biến kiểu reference type chứa địa chỉ (tham chiếu) đến vùng nhớ trên **heap** (vùng nhớ lớn hơn, quản lý bởi Garbage Collector) nơi dữ liệu thực tế được lưu trữ.
    - Khi gán (=), chỉ địa chỉ tham chiếu được sao chép. Cả hai biến cùng trỏ đến một đối tượng duy nhất trên heap.
    - Khi truyền vào phương thức (pass-by-value), địa chỉ tham chiếu được sao chép vào tham số. Cả biến gốc và tham số cùng trỏ đến một đối tượng. Thay đổi trạng thái của đối tượng thông qua tham số sẽ ảnh hưởng đến đối tượng mà biến gốc đang trỏ tới.

2. LẬP TRÌNH C# CĂN BẢN

- **Struct**
  - Là một kiểu dữ liệu giá trị do người dùng định nghĩa, tương tự như class nhưng có những khác biệt quan trọng.
  - Cú pháp: **struct TenStruct { /* thành viên */ }**.
  - **Đặc điểm chính**:
    - Là value type, hoạt động theo ngữ nghĩa sao chép giá trị.
    - Kế thừa từ **System.ValueType**, không thể kế thừa từ class hoặc struct khác.
    - Có thể cài đặt (implement) một hoặc nhiều interface.
    - Constructor: (Từ C# 10 trở đi): Được phép khai báo constructor không tham số tường minh.
    - **Sử dụng**: Thường dùng cho các kiểu dữ liệu nhỏ, đơn giản, tập trung vào việc lưu trữ dữ liệu, và khi bạn muốn ngữ nghĩa sao chép giá trị (ví dụ: tọa độ điểm Point, số phức Complex, màu sắc Color, cặp khóa-giá trị đơn giản).

- **Boxing và Unboxing**
  - Là **quá trình chuyển đổi** qua lại giữa một kiểu giá trị (**value type**) và kiểu tham chiếu **object** (hoặc một kiểu interface mà value type đó cài đặt).



2. LẬP TRÌNH C# CĂN BẢN

- **Boxing và Unboxing**
  - **Boxing** (Đóng hộp):
    - Chuyển đổi từ value type sang object.
    - Xảy ra khi gán một biến value type cho một biến kiểu object, hoặc truyền value type vào phương thức yêu cầu tham số object, hoặc thêm value type vào collection không generic (như ArrayList).
  - **Quá trình**:
    - Cấp phát một vùng nhớ trên heap.
    - Sao chép giá trị của value type vào vùng nhớ heap vừa cấp phát.
    - Trả về một tham chiếu (kiểu object) trỏ đến đối tượng trên heap đó.
    - Có thể xảy ra ngầm định (**implicit**).

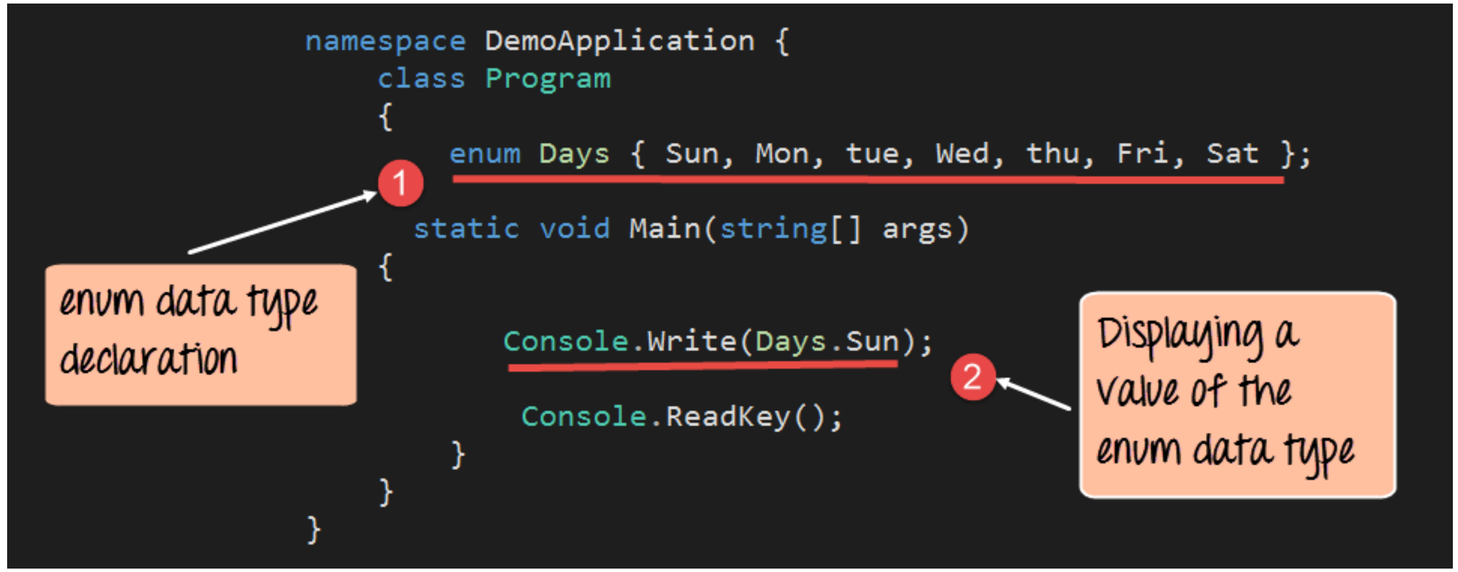
- **Boxing và Unboxing**
  - **UnBoxing** (Mở hộp):
    - Chuyển đổi từ object (mà thực chất đang giữ một giá trị đã được boxing) trở lại kiểu value type gốc.
    - Bắt buộc phải thực hiện ép kiểu tường minh `(KiểuValueTypeGốc) bienObject;`..
    - **Quá trình**:
      - Kiểm tra xem đối tượng trên heap có thực sự chứa giá trị thuộc `KiểuValueTypeGốc` (hoặc kiểu nullable của nó) hay không.
      - Nếu đúng, sao chép giá trị từ heap vào biến value type (thường trên stack).
      - Nếu kiểu không khớp, ném ra **InvalidCastException**..

- **Boxing và Unboxing**
  - **Ảnh hưởng hiệu năng:** Boxing và Unboxing là các thao tác tốn kém vì liên quan đến việc cấp phát bộ nhớ trên heap và sao chép dữ liệu. Nên tránh chúng trong các vòng lặp chặt hoặc các đoạn code yêu cầu hiệu năng cao. Generics (Chương 11) là một giải pháp hiệu quả để tránh boxing/unboxing khi làm việc với collections.

```
1   const int noNumbers = 10000000;  // 10 mil
2
3   ArrayList numbers = new ArrayList();
4   Random random = new Random(1);   // use the same seed as to make
5                                    //  benchmarking consistent
6
7   for (int i = 0; i < noNumbers; i++)
8   {
9       int currentNumber = random.Next(10);  // generate a non-negative
10                                            //  random number less than 10
11      object o = currentNumber;  // BOXING occurs here
12
13      numbers.Add(o);
14  }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Enum (Kiểu liệt kê)**
  - Là một kiểu dữ liệu giá trị đặc biệt, cho phép định nghĩa một tập hợp các hằng số có tên (gọi là enumerators).
  - Mục đích: Giúp mã nguồn dễ đọc, dễ hiểu và dễ bảo trì hơn so với việc sử dụng các giá trị số nguyên "ma thuật" (magic numbers).



2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1:** So sánh Struct vs Class (Value vs Reference Semantics)

```
using System;

namespace StructVsClass
{
    // Struct (Value Type)
    struct PointStruct
    {
        public int X;
        public int Y;
        public PointStruct(int x, int y) { X = x; Y = y; }
        public void Display() => Console.WriteLine($"Struct Point: ({X}, {Y})");
    }


// Class (Reference Type)
    class PointClass
    {
        public int X;
        public int Y;
        public PointClass(int x, int y) { X = x; Y = y; }
        public void Display() => Console.WriteLine($"Class Point: ({X}, {Y})");
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1:** So sánh Struct vs Class (Value vs Reference Semantics)

```
class Program
    {
        static void ModifyPointStruct(PointStruct ps) { // Nhận bản sao
            ps.X = 100;
            Console.Write(" -> Bên trong ModifyPointStruct: "); ps.Display();
        }

         static void ModifyPointClass(PointClass pc) { // Nhận bản sao của tham chiếu
             pc.X = 100; // Thay đổi đối tượng gốc trên heap
             Console.Write(" -> Bên trong ModifyPointClass: "); pc.Display();
         }

        static void Main(string[] args)
        {
            Console.WriteLine("--- Phép gán ---");
            PointStruct ps1 = new PointStruct(10, 20);
            PointStruct ps2 = ps1; // Sao chép giá trị
            ps2.X = 50;
            Console.Write("ps1 sau khi thay đổi ps2: "); ps1.Display(); // ps1 không đổi
            Console.Write("ps2 sau khi thay đổi ps2: "); ps2.Display();

            PointClass pc1 = new PointClass(10, 20);
            PointClass pc2 = pc1; // Sao chép tham chiếu (cùng trỏ đến 1 đối tượng)
            pc2.X = 50;
            Console.Write("pc1 sau khi thay đổi pc2: "); pc1.Display(); // pc1 THAY ĐỔI
            Console.Write("pc2 sau khi thay đổi pc2: "); pc2.Display();
        }
    }
}
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 1:** So sánh Struct vs Class (Value vs Reference Semantics)

```
class Program
    {
        static void ModifyPointStruct(PointStruct ps) { // Nhận bản sao
            ps.X = 100;
            Console.Write(" -> Bên trong ModifyPointStruct: "); ps.Display();
        }

         static void ModifyPointClass(PointClass pc) { // Nhận bản sao của tham chiếu
             pc.X = 100; // Thay đổi đối tượng gốc trên heap
             Console.Write(" -> Bên trong ModifyPointClass: "); pc.Display();
         }

        static void Main(string[] args)
        {
            Console.WriteLine("\n--- Truyền tham số vào phương thức (Pass by Value) ---");
            PointStruct ps_orig = new PointStruct(1, 1);
            Console.Write("Struct gốc trước khi gọi Modify: "); ps_orig.Display();
            ModifyPointStruct(ps_orig);
            Console.Write("Struct gốc sau khi gọi Modify: "); ps_orig.Display(); // Không đổi

            PointClass pc_orig = new PointClass(1, 1);
            Console.Write("Class gốc trước khi gọi Modify: "); pc_orig.Display();
            ModifyPointClass(pc_orig);
            Console.Write("Class gốc sau khi gọi Modify: "); pc_orig.Display(); // Bị thay đổi!
        }
    }
}
```

- **Ví dụ 2:** Khai báo và sử dụng Struct

```
struct Rectangle
    {
        // Fields (có thể dùng auto-property thay thế)
        public double Width;
        public double Height;

        // Constructor có tham số (phải gán tất cả fields)
        public Rectangle(double width, double height)
        {
            this.Width = width;
            this.Height = height;
        }

        // // Constructor không tham số (chỉ từ C# 10+)
        // public Rectangle() {
        //      this.Width = 1.0;
        //      this.Height = 1.0;
        // }

        // Method
        public double CalculateArea()
        {
            return Width * Height;
        }

        public void Display()
        {
            Console.WriteLine($"Rectangle [Width={Width}, Height={Height}, Area={CalculateArea()}]");
        }
}
```

- **Ví dụ 2:** Khai báo và sử dụng Struct

```
class Program
    {
        static void Main(string[] args)
        {
            // Tạo struct dùng constructor có tham số
            Rectangle rect1 = new Rectangle(5.0, 3.0);
            rect1.Display();

            // Tạo struct dùng constructor không tham số mặc định (fields là 0)
            Rectangle rect2 = new Rectangle(); // Width=0, Height=0
            rect2.Display();
            // Gán giá trị cho fields sau
            rect2.Width = 7.0;
            rect2.Height = 2.5;
            rect2.Display();

             // Sao chép giá trị
            Rectangle rect3 = rect1;
            rect3.Width = 100; // Thay đổi rect3 không ảnh hưởng rect1
            Console.Write("Rect1 sau khi thay đổi rect3: "); rect1.Display();
            Console.Write("Rect3 sau khi thay đổi rect3: "); rect3.Display();
        }
}
```

- **Ví dụ 3:** Boxing và Unboxing

```
struct SimpleStruct { public int Value; }
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("--- Boxing ---");
            int i = 123;
            double d = 45.6;
            SimpleStruct ss = new SimpleStruct { Value = 789 };

            // Boxing: Value type được chuyển thành object (tham chiếu đến heap)
            object obj_i = i;
            object obj_d = d;
            object obj_ss = ss;

            Console.WriteLine($"i (int): {i}, obj_i (object): {obj_i}, Type of obj_i: {obj_i.GetType()}");
            Console.WriteLine($"d (double): {d}, obj_d (object): {obj_d}, Type of obj_d: {obj_d.GetType()}");
            Console.WriteLine($"ss (SimpleStruct): {ss.Value}, obj_ss (object): {obj_ss}, Type of obj_ss: {obj_ss.GetType()}");

    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 3:** Boxing và Unboxing

```
struct SimpleStruct { public int Value; }
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n--- Unboxing ---");
            try
            {
                // Unboxing: Ép kiểu tường minh từ object về value type gốc
                int j = (int)obj_i;
                double e = (double)obj_d;
                SimpleStruct ss2 = (SimpleStruct)obj_ss;

                Console.WriteLine($"j (unboxed int): {j}");
                Console.WriteLine($"e (unboxed double): {e}");
                 Console.WriteLine($"ss2 (unboxed SimpleStruct): {ss2.Value}");

                // Thử unbox sai kiểu -> InvalidCastException
                Console.WriteLine("\nThử unbox sai kiểu:");
                 // int k = (int)obj_d; // Lỗi! obj_d chứa double, không phải int
                 long l = (long)obj_i; // Lỗi! obj_i chứa int, không phải long
            }
            catch (InvalidCastException ex)
            {
                Console.WriteLine($"(!) Lỗi Unboxing: {ex.Message}");
            }
        }
}
```

- **Ví dụ 3:** Boxing và Unboxing

```
struct SimpleStruct { public int Value; }
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n--- Boxing trong Collection không Generic ---");
            ArrayList list = new ArrayList();
            list.Add(10);       // Boxing int -> object
            list.Add(3.14);   // Boxing double -> object
            list.Add(new SimpleStruct { Value = 5 }); // Boxing struct -> object
            Console.WriteLine($"ArrayList chứa {list.Count} đối tượng (đã bị boxing).");

            // Lấy ra phải unbox
            int firstItem = (int)list[0]; // Unboxing
            Console.WriteLine($"Phần tử đầu tiên (sau unboxing): {firstItem}");

            // Giải pháp tốt hơn: Dùng List<T> (Generics - Chương 11) để tránh boxing/unboxing
            // List<int> intList = new List<int>();
            // intList.Add(10); // Không boxing!
        }
}
```

- **Ví dụ 4:** Enum cơ bản, sử dụng trong switch

```
public enum OrderStatus
    {
        Pending,    // Giá trị 0 (mặc định)
        Processing, // Giá trị 1
        Shipped,    // Giá trị 2
        Delivered,  // Giá trị 3
        Cancelled   // Giá trị 4
    }
```

```
class Program
    {
        static void DisplayOrderStatusMessage(OrderStatus status)
        {
            switch (status)
            {
                case OrderStatus.Pending:
                    Console.WriteLine("Đơn hàng đang chờ xử lý.");
                    break;
                case OrderStatus.Processing:
                    Console.WriteLine("Đơn hàng đang được chuẩn bị.");
                    break;
                case OrderStatus.Shipped:
                    Console.WriteLine("Đơn hàng đã được giao cho đơn vị vận chuyển.");
                    break;
                case OrderStatus.Delivered:
                    Console.WriteLine("Đơn hàng đã giao thành công.");
                    break;
                case OrderStatus.Cancelled:
                    Console.WriteLine("Đơn hàng đã bị hủy.");
                    break;
                default: // Trường hợp dự phòng (ít khi xảy ra với enum)
                    Console.WriteLine("Trạng thái đơn hàng không xác định.");
                    break;
            }
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 4:** Enum cơ bản, sử dụng trong switch

```
public enum OrderStatus
    {
        Pending,    // Giá trị 0 (mặc định)
        Processing, // Giá trị 1
        Shipped,    // Giá trị 2
        Delivered,  // Giá trị 3
        Cancelled   // Giá trị 4
    }
```

```
static void Main(string[] args)
        {
            // Khai báo và gán giá trị enum
            OrderStatus currentStatus = OrderStatus.Processing;

            Console.WriteLine($"Trạng thái hiện tại: {currentStatus}"); // In ra tên hằng
số: Processing
            Console.WriteLine($"Giá trị số của trạng thái: {(int)currentStatus}"); // Ép
kiểu sang int: 1

            // Sử dụng trong phương thức hoặc switch
            DisplayOrderStatusMessage(currentStatus);

            // Thay đổi trạng thái
            currentStatus = OrderStatus.Shipped;
            DisplayOrderStatusMessage(currentStatus);

            // So sánh
            if (currentStatus == OrderStatus.Shipped)
            {
                Console.WriteLine("=> Đã ship!");
            }
    }
```

- **Ví dụ 5:** Enum với kiểu nền, giá trị tường minh, System.Enum

```
public enum ErrorLevel : byte // Kiểu nền là byte
    {
        None = 0,
        Information = 1,
        Warning = 10,  // Giá trị không cần liên tiếp
        Error = 20,
        Critical = 100
    }

class Program
    {
        static void Main(string[] args)
        {
            ErrorLevel level = ErrorLevel.Warning;
            Console.WriteLine($"Mức lỗi: {level}"); // Warning
            Console.WriteLine($"Giá trị byte: {(byte)level}"); // 10

            // Lấy tên từ giá trị
            string warningName = Enum.GetName(typeof(ErrorLevel), 10);
            Console.WriteLine($"Tên của giá trị 10: {warningName}"); // Warning

            // Lấy tất cả tên
            Console.WriteLine("\nTất cả các mức lỗi (tên):");
            string[] names = Enum.GetNames(typeof(ErrorLevel));
            foreach (string name in names)
            {
                Console.WriteLine($"- {name}");
            }
        }
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 5:** Enum với kiểu nền, giá trị tường minh, System.Enum

```
public enum ErrorLevel : byte // Kiểu nền là byte
    {
        None = 0,
        Information = 1,
        Warning = 10,  // Giá trị không cần liên tiếp
        Error = 20,
        Critical = 100
    }
class Program
    {
        static void Main(string[] args)
    {
        // Lấy tất cả giá trị
        Console.WriteLine("\nTất cả các mức lỗi (giá trị byte):");
        Array values = Enum.GetValues(typeof(ErrorLevel)); // Trả về Array
        foreach (var val in values) // Kiểu thực tế là ErrorLevel, nhưng có thể duyệt qua
        {
             Console.WriteLine($"- {(byte)val}"); // Ép kiểu sang byte để xem giá trị nền
        }

         // Chuyển đổi từ chuỗi (Parse/TryParse) - phân biệt chữ hoa/thường mặc định
        string errorStr = "Error";
         // ErrorLevel parsedLevel = (ErrorLevel)Enum.Parse(typeof(ErrorLevel), errorStr); // Ném lỗi nếu chuỗi sai
        if (Enum.TryParse<ErrorLevel>(errorStr, out ErrorLevel tryParsedLevel)) // Dùng TryParse an toàn hơn
        {
             Console.WriteLine($"\nParse chuỗi \"{errorStr}\" thành công: {tryParsedLevel}");
        } else {
              Console.WriteLine($"\nKhông thể parse chuỗi \"{errorStr}\"");
        }
    }
}
```

- **Ví dụ 5:** Enum với kiểu nền, giá trị tường minh, System.Enum

```
public enum ErrorLevel : byte // Kiểu nền là byte
    {
        None = 0,
        Information = 1,
        Warning = 10,  // Giá trị không cần liên tiếp
        Error = 20,
        Critical = 100
    }


class Program
{
    static void Main(string[] args)
    {
        string invalidStr = "Fatal";
        if (!Enum.TryParse<ErrorLevel>(invalidStr, true, out ErrorLevel tryParsedLevelIgnoreCase))
          // true -> không phân biệt hoa/thường
        {
              Console.WriteLine($"Không thể parse chuỗi \"{invalidStr}\"");
         }
    }
}
```

- **Ví dụ 6:** Enum với [Flags]

```
// Enum với kiểu nền là byte và giá trị tường minh
    public enum ErrorLevel : byte // Kiểu nền là byte
    {
        None = 0,
        Information = 1,
        Warning = 10,  // Giá trị không cần liên tiếp
        Error = 20,
        Critical = 100
    }
```

```
// Thuộc tính [Flags] cho biết enum này dùng cho cờ bit
    [Flags]
    public enum FileAccessPermissions
    {
        None = 0,       // 0000
        Read = 1,       // 0001
        Write = 2,      // 0010
        Execute = 4,    // 0100
        ReadWrite = Read | Write // Kết hợp cờ: 0011 (giá trị 3)
        // Thường không cần định nghĩa các giá trị kết hợp sẵn
    }
```

2. LẬP TRÌNH C# CĂN BẢN

- **Ví dụ 6:** Enum với [Flags]

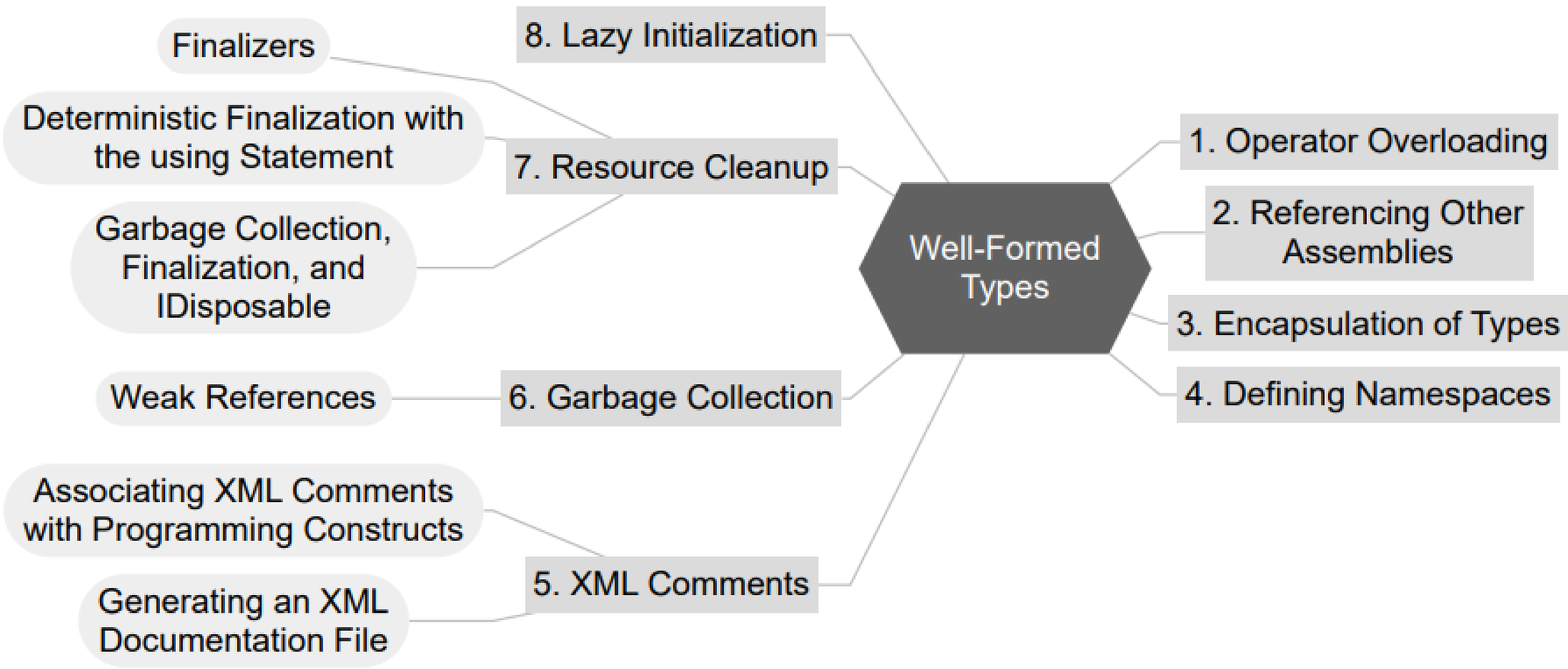
```
static void CheckPermissions(FileAccessPermissions currentPermissions)
        {
            Console.WriteLine($"\nQuyền hiện tại: {currentPermissions}"); // ToString() hiển thị các cờ kết hợp

            // Kiểm tra cờ bằng toán tử &
            if ((currentPermissions & FileAccessPermissions.Read) == FileAccessPermissions.Read)
            {
                Console.WriteLine("- Có quyền Đọc (dùng &)");
            }

             // Kiểm tra cờ bằng HasFlag() (rõ ràng hơn)
            if (currentPermissions.HasFlag(FileAccessPermissions.Write))
            {
                Console.WriteLine("- Có quyền Ghi (dùng HasFlag)");
            }

            if (currentPermissions.HasFlag(FileAccessPermissions.Execute))
            {
                Console.WriteLine("- Có quyền Thực thi");
            }

            if (currentPermissions == FileAccessPermissions.None) {
                 Console.WriteLine("- Không có quyền nào.");
            }
        }
```

- **Ví dụ 6:** Enum với [Flags]

```
static void Main(string[] args)
        {
            // Gán một cờ
            FileAccessPermissions user1Permissions = FileAccessPermissions.Read;
            CheckPermissions(user1Permissions);

            // Kết hợp nhiều cờ bằng toán tử | (OR)
            FileAccessPermissions user2Permissions = FileAccessPermissions.Read | FileAccessPermissions.Write;
            CheckPermissions(user2Permissions); // Output: Read, Write (hoặc ReadWrite nếu có định nghĩa)

            FileAccessPermissions adminPermissions = FileAccessPermissions.Read | FileAccessPermissions.Write |
FileAccessPermissions.Execute;
            CheckPermissions(adminPermissions); // Output: Read, Write, Execute

            // Gán giá trị kết hợp đã định nghĩa (nếu có)
            FileAccessPermissions user3Permissions = FileAccessPermissions.ReadWrite;
             CheckPermissions(user3Permissions);

            // Xóa một cờ bằng toán tử & và ~ (NOT)
            Console.WriteLine("\nXóa quyền Write khỏi admin:");
            adminPermissions = adminPermissions & ~FileAccessPermissions.Write; // Giữ lại các bit khác, xóa bit Write
             CheckPermissions(adminPermissions); // Output: Read, Execute
        }
```

2. LẬP TRÌNH C# CĂN BẢN

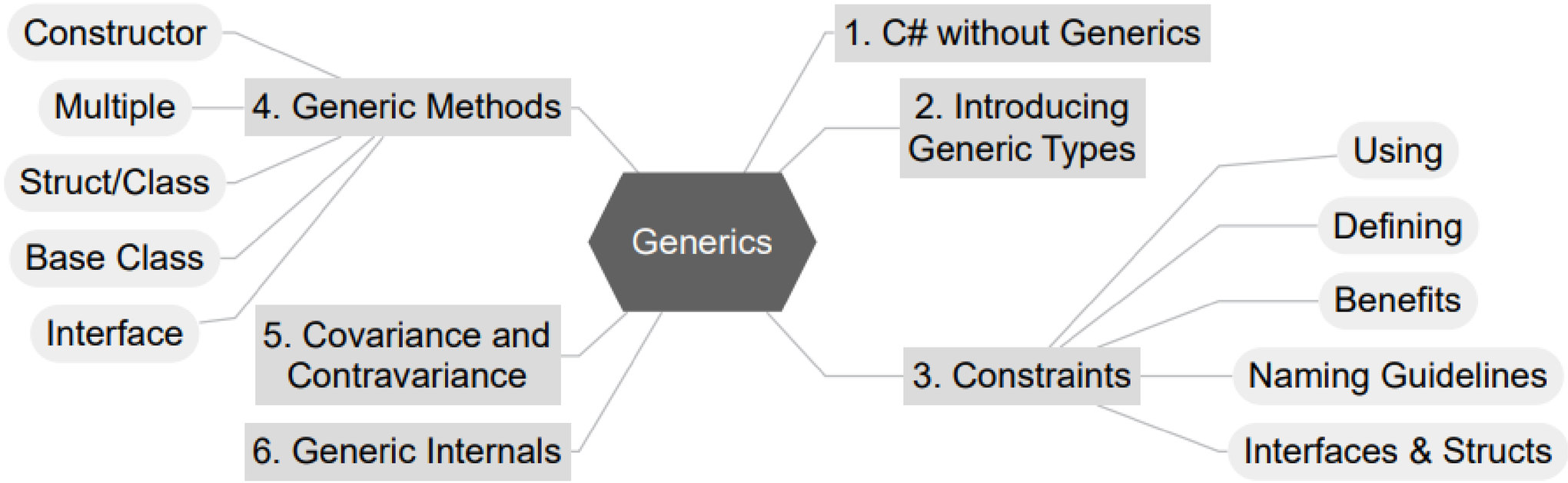
- **Nội dung Chương 10**



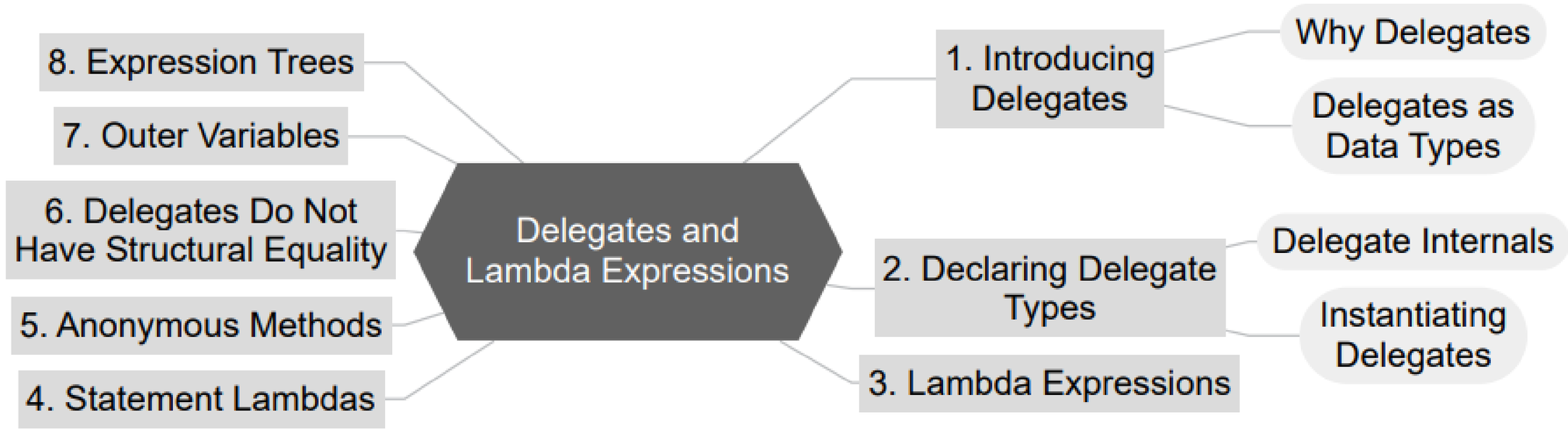
- **Nội dung Chương 11**

2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 12**

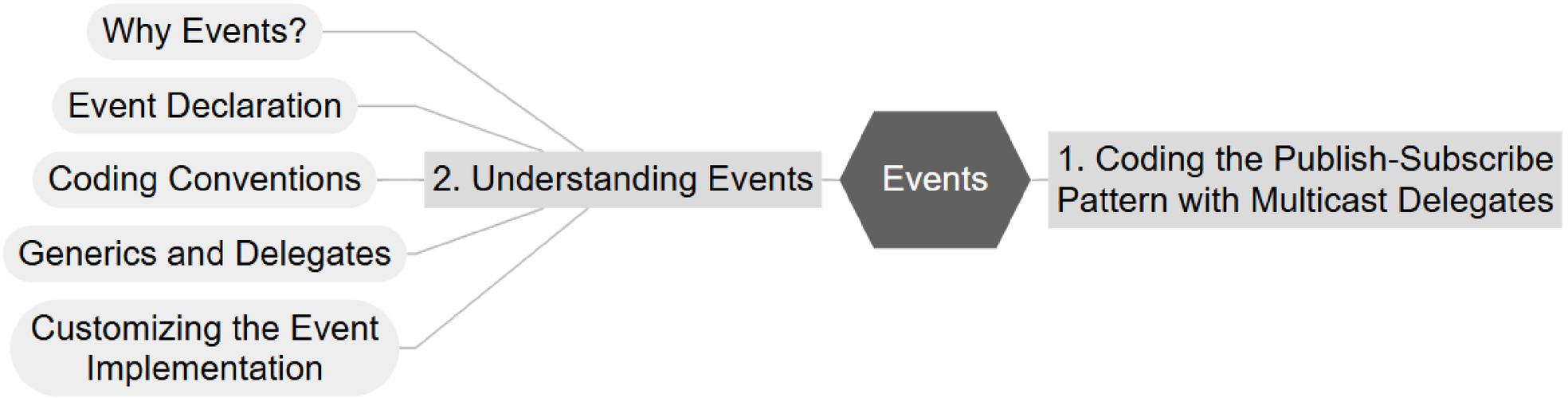


- **Nội dung Chương 13**

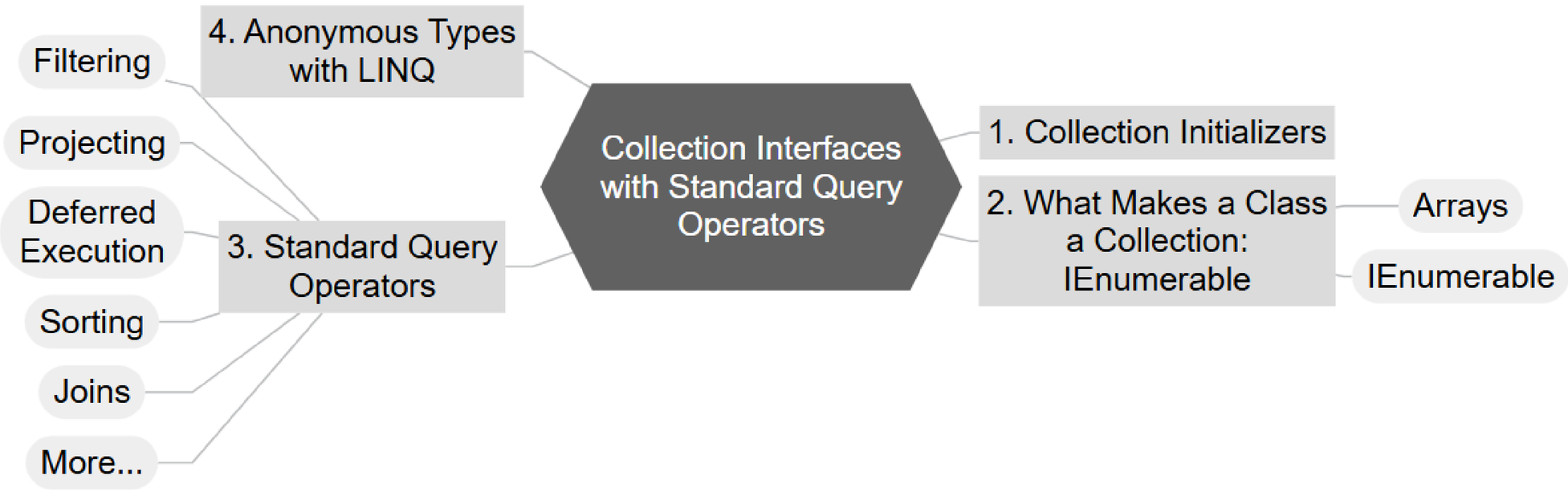


2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 14**



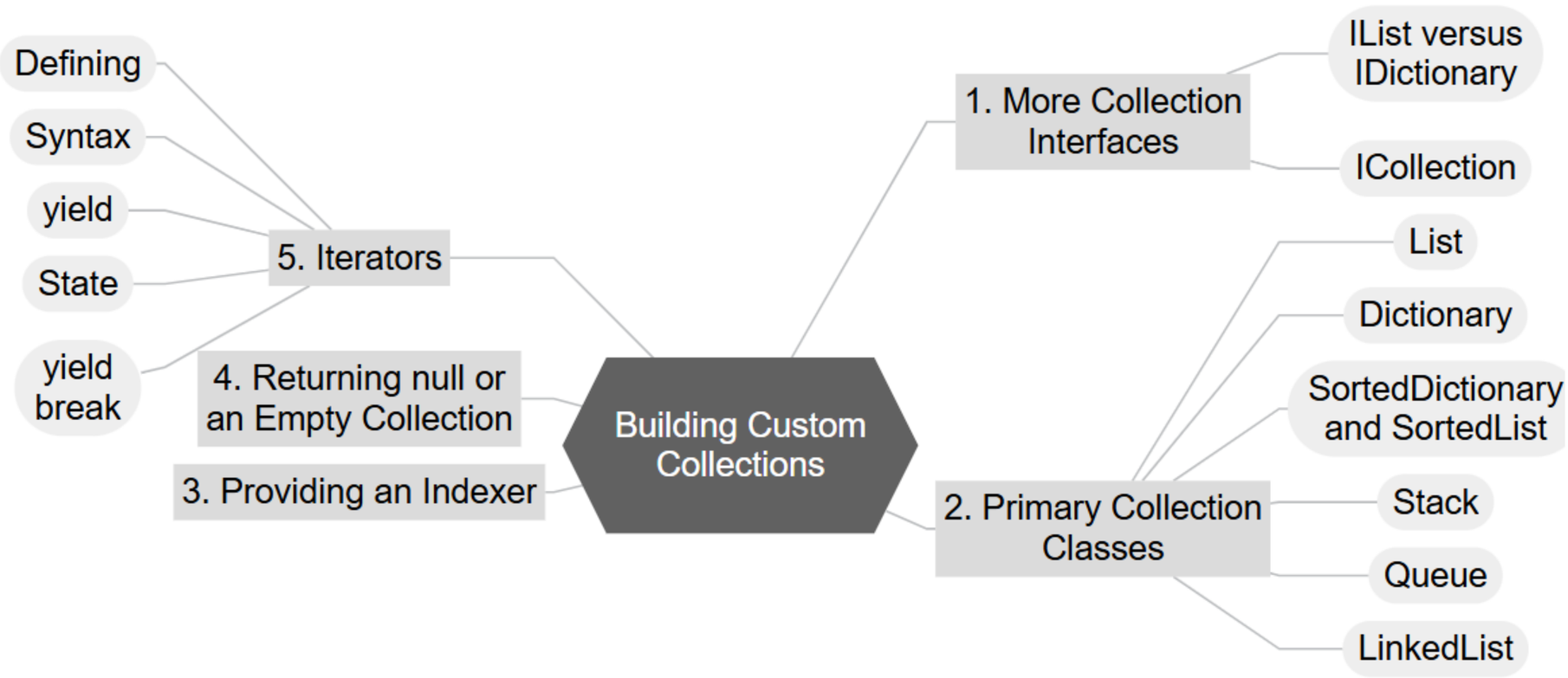
- **Nội dung Chương 15**



- **Nội dung Chương 16**

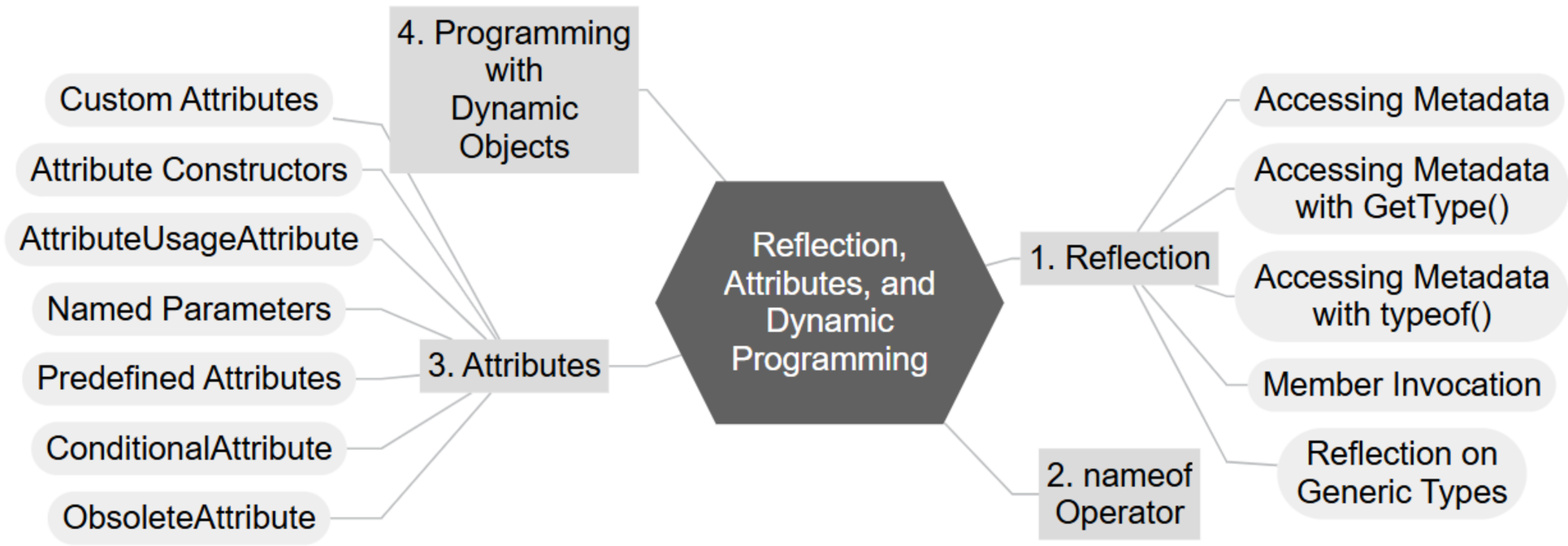
2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 17**



2. LẬP TRÌNH C# CĂN BẢN

- **Nội dung Chương 18**



ĐẠI NAM
UNIVERSITY

HỎI & ĐÁP