TEDU-46: C# căn bản đến nâng cao

Contents

Bài 1: Giới thiệu về ngôn ngữ C# và .NET Platform

Giới thiệu khoá học

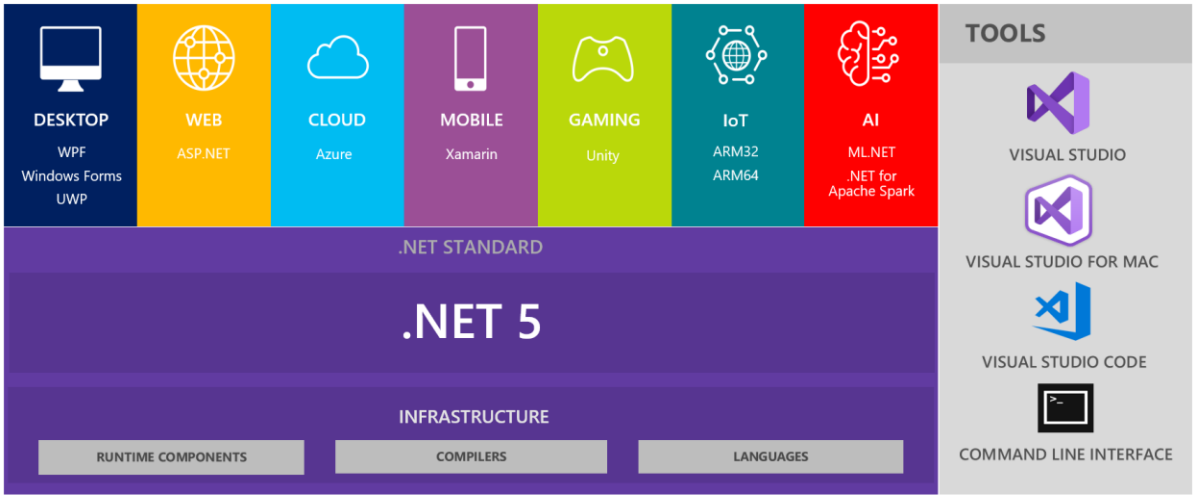
- Đây là khoá học hướng dẫn cho các bạn chưa biết gì về C# muốn theo lập trình C# và .NET.
- Trong quá trình học các bạn nhớ like và share video nếu thấy hữu ích. Các bạn cũng có thể comment để hỏi bất cứ vấn đề gì chưa rõ.
- Khoá học này làm nền tảng cho các khoá học khác tại TEDU.COM.VN.
- Sau khoá học này mình sẽ ra mắt khoá học nâng cao hơn về C# ví dụ như WPF.
- Các bạn có thể join vào link Group facebook hoặc Discord của TEDU ở link mô tả Video.
- Các bạn có thể download tài liệu này tại lớp học khoá học TEDU-46 (Lập trình C# toàn tập cho người mới bắt đầu) trên trang web chính thức TEDU.COM.VN

Giới thiệu .NET

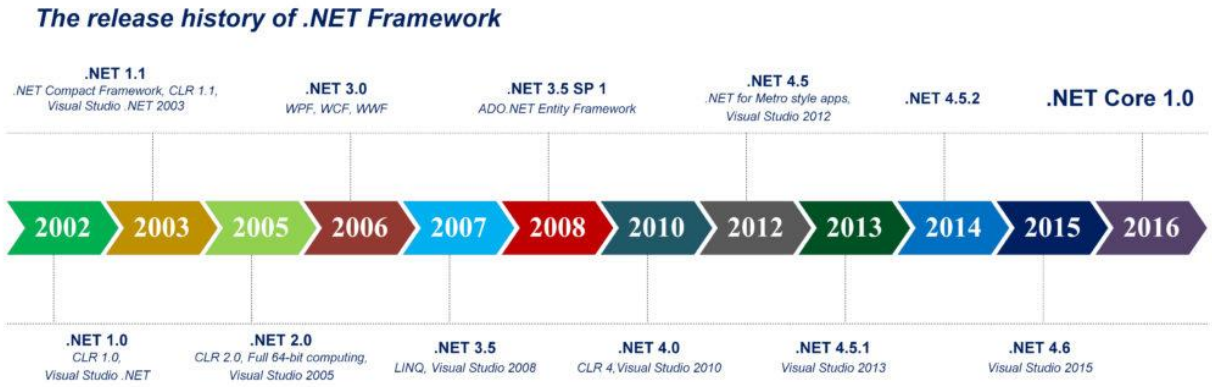
- Giới thiệu về .NET Platform là một nền tảng thống nhất phát triển nhiều loại ứng dụng từ Mobile, Desktop cho đến Web được phát triển bởi Microsoft.

- .NET Framework được Microsoft đưa ra chính thức từ năm 2002. .NET Framework chỉ hoạt động trên Windows. Những nền tảng ứng dụng như WPF, Winforms, ASP.NET (1-4) hoạt động dựa trên .NET Framework.
- Mono là phiên bản cộng đồng nhằm mang .NET đến những nền tảng ngoài Windows. Mono được phát triển chủ yếu nhằm xây dựng những ứng dụng với giao diện người dùng và được sử dụng rất rộng rãi: Unity Game, Xamarin…
- Cho đến năm 2013, Microsoft định hướng đi đa nền tảng và phát triển .NET core. .NET core hiện được sử dụng trong các ứng dụng Universal Windows platform và ASP.NET Core. Từ đây, C# có thể được sử dụng để phát triển các loại ứng dụng đa nền tảng trên các hệ điều hành khác nhau (Windows, Linux, MacOS,)

.NET – A unified platform

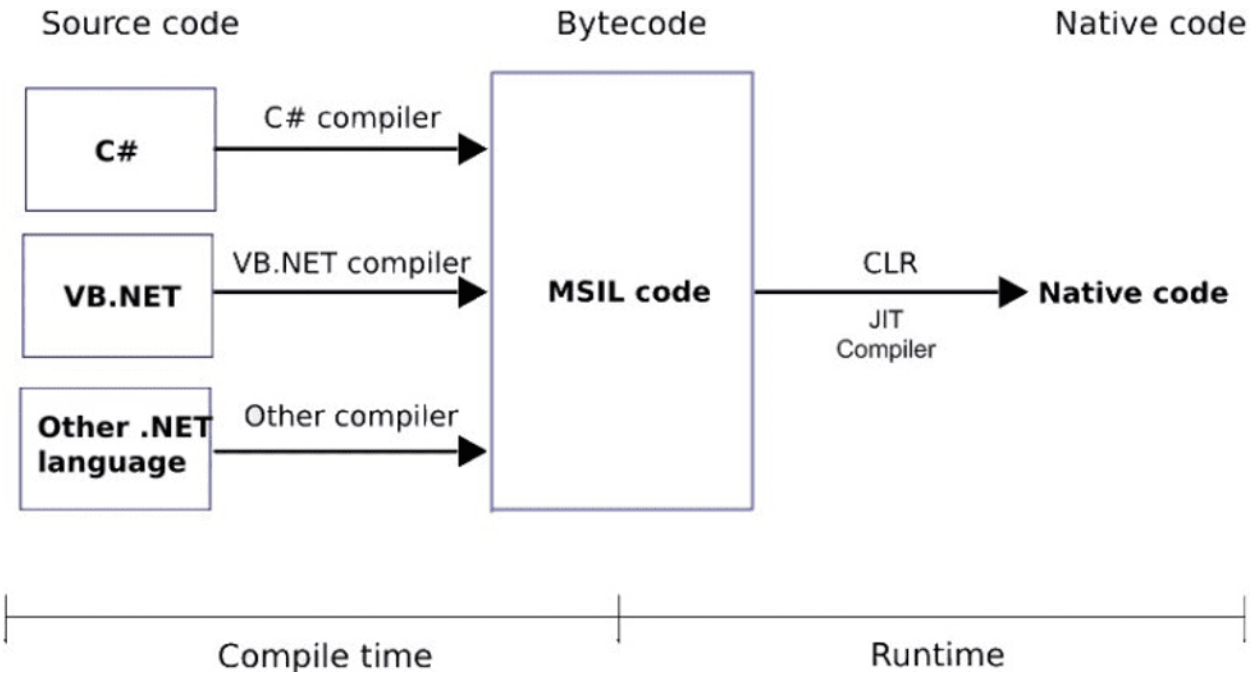


Lịch sử phát triển



Ngôn ngữ C#

- C# (hay C sharp) là một ngôn ngữ lập trình đơn giản, được phát triển bởi đội ngũ kỹ sư của Microsoft vào năm 2000.
- C# là ngôn ngữ lập trình hiện đại, hướng đối tượng và được xây dựng trên nền tảng của hai ngôn ngữ mạnh nhất là C++ và Java.



- Lịch sử phiên bản C#: [The history of C# - C# Guide | Microsoft Docs]

Bài 2: Cài đặt Visual Studio Community 2022 và giới thiệu giao diện

Cài đặt môi trường

1. Download Visual Studio 2022 tại: https://visualstudio.microsoft.com/ sau đó cài đặt trực tiếp trên máy.
2. Download .NET 6 SDK tại: https://dotnet.microsoft.com/en-us/download/dotnet/6.0
3. Test thử xem đã cài đặt thành công hay chưa? Bằng cách mở cửa sổ CMD gõ **dotnet –list-sdks**

Giới thiệu giao diện Visual Studio

- Màn hình khởi động
- Các cửa số cần thiết
- Tuỳ chỉnh giao diện

Bài 3: Viết chương trình C# đầu tiên

1. Giới thiệu cấu trúc chương trình HelloWorld
2. Giới thiệu về file Program.cs
3. Giới thiệu về namespace, class và method
4. Câu lệnh Console.Writeline()

Tham khảo: https://docs.microsoft.com/en-us/dotnet/core/tutorials/top-level-templates

Bài 4: Tổng quan về khái niệm kiểu dữ liệu và biến

Khái niệm về biến

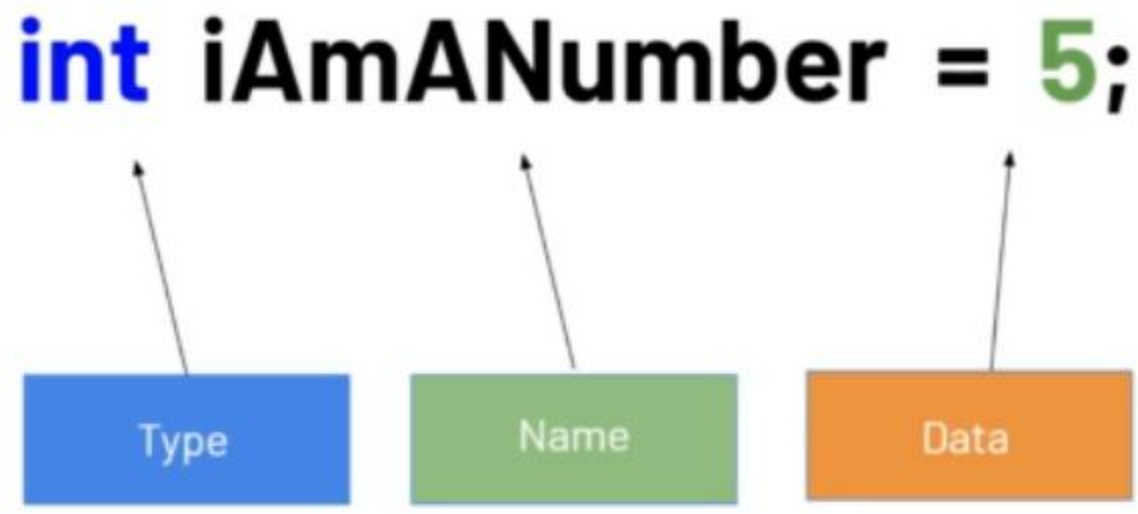
- Biến là một khái niệm cơ bản và quan trọng nhất của tất cả các ngôn ngữ lập trình.
- Biến là một không gian chứa một giá trị dữ liệu, và có thể được gán đi gán lại các giá trị khác nhau trên cùng 1 biến.
- Biến cấu tạo bởi kiểu dữ liệu, tên biến và dữ liệu lưu trong biến (có thể có hoặc chưa có)

Khái niệm kiểu dữ liệu

- Các bạn quan sát ở trên bàn có rất nhiều các loại thức ăn khác nhau như thị, rau, nước chấm, nước hoa quả…đây chính là các loại kiểu liệu khác nhau vì chúng có các đặc tính vật lý khác nhau.
- Chúng ta cần có các vật chứa khác nhau để chứa các loại thức ăn khác nhau: như cốc để đựng nước hoa quả, đĩa để đựng thịt, nồi để đựng nước lẩu, bát để đựng nước chấm → Đây chính là các biến để lưu trữ dữ liệu.
- Như vậy với mỗi loại dữ liệu chúng ta cần một kiểu dữ liệu tương ứng để lưu trữ loại dữ liệu đó. Ví dụ: Không thể để nước hoa quả vào một cái đĩa hay thịt vào một cái cốc nước. Vẫn được nhưng không ai làm thế cả.

-

Cách khai báo biến



Cách khai báo biến với các kiểu dữ liệu tiêu biểu

```
float pi = 3.1415;

bool isGPSEnabled = true;

string myName = "Denis";

char at = '@';
```

Bài 5: Tìm hiểu về các kiểu dữ liệu hay dùng nhất.

Khai báo giá trị với giá trị mặc định

1. Gán giá trị mặc định cho biến khi khai báo

```
public class Lecture {

    int age = 15; // This is a variable of type integer

    public static void Main(string[] args){

        Console.WriteLine(age); // Output will be 15

    }

}
```

2. Gán lại giá trị cho biến

```
public class Lecture {
    int age = 15;
    public static void Main(string[] args){
        age = 20; // New value gets assigned
        Console.WriteLine(age); // Output will be 20
    }
```

3. Khai báo mà không gán thì biến sẽ có giá trị là giá trị mặc định của kiểu dữ liệu đó.

```
public class Lecture {
    int age; // default value assigned = 0
    public static void Main(string[] args){
        Console.WriteLine(age); // Output will be 0
    }
}
```

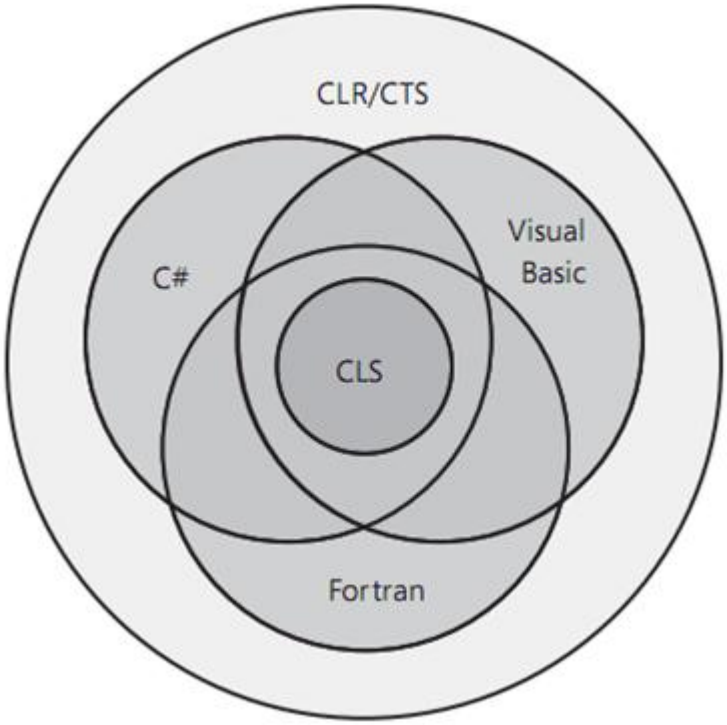
Khái niệm CLS và CTS

4. Common Type System (CTS): .NET framework hỗ trợ nhiều ngôn ngữ và đều dùng một thành phần gọi là hệ thống kiểu chung CTS trong CLR. CTS hỗ trợ một loạt kiểu và toán tử có thể thấy trong hầu hết các ngôn ngữ lập trình nên gọi một ngôn ngữ từ một ngôn ngữ khác sẽ không yêu

cầu chuyển kiểu. Dẫn đến chúng ta có thể xây dựng các ứng dụng .NET sử dụng cả ngôn ngữ VB.NET lẫn C#, C++…

5.

6. Common Language Specification (CLS): Đặc tả ngôn ngữ chung CLS là một tập con của CTS, nó định nghĩa một tập các quy tắc cho phép liên kết hoạt động trên nền tảng .NET. Các quy tắc này sẽ trợ giúp và chỉ dẫn cho các nhà thiết kế compiler của hãng thứ 3 hoặc những người muốn xây dựng thư viện dùng chung.



7.

Bảng tổng hợp một số kiểu dữ liệu cơ bản

Nhóm	Kiểu dữ liệu	Kích thước (bytes)	Ý nghĩa	.NET Type (CTS)
Kiểu số nguyên	byte	1	Số nguyên dương **không dấu** có giá trị từ **0** đến **255.**	System.Byte

(Giá trị mặc định là 0)	sbyte	1	Số nguyên có dấu có giá trị từ **-128** đến **127**	System.SByte
	short	2	Số nguyên có dấu có giá trị từ **-32,768** đến **32,767**	System.Int16
	ushort	2	Số nguyên **không dấu** có giá trị từ **0** đến **65,535**	System.UInt16
	int	4	Số nguyên có dấu có giá trị từ **-2,147,483,647** đến **2,147,483,647**	System.Int32
	uint	4	Số nguyên **không dấu** có giá trị từ **0** đến **4,294,967,295**	System.UInt32
	long	8	Số nguyên có dấu có giá trị từ **-9,223,370,036,854,775,808** đến **9,223,370,036,854,775,807**	System.Int64
	ulong	8	Số nguyên **không dấu** có giá trị từ **0** đến **18,446,744,073,709,551,615**	System.UInt64
Kiểu ký tự (Giá trị mặc định là `'\0'`)	char	2	Chứa một ký tự Unicode	System.Char
Kiểu logic (Giá trị mặc định là false)	bool	1	Chứa 1 trong 2 giá trị logic là **true** hoặc **false**	System.Boolean
Kiểu số thực (Giá trị mặc định là 0)	float	4	Kiểu số thực dấu chấm động có giá trị dao động từ **±1.5 x 10^{-45} to ±3.4 x 10^{38}**, với 7 chữ số có nghĩa. Thường sử dụng cho các thư viện đồ hoạ cần yêu cầu sức mạnh xử lý cao.	System.Single

	double	8	Kiểu số thực dấu chấm động có giá trị dao động từ **±5.0 × 10^{-324} to ±1.7 × 10^{308}**, với 15, 16 chữ số có nghĩa. Thường dùng cho các tính toán thực tế phổ biến ngoại trừ tài chính.	System.Double
	decimal	16	Kiểu số thực có dấu chấm động có giá trị giao động từ **±1.0 x 10^{-28} to ±7.9228 x 10^{28}**. Có độ chính xác đến 28 con số và giá trị thập phân, được dùng trong tính toán tài chính. Thường dùng cho các ứng dụng tài chính có độ chính xác cao.	System.Decimal

Kiểu string

- Kiểu string khác với các kiểu trên là kiểu dữ liệu tham chiếu dùng để lưu chuỗi ký tự văn bản.
- Nếu không gán giá trị thì mặc định giá trị của kiểu String sẽ là null

Lưu ý

- Kiểu dữ liệu có miền giá trị lớn hơn sẽ chứa được kiểu dữ liệu có miền giá trị nhỏ hơn. Như vậy biến kiểu dữ liệu nhỏ hơn có thể gán giá trị qua biến kiểu dữ liệu lớn hơn (sẽ được trình bày trong phần tiếp theo).
- Giá trị của kiểu char sẽ nằm trong dấu **‘ ’** (nháy đơn).
- Giá trị của kiểu string sẽ nằm trong dấu **“ ”** (nháy kép).
- Giá trị của biến kiểu float phải có chữ **F** hoặc **f** làm hậu tố.
- Giá trị của biến kiểu decimal phải có chữ **m** hoặc **M** làm hậu tố.
- Trừ kiểu string, tất cả kiểu dữ liệu trên đều không được có giá trị null:
  - Null là giá trị rỗng, không tham chiếu đến vùng nhớ nào.
  - Để có thể gán giá trị null cho biến thì ta thêm ký tự ? vào sau tên kiểu dữ liệu là được. Ví dụ: int? hay bool? . . .

Bài 6: Thực hành về kiểu int, float, double

```
 int number1 = 10;
int number2 = 12;
int number3, number4, number5;

int sum = number1 + number2;

Console.WriteLine("Sum of " + number1 + " and " + number2 + ": " + sum);
```

```
            float floatNumber1 = 1.2f;
            float floatNumber2 = 1.3f;

            float sumFloat = floatNumber1 + floatNumber2;
            float divFloat = floatNumber1/ floatNumber2;
            float multiFloat = floatNumber1 * floatNumber2;

            Console.WriteLine("Sum of " + floatNumber1 + " and " + floatNumber2 + ":
" + sumFloat);
            Console.WriteLine("Division of " + floatNumber1 + " for " + floatNumber2
+ ": " + divFloat);
            Console.WriteLine("Multiple of " + floatNumber1 + " for " + floatNumber2
+ ": " + multiFloat);


            Console.Read();
```

Bài 7: Kiểu dữ liệu String

```
            string myName = "Bach Ngoc Toan";
            string[] myNames  = myName.Split(' ');
            Console.WriteLine("My name is: " + myName);
            Console.Read();
```

Bài 8: Quy tắc code chuẩn

1. Quy tắc đặt tên
    a. Tên file
    b. Tên biến
    c. Tên phương thức
    d. Tên class
2. Comment
    a. Comment 1 dòng
    b. Comment nhiều dòng
    c. Comment XML

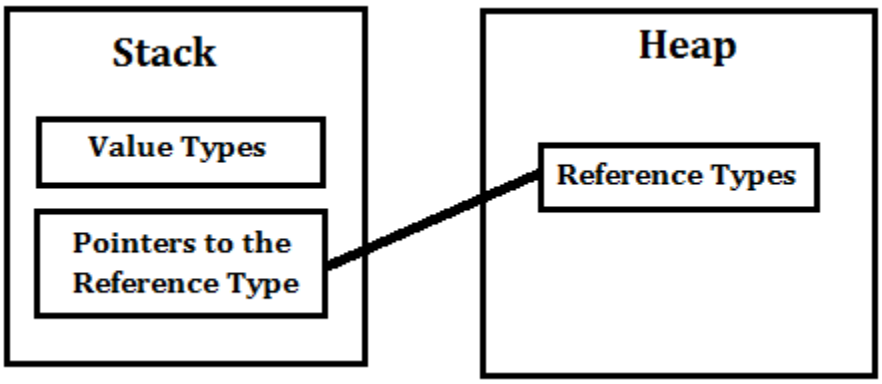
Tham khảo: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

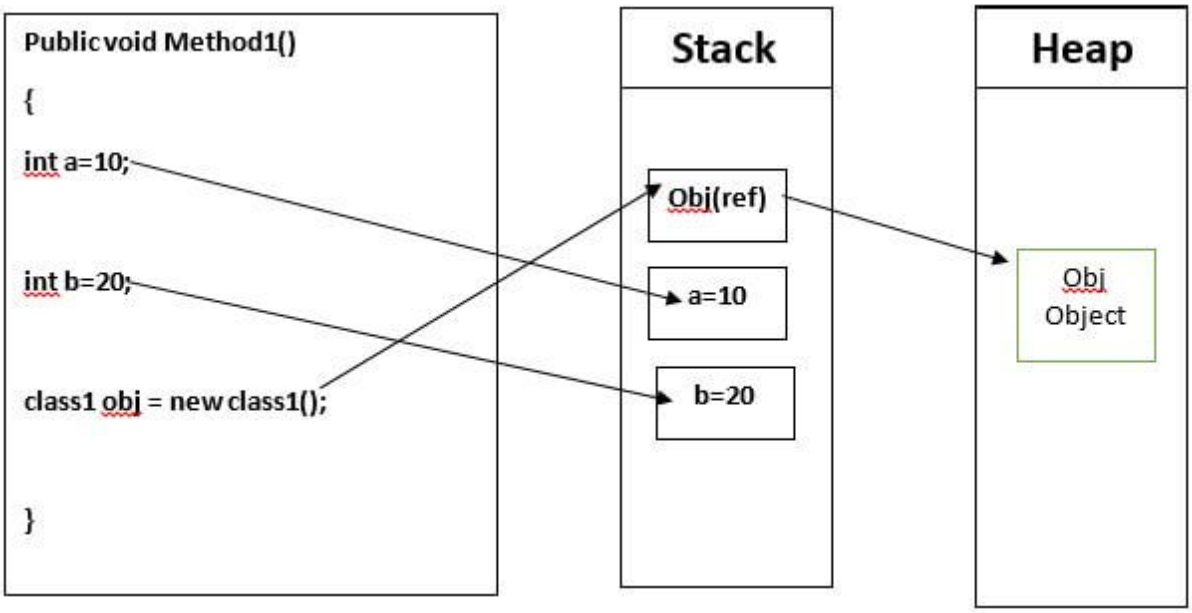
https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md

Bài 9: Kiểu tham trị và kiểu tham chiếu

Trong các kiểu của .NET Framework chúng ta có 2 kiểu là Value Type (kiểu tham trị) và Reference Type (kiểu tham chiếu).

- Kiểu tham trị lưu trực tiếp dữ liệu trong bộ nhớ Stack
- Kiểu tham chiếu chỉ lưu địa chỉ trong Stack còn giá trị biến nằm ở nơi khác (Heap)





Kiểu tham trị (Value Type)

Một kiểu tham trị lưu nội dung của nó trong bộ nhớ cấp phát là Stack. Khi chúng ta tạo một biến kiểu tham trị thì một vùng nhớ sẽ được cấp phát để lưu giá trị của biến một cách trực tiếp.

Nếu bạn gán nó cho một biến khác thì giá trị sẽ được copy trực tiếp và cả 2 biến sẽ làm việc độc lập.

Các kiểu dữ liệu tham trị trong .NET:

- Các kiểu số nguyên
- Kiểu số thực
- Kiểu logic bool
- Kiểu ký tự char
- Kiểu struct
- Enum

Kiểu tham chiếu

Kiểu tham chiếu được dùng để lưu một giá trị tham chiếu (địa chỉ ô nhớ) của một đối tượng nhưng không lưu trữ đối tượng đó.

Bởi vì kiểu tham chiếu chỉ lưu địa chỉ của ô nhớ của biến thay vì lưu giá trị của biến, nên khi gán một biến tham chiếu cho một biến khác thì nó không copy data mà nó chỉ copy địa chỉ tham chiếu.

Nên cả 2 biến sẽ cùng tham chiếu đến một địa chỉ giống nhau trên bộ nhớ Heap.

Điều này có nghĩa khi một biến tham chiếu không được dùng nữa, nó sẽ được đánh dấu cho Garbage collection.

Các kiểu dữu liệu tham chiếu:

- Classs
- Object
- Array
- Indexer
- Interface

Bộ nhớ Stack và Heap

Stack là bộ nhớ cấp phát tĩnh còn Heap là bộ nhớ cấp phát động, cả 2 đều được lưu trữ trên RAM của máy tính.

Bạn có thể dùng Stack nếu bạn biết chắc độ lớn của dữ liệu cần lưu trước khi biên dịch chương trình, nó không được quá lớn.

Bạn có thể dùng heap nếu bạn không biết chính xác độ lớn dữ liệu bạn sẽ cần ở lúc chạy hoặc nếu bạn cần cấp phát rất nhiều dữ liệu.

Trong ứng dụng mutilple thread, mỗi một thread sẽ có một stack riêng nhưng chúng sẽ chia sẻ bộ nhớ heap. Stack là riêng cho từng thread còn heap là chia sẻ cho toàn ứng dụng.

So sánh giữa Stack và Heap

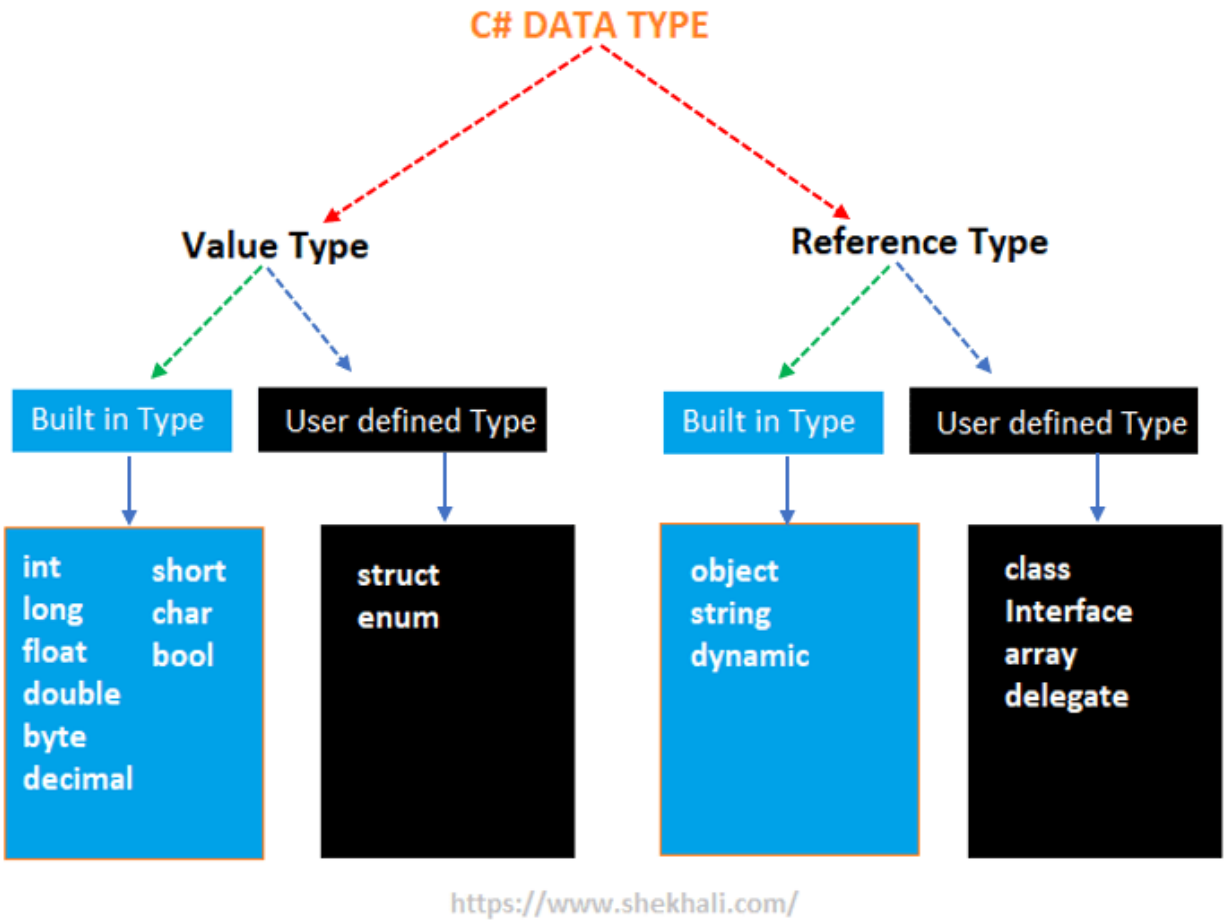
Stack	Heap
Bộ nhớ được quản lý tự động	Bộ nhớ được quản lý bằng tay
Kích cỡ nhỏ	Kích cỡ lớn
Truy cập dễ dàng và nhanh chóng, dễ dàng cache	Khó cache vì bị phân tán trong bộ nhớ
Không linh hoạt, bộ nhớ cấp phát không thể thay đổi	Lich hoạt, bộ nhớ cấp phát có thể đổi
Giới hạn trong phạm vi thread	Được truy cập toàn bộ ứng dụng
Hệ điều hành cấp phát stack khi thread được tạo	Hệ điều hành được gọi bởi ngôn ngữ lúc chạy ứng dụng để cấp phát heap cho ứng dụng.

So sánh cấp phát tĩnh và cấp phát động

Cấp phát tĩnh	**Cấp phát động**
Kích thước phải biết lúc biên dịch	Không biết kích thước biến lúc biên dịch
Thực hiện lúc biên dịch	Thực hiện lúc runtime

Được gán cho stack	Gán cho heap
FILO (First – in, last- out)	Không có thứ tự

Tóm lại



Chúng ta có 2 kiểu dữ liệu là tham trị và tham chiếu:

1. Tham trị là các kiểu dữ liệu kích thước nhỏ, size cố định giá trị lưu trực tiếp trên bộ nhớ Stack.
2. Tham chiếu là kiểu chỉ lưu địa chỉ trên stack và giá trị lưu trên heap.
3. Cả 2 loại bộ nhớ này đều được lưu trên RAM, nó chỉ là khác vùng thôi. Một bên là cấp phát tĩnh 1 bên là cấp phát động.

Tại sao lại phải chia ra thế?

Vì với kiểu dữ liệu lớn, như là object thì không thể biết kích thước của nó lúc biên dịch nên không lưu trên stack mà chỉ lưu địa chỉ của Heap thôi, vì heap là bộ nhớ cấp phát động.

Ví dụ: Ta có đọc 1 cuốn sách Harry Porter, thay vì tôi nói tôi đã đọc cuốn sách **Harry Porter 1** thì tôi phải nói là **Tôi đọc …. từ câu đầu cho đến câu cuối…**bạn có biết không? Thay vì chúng ta phải nêu ra toàn bộ nội dung cuốn sách rất dài thì ta chỉ cần refer đến tên cuốn sách, chính là địa chỉ ô nhớ trỏ đến nội dung cuốn sách. Còn cuốn sách có nội dung như thế nào thì bạn đó sẽ tự tìm đọc.

Còn những câu văn ngắn thì chúng ta có thể nói luôn trong cuộc trò chuyện. Ví dụ: Bạn ăn cơm chưa? Thì câu đó không nằm trong cuốn sách nào hoặc nó rất ngắn nên có thể nói luôn trong cuộc hội thoại.

Đó là kiểu tham trị.

Bài 10: Nhập xuất cơ bản với màn hình Console

Ứng dụng Console là gì?

Trích dẫn nguyên văn: "A console application, in the context of C#, is an application that takes input and displays output at a command line console with access to three basic data streams: standard input, standard output and standard error. A console application facilitates the reading and writing of characters from a console - either individually or as an entire line. It is the simplest form of a C# program and is typically invoked from the Windows command prompt. A console application usually exists in the form of a stand-alone executable file with minimal or no graphical user interface (GUI)."

Nguồn: https://www.techopedia.com/definition/25593/console-application-c

Để tạo ứng dụng Console, chúng ta cần làm việc với Class System.Console. Trong đó, System là một namespace, Console là lớp bên trong namespace System.

Xuất dữ liệu với Console

Để xuất nội dung trong C#, chúng ta có thể sử dụng:

- System.Console.WriteLine()
- System.Console.Write()

Sự khác biệt cơ bản của WriteLine() và Write() là phương thức Write() chỉ in chuỗi được cung cấp, trong khi phương thức WriteLine() in chuỗi và chuyển đến đầu dòng tiếp theo.

```
Program.cs
HelloWorld                                  HelloWorld.Program
1   using System;
2
3   namespace HelloWorld
4   {
        0 references
5       internal class Program
6       {
            0 references
7           public static void Main(string[] args)
8           {
9               Console.WriteLine("Prints on ");
10
11              Console.WriteLine("New line");
12
13              Console.Write("Prints on ");
14
15              Console.Write("Same line");
16          }
17
18      }
19  }
```

```
C:\Users\NGOC TOAN\source\repos\HelloWorld\HelloWorld\bin\Debug\net6.0\HelloWorld.exe
Prints on
New line
Prints on Same line
```

In Variable và Literal sử dụng phương thức WriteLine() và Write()

Phương thức WriteLine() và phương thức Write() có thể sử dụng để in biến và hằng. Hãy xem ví dụ dưới đây:

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        int value = 10;

        // Variable
        Console.WriteLine(value);

        // Literal
        Console.WriteLine(50.05);

        Console.Read();
    }

}
```

```
C:\Users\NGOC TOAN\source\repos\HelloWorld\HelloWorld\bin\Debug\net6.0\HelloWorld.exe
10
50.05
```

Kết hợp (nối) hai chuỗi sử dụng toán tử cộng (+) và in chúng

Các chuỗi có thể được kết hợp / nối (concatenated string) bằng cách sử dụng toán tử + trong khi in.

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        int val = 55;

        Console.WriteLine("Hello " + "World");

        Console.WriteLine("Value = " + val);

        Console.Read();
    }

}
```

```
C:\Users\NGOC TOAN\source\repos\HelloWorld\HelloWorld\bin\Debug\net6.0\HelloWorld.exe
Hello World
Value = 55
```

In chuỗi kết hợp bằng chuỗi đã được định dạng (Formatted String)

Một giải pháp tốt hơn để in chuỗi kết hợp là sử dụng chuỗi đã được định dạng. Formatted string cho phép lập trình viên sử dụng trình giữ chỗ (placeholder) cho các biến.

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        int val = 55;

        Console.WriteLine("Value = " + val);

        //Có thể thay thế bởi:
        Console.WriteLine("Value = {0}", val);

        Console.Read();
    }

}
```

{0} là trình giữ chỗ cho biến val , nó sẽ bị thay thế bởi giá trị của val. Chỉ có 1 biến được sử dụng nên chỉ có một placeholder.

Ví dụ dưới đây là nhiều biến:

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        int firstNumber = 5, secondNumber = 10, result;

        result = firstNumber + secondNumber;

        Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);

        Console.Read();
    }

}
```

Khi chạy chương trình, output sẽ hiển thị

5 + 10 = 15

Ở đây, {0} bị thay thế bởi firstNumber, {1} bịthay thế bởi secondNumber và {2} bị thay thế bởi result.

Cách tiếp cận output in này dễ đọc hơn và ít bị lỗi hơn toán tử +.

Nhập dữ liệu với Console

Trong C#, cách đơn giản nhất để lấy input từ user là sử dụng phương thức ReadLine() của lớp Console. Tuy nhiên, Read() và ReadKey() cũng có thể làm điều đó được. Hai phương thức này cùng thuộc lớp Console.

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        string testString;

        Console.Write("Enter a string - ");

        testString = Console.ReadLine();

        Console.WriteLine("You entered '{0}'", testString);

        Console.Read();
    }

}
```

```
C:\Users\NGOC TOAN\source\repos\HelloWorld\HelloWorld\bin\Debug\net6.0\HelloWorld.exe
Enter a string - Hello World
You entered 'Hello World'
```

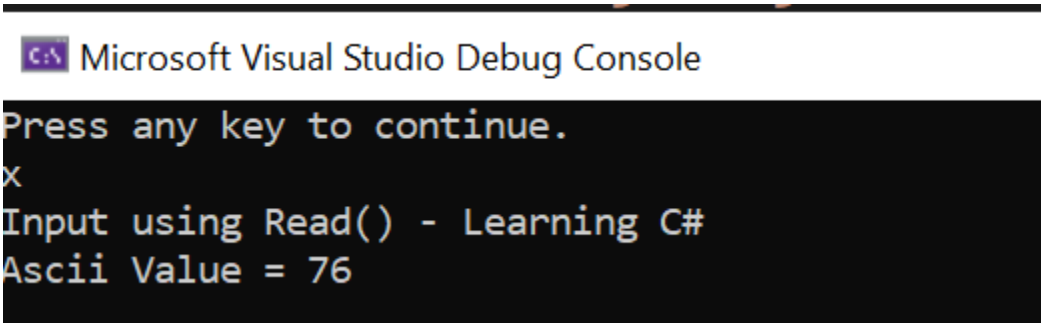
Sự khác nhau của phương thức ReadLine(), Read() và ReadKey()

- **ReadLine():** Phương thức ReadLine() đọc dòng tiếp theo của input (từ dòng input chuẩn). Nó trả về cùng một chuỗi.
- **Read():** Phương thức Read() đọc ký tự tiếp theo từ dòng input chuẩn. Nó trả về giá trị Ascii của ký tự.

- **ReadKey():** Phương thức ReadKey() tiếp nhận phím tiếp theo mà user nhấn. Phương pháp này thường được sử dụng để giữ màn hình không hiển thị output cho đến khi người dùng nhấn một phím.

Ví dụ: Sự khác nhau của phương thức Read() và ReadKey():

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        int userInput;

        Console.WriteLine("Press any key to continue…");

        Console.ReadKey();

        Console.WriteLine();

        Console.Write("Input using Read() - ");

        userInput = Console.Read();

        Console.WriteLine("Ascii Value = {0}",userInput);

        Console.Read();
    }

}
```



Từ ví dụ trên, bạn có thể thấy cách hoạt động của Console.ReadKey() và Console.Read(). Khi sử dụng Console.ReadKey(), khi nhấn phím bất kỳ màn hình sẽ hiển thị kết quả.

Khi sử dụng Console.Read(), bạn sẽ phải viết cả dòng lệnh trong khi nó chỉ trả về giá trị ASCII value của ký tự đầu tiên. Trong ví dụ này, 76 là giá trị ASCII của L.

Đọc giá trị số

Việc đọc một ký tự hoặc chuỗi rất đơn giản trong C #. Tất cả những gì bạn cần làm là gọi đúng tên các phương thức tương ứng.

Tuy nhiên, việc đọc các giá trị số có thể khá phức tạp.

Chúng ta sẽ vẫn sử dụng cùng một phương thức ReadLine(). Nhưng vì phương thức này nhận đầu vào là chuỗi, nên nó cần được chuyển đổi thành kiểu số nguyên hoặc dấu chấm động.

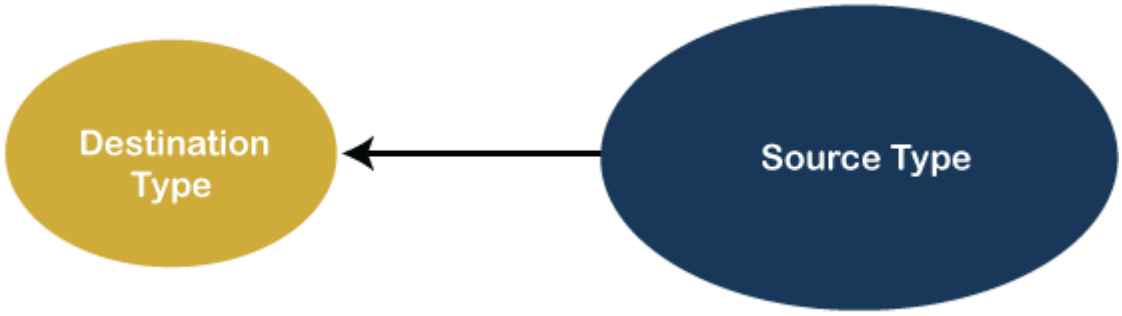
Một cách tiếp cận đơn giản để chuyển đổi đầu vào là sử dụng các phương thức của lớp Convert .

```
0 references
internal class Program
{
    0 references
    public static void Main(string[] args)
    {
        string userInput;

        int intVal;

        double doubleVal;

        Console.Write("Enter integer value: ");

        userInput = Console.ReadLine();

        /* Converts to integer type */

        intVal = Convert.ToInt32(userInput);

        Console.WriteLine("You entered {0}", intVal);

        Console.Write("Enter double value: ");

        userInput = Console.ReadLine();

        /* Converts to double type */

        doubleVal = Convert.ToDouble(userInput);

        Console.WriteLine("You entered {0}", doubleVal);

        Console.Read();
    }

}
```

Phương thức **ToInt32()** và **ToDouble()** của Convert class chuyển đổi input thành kiểu integer và double. Tương tự, chúng ta có thể chuyển đổi input sang các kiểu khác.

Bài 11: Chuyển đổi kiểu dữ liệu (Type Casting)

Chuyển đổi kiểu dữ liệu là khi bạn muốn gán giá trị của một kiểu dữ liệu này sang kiểu dữ liệu khác



Trong C# có 2 cách chuyển đổi kiểu dữ liệu:

Implicit Casting (chuyển đổi ngầm định tự động)

Chuyển đổi kiểu dữ liệu range nhỏ hơn sang kiểu dữ liệu range lớn hơn.

`char -> int -> long -> float -> double`

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double

Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9
```

Explicit Casting (chuyển đổi tường minh bằng tay)

Chuyển đổi từ kiểu dữ liệu lớn hơn sang kiểu dữ liệu nhỏ hơn

`double -> float -> long -> int -> char`

```
int five = 5;
var doubleFive = (double)five;

char a = 'a';
var valueA = (int)a;

float myFloat = 4.56F;
decimal myMoney = (decimal)myFloat;
```

Khác nhau giữa casting, parsing và converting

Casting: là chuyển 1 giá trị từ kiểu này sang kiểu khác hoặc xuất ra lỗi

```
int five = 5;
var doubleFive = (double)five;

char a = 'a';
var valueA = (int)a;

float myFloat = 4.56F;
decimal myMoney = (decimal)myFloat;
```

Conversion: Là cố chuyển một kiểu đối tượng sang kiểu khác, ít lỗi hơn nhưng chậm hơn

```
int five = 5;
decimal decFive = Convert.ToDecimal(five);

decimal myMoney = 5.67M;
int intMoney = Convert.ToInt32(myMoney); //Value is now 6;
                                         //the decimal value was rounded
```

Parsing: Là cố chuyển một chuỗi sang một kiểu nguyên thuỷ

```
string testString = "10.22.2000";
double decValue = double.Parse(testString); //Exception thrown here!

string intTest = "This is a test string";
int intValue = int.Parse(intTest); //Exception thrown here!

string value = "5.0";
decimal result;


bool isValid = decimal.TryParse(value, out result);
```

Khác nhau giữa Parse và Convert

```
string s1 = "1234";
string s2 = "1234.65";
string s3 = null;
string s4 = "123456789123456789123456789123456789123456789";
result = Int32.Parse(s1);    //1234
result = Int32.Parse(s2);    //FormatException
result = Int32.Parse(s3);    //ArgumentNullException
result = Int32.Parse(s4);    //OverflowException

result = Convert.ToInt32(s1);    // 1234
result = Convert.ToInt32(s2);    // FormatException
result = Convert.ToInt32(s3);    // 0
result = Convert.ToInt32(s4);    // OverflowException
```

TryParse()

Trong trường hợp chúng ta không biết liệu một string có thể chuyển sang kiểu nào đó hay không thì chúng ta có thể dùng TryParse()

```
string value = "5.0";
decimal result;
bool isValid = decimal.TryParse(value, out result);
```

Nếu giá trị biến isValid là true có nghĩa là chuỗi có thể chuyển thành công, còn ngược lại thì là fail. Từ khoá out chúng ta sẽ học trong phần sau.

Từ khoá mới

Từ khoá is: Sử dụng để kiểm tra nếu một giá trị cụ thể là một kiểu cụ thể

```
var myValue = 6.5M; //M literal means type will be decimal
if (myValue is decimal) { /*...*/ }
```

Từ khoá as: Sử dụng để chuyển một object từ một kiểu sang một kiểu khác

Ví dụ 1:

```
string testString = "This is a test"; //string is a reference type
object objString = (object)testString; //Cast the string to an object

string test2 = objString as string; //Will convert to string successfully
```

Ví dụ 2:

```
public class ClassA { /*...*/ }
public class ClassB { /*...*/ }

var myClass = new ClassA();
var newClass = myClass as ClassB; //Exception thrown here!
```

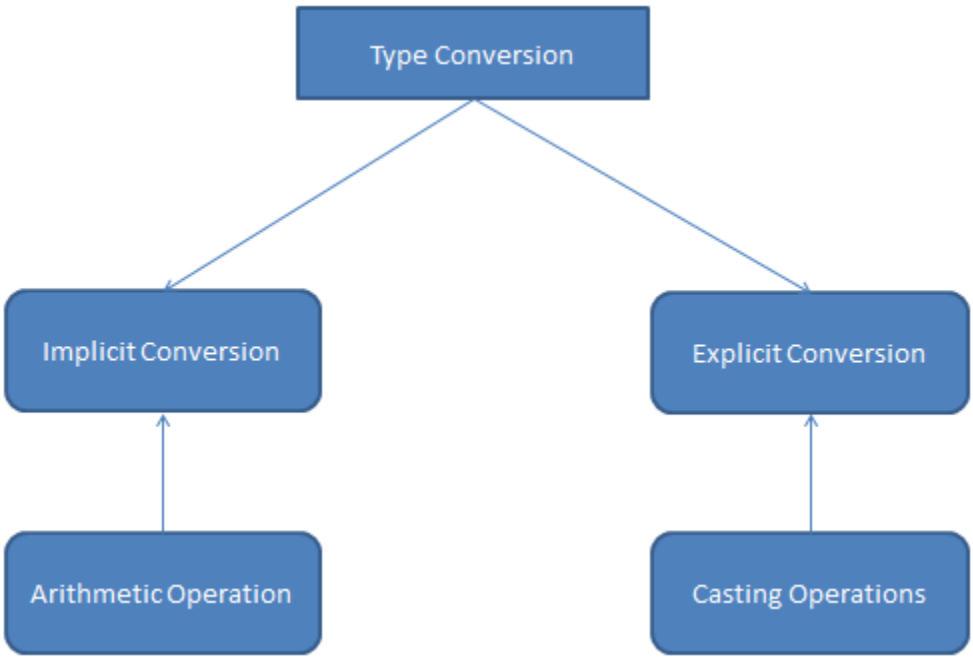
Ví dụ 3:

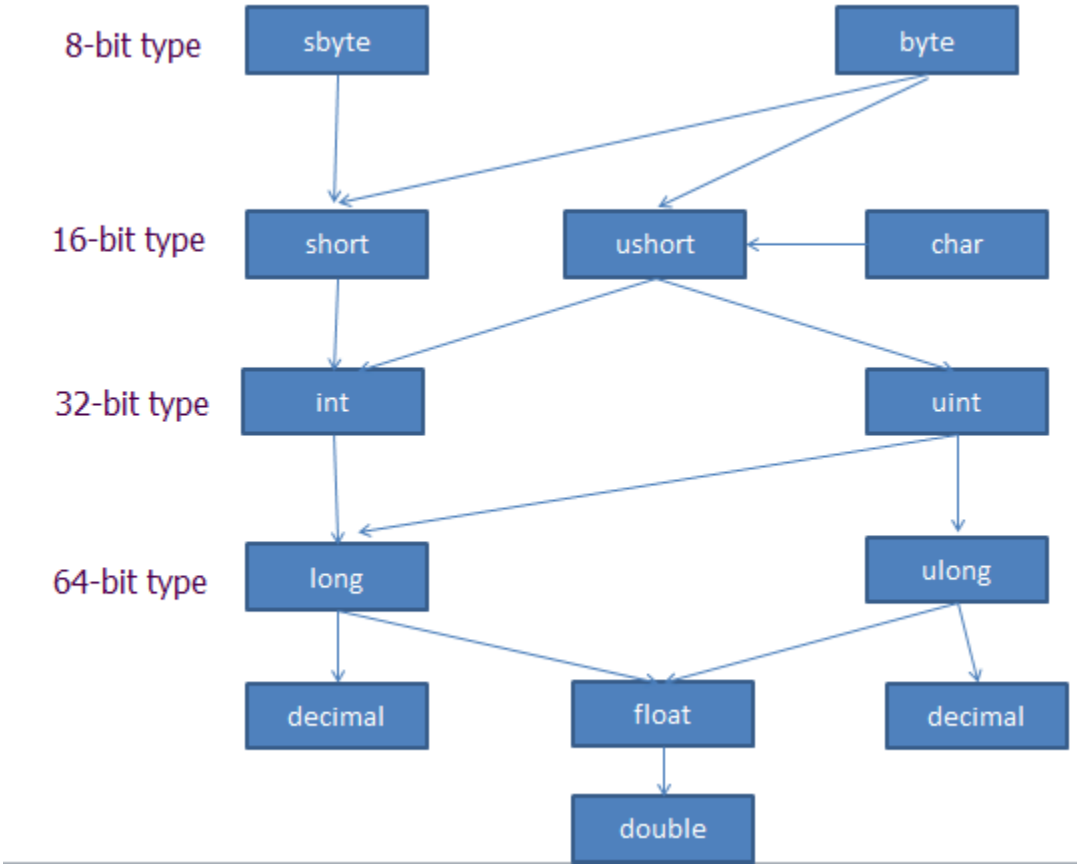
```
public class ClassA { /*...*/ }
public class ClassB : ClassA { /*...*/ }

var myClass = new ClassB();
var convertedClass = myClass as ClassA;
```

Từ khoá typeof: Trả về kiểu của một đối tượng

```
var sentence = "This is a sentence.";
var type = sentence.GetType();
if (type == typeof(string)) { /*...*/ }
else if (type == typeof(int)) { /*...*/ }
```

Tổng kết





Bài 12: Hằng số (const)

Tham khảo tài liệu: https://docs.microsoft.com/vi-vn/dotnet/csharp/language-reference/keywords/const

1. Bạn sử dụng từ khoá const để khai báo một trường là hằng số hoặc một local constant là hằng số.
2. Trường constant (constant field) và local constant không phải là các biến, chúng không thể bị thay đổi.
3. Constant có thể là số, giá trị boolean, string hay một tham chiếu null.
4. Không tạo một constant cho việc hiển thị thông tin mà bạn có thể thay đổi bất cứ khi nào.

```
using System;

namespace HelloWorld
{
    internal class Program
    {
        // Constants as a fields
        const double PI = 3.14159;
        const int NumberOfWeeksInYear = 52;
        const int NumberOfMonthsInYear = 12;
        const string MyBirthDay = "2000-11-11";
        public static void Main(string[] args)
        {
```

```
            double radius = 10;
            Console.WriteLine("My birthday is {0}", MyBirthDay);
            Console.WriteLine(radius * radius * PI);
            Console.ReadKey();
        }

    }
}
```

Bài 13: Bài tập tổng kết chương 1

Bài tập 1: Viết chương trình tính tổng 2 số nguyên được nhập vào từ người dùng sau đó in ra dòng chữ: "Sum of <a> and <b> is: <total>". Trong đó giá trị của a, b và total là giá trị của biến.

Bài tập 2: Viết chương trình nhập vào họ tên, số điện thoại và giới tính để in ra thông tin người đó.

Bài tập 3: Viết chương trình tính diện tích hình tròn với bán kính được nhập vào từ bàn phím.

Review code email: tedu.international@gmail.com

Dừng lại suy nghĩ, sau đó mới được xem lời giải.

Bài 14: Methods

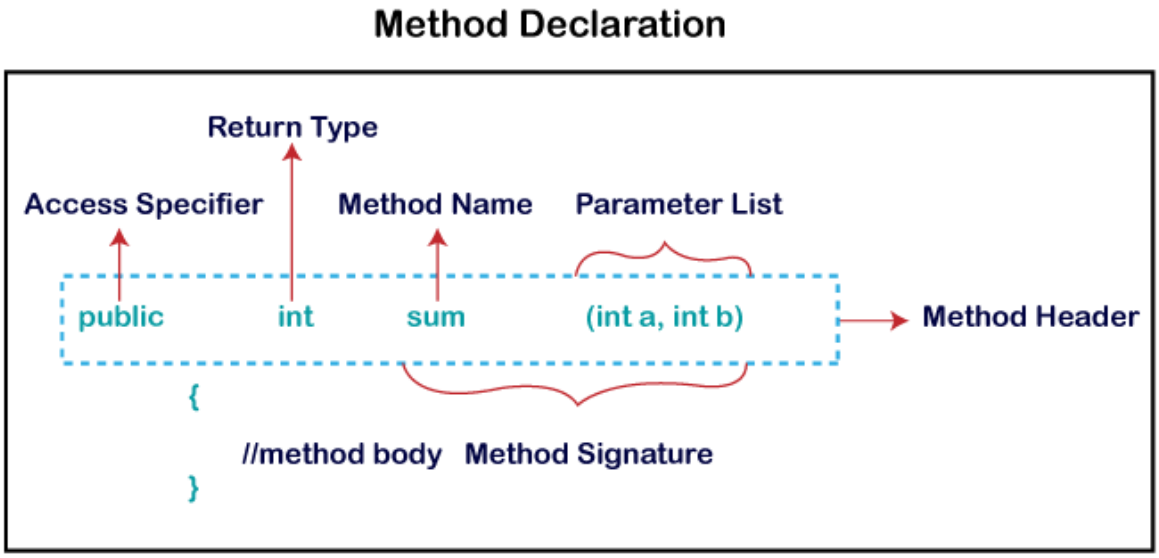
Phương thức là gì?

Phương thức là một khối lệnh chứa một tập hợp các dòng lệnh. Tập hợp lệnh đó sẽ được thực thi thông qua lời gọi phương thức và chỉ ra tham số cho phương thức đó (nếu có).

Trong C# thì tất cả các lệnh được thực thi đều phải được nằm trong một phương thức.

Phương thức Main là điểm khởi đầu (entry point) chủa tất cả các các ứng dụng C# và nó được gọi bởi CLR khi chương trình bắt đầu chạy.

Cú pháp khai báo



Trong đó:

- Access specifier: quyết định mức độ hiện diện của biến hoặc phương thức với class khác.
- Return type: Một phương thức có thể có 1 giá trị trả về, kiểu dữ liệu trả về của phương thức. Nếu phương thức không trả về bất cứ giá trị nào thì nó sẽ là void.
- Method name: Là tên của phương thức, nó phải là duy nhất trong class và có phân biệt hoa thường. Nó không thể trùng với bất cứ khai báo nào trong class
- Parameter list: Bao bởi cặp ngoặc đơn, tham số được sử dụng để truyền và nhận dữ liệu từ một phương thức. Danh sách tham số bao gồm kiểu, thứ tự và số lượng tham số của phương thức. Tham số là tuỳ chọn vì có thể có phương thức không có bất cứ tham số nào.
- Method body: Là phần chứa tập lệnh cần thiết để xử lý nghiệp vụ trong phương thức.

Ví dụ:

```
public int Add(int num1, int num2) {
    int result = num1 + num2;
    return result;
}
```

```
public int Add(int num1, int num2) {
    return num1 + num2;
}
```

Void method

```
static void Main(string[] args)
{
    ChangeName("John", "Doe");
}

1 reference
private static void ChangeName(string firstName, string lastName)
{
}
```

Ví dụ thực hành

- Viết chương trình máy tính, cộng trừ nhân chia 2 số sử dụng phương thức với tham số và giá trị trả về.

Bài 15: Exception Handler

Khái niệm Exception

Xử lý ngoại lệ trong C# có nghĩa là xử lý các lỗi exceptions có thể xảy ra, giúp chương trình không bị gián đoạn.

Trong đó Exceptions là một sự kiện xảy ra khi một chương trình đang chạy (thực thi) , sự kiện đó làm cho luồng xử lý thông thường của chương trình không thể thực hiện một cách bình thường, thậm chí chết chương trình.

Một Exception trong C# là một phản hồi về một tình huống ngoại lệ mà xuất hiện trong khi một chương trình đang chạy, ví dụ như chia cho số 0.

Từ khoá

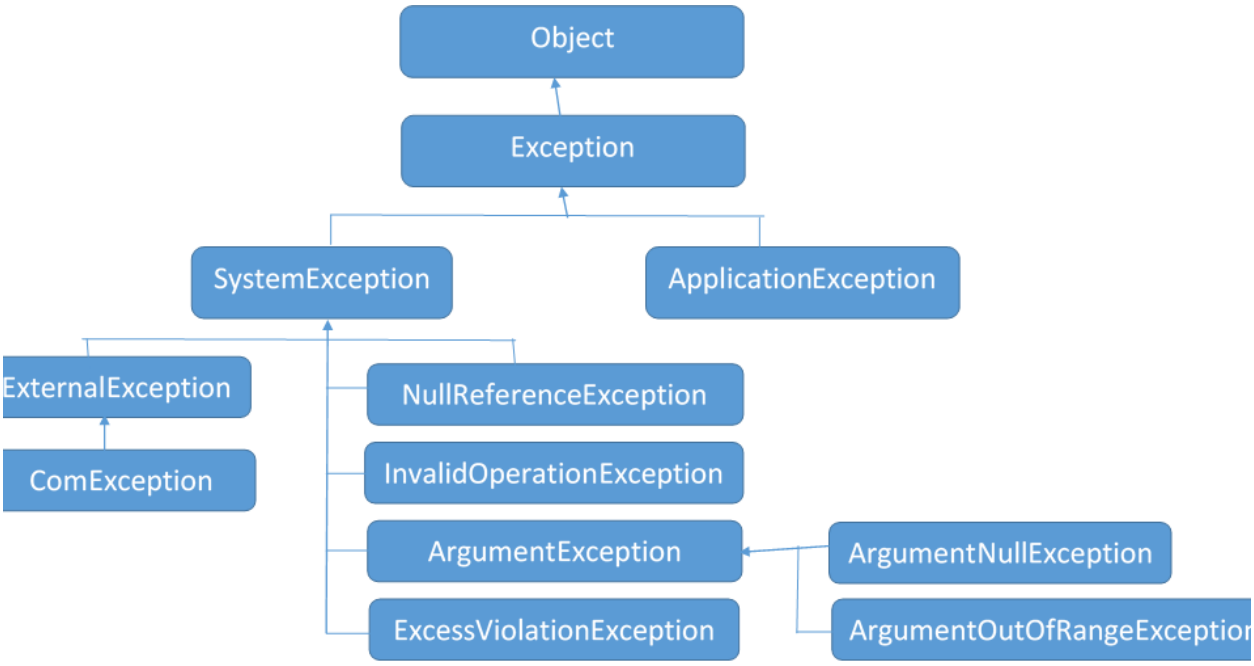
Exception cung cấp một cách để truyền điều khiển từ một phần của một chương trình tới phần khác. Xử lý ngoại lệ trong C# được xây dựng dựa trên 4 từ khóa là: try, catch, finally, và throw.

- **try**: Một khối try nhận diện một khối code mà ở đó các exception cụ thể được kích hoạt. Nó được theo sau bởi một hoặc nhiều khối catch.
- **catch**: Một chương trình bắt một Exception với một Exception Handler tại vị trí trong một chương trình nơi bạn muốn xử lý vấn đề đó. Từ khóa catch trong C# chỉ dẫn việc bắt một **exception**.
- **finally**: Một khối finally được sử dụng để thực thi một tập hợp lệnh đã cho, khối lệnh finally luôn luôn được thực thi dù có hay không một exception đươc ném hoặc không được ném. Ví dụ, nếu bạn mở một file, nó phải được đóng, nếu không sẽ có một exception được tạo ra.
- **throw**: Một chương trình ném một exception khi có một vấn đề xuất hiện. Điều này được thực hiện bởi sử dụng từ khóa throw trong C#.

Cú pháp xử lý

```
try
{
   // các lệnh có thể gây ra ngoại lệ (exception)
}
catch( tên_ngoại_lệ e1 )
{
   // phần code để xử lý lỗi
}
catch( tên_ngoại_lệ e2 )
{
   // phần code để xử lý lỗi
}
catch( tên_ngoại_lệ eN )
{
   // phần code để xử lý lỗi
}
finally
{
   // các lệnh được thực thi
}
```

Các lớp Exception của hệ thống



Ví dụ

Bài 16: Toán tử (operators)

- Toán tử được định nghĩa như sau:
  - Là một công cụ để thao tác với dữ liệu.
  - Một toán tử là một ký hiệu dùng để đại diện cho một thao tác cụ thể được thực hiện trên dữ liệu.
- Có 7 loại toán tử cơ bản:
  - Toán tử số học.
  - Toán tử logic.
  - Toán tử nhị phân.
  - Toán tử so sánh.
  - Toán tử gán.
  - Toán tử nối.
  - Toán tử chuyển đổi
  - Toán tử khác

Category	Operators
arithmetic	-, +, *, /, %, ++, --
logical	&&, ||, !, ^
binary	&, |, ^, ~, <<, >>
comparison	==,!=, >, <, >=, <=
assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
string concatenation	+
type conversion	(type), as, is, typeof, sizeof
other	., new, (), [], ?:, ??

Tham khảo: https://docs.microsoft.com/vi-vn/dotnet/csharp/language-reference/operators/

Bài 17: Cấu trúc điều khiển if else

Cấu trúc điều kiện là cấu trúc rẽ nhánh cho phép phân tách việc thực thi code thành nhiều hướng khác nhau tuỳ thuộc vào một điều kiện nào đó. Điều kiện này thông thường được xác định theo giá trị của biến hoặc biểu thức.

C# sử dụng 2 cấu trúc điều kiện là if-else và cấu trúc switch-case.

- Condition: Là một biểu thức hoặc 1 giá trị trả về dạng true/false.
- Statement 1: Là các lệnh sẽ được thực thi nếu condition có giá trị là true
- Statement 2: Là các lệnh sẽ được thực thi nếu condition có giá trị là false.

```
if (condition)
{
    statements 1
}
else
{
    statements 2
}
```

Một số lưu ý khi sử dụng if-else:

- Nếu statement 1 hoặc statement 2 chỉ có 1 lệnh duy nhất thì có thể không cần cặp ngoặc {}
- Nhánh else {} không bắt buộc, if thì bắt buộc phải có
- Bình thường bạn chỉ có thể tạo ra 2 nhánh rẽ: 1 nhánh if và 1 nhánh else.
- Để taoh thêm nhiều nhánh rẽ nữa, bạn có thể kết hợp thêm các nhánh else if vào cấu trúc trên. Số lượng else if không giới hạn.
- Bạn có thể lồng nhiều if-else với nhau.

Bài 18: Cấu trúc điều khiển Swicth case

Ở bài trước chúng ta đã tìm hiểu cấu trúc rẽ nhánh if-else. Cấu trúc này chỉ cho phép rẽ tới 2 nhánh. Nếu muốn rẽ nhiều nhánh bạn phải lồng ghép các nhánh else-if khiến code trở nên khó đọc.

C# cung cấp một cấu trúc khác để thực hiện rẽ nhiều nhánh thay cho việc lồng ghép nhiều if-else là cấu trúc switch-case.

```
switch(expression)
{
    case <value1>
        // code block
    break;
    case <value2>
        // code block
    break;
    case <valueN>
        // code block
    break;
    default
        // code block
    break;
}
```

Bài 19: Vòng lặp (loop)

Trong thực tế khi bạn cần thực thi một khối lệnh nhiều lần. Vòng lặp cho phép chúng ta thực thi một câu lệnh hoặc một khối lệnh nhiều lần.

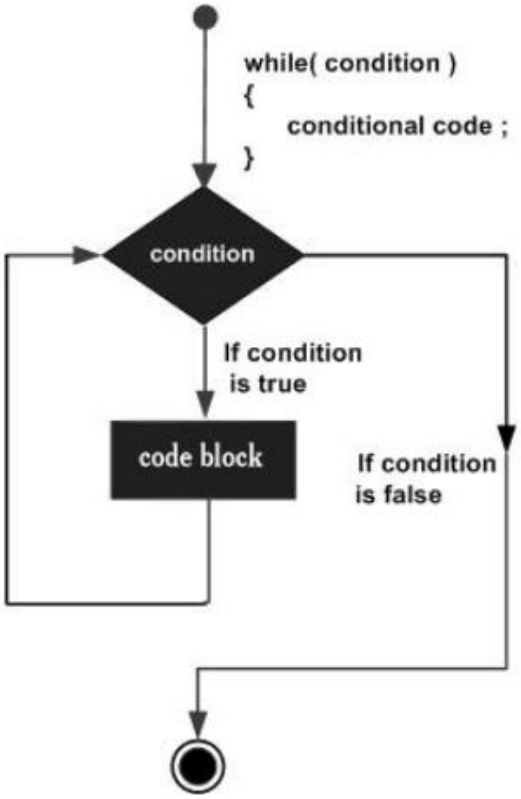
Các kiểu vòng lặp

Kiểu vòng lặp	**Mô tả**
While	Lặp lại một hoặc một nhóm các lệnh trong khi điều kiện đã cho là đúng. Nó kiểm tra điều kiện trước khi thực thi thân vòng lặp
For	Thực thi một dãy các lệnh nhiều lần và tóm tắt đoạn code mà quản lý biến vòng lặp
Do-while	Giống lệnh while ngoại trừ điểm là nó kiểm tra điều kiện ở cuối thân vòng lặp và luôn thực hiện vòng lặp đầu tiên dù điều kiện đúng hay không.

Foreach	Được sử dụng để duyệt lần lượt từng phần tử trong một tập hợp, mảng có sẵn
Vòng lặp lồng nhau	Bạn có thể sử dụng một hoặc nhiều vòng lặp trong các vòng lặp while, for hoặc do..while khác.

Vòng lặp while

Nếu biểu thức điều kiện theo sau while là đúng (true) thì khối lệnh bên trong vòng lặp được thực hiện và sau mỗi lần lặp biểu thức điều kiện được kiểm tra lại và nếu biểu thức điều kiện là sai (false) vòng lặp sẽ kết thúc.



Cú pháp:

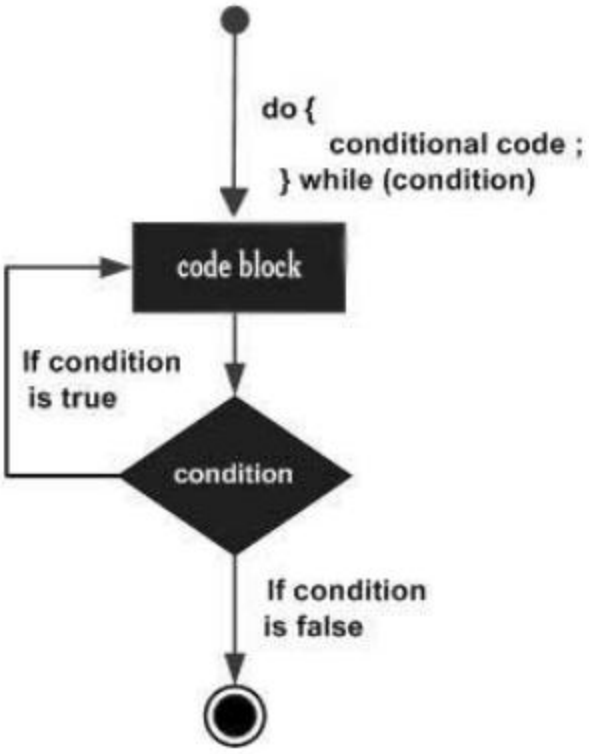
```
while (condition)
{
 // Thực hiện xử lý nếu condition là true
}
```

Ví dụ:

```
Console.WriteLine("While statement example");
int i = 1;
while(i <= 5)
{
 Console.Write(i + " ");
 i++;
}
```

Vòng lặp do…while

Vòng lặp do…while cũng tương tự như vòng lặp while tuy nhiên nó luôn luôn thực thi khối lệnh bên trong ít nhất một lần vì vòng lặp do…while kiểm tra điều kiện lặp ở cuối. Đó chính là điểm khác biệt duy nhất giữa vòng lặp while và vòng lặp do…while



Cú pháp:

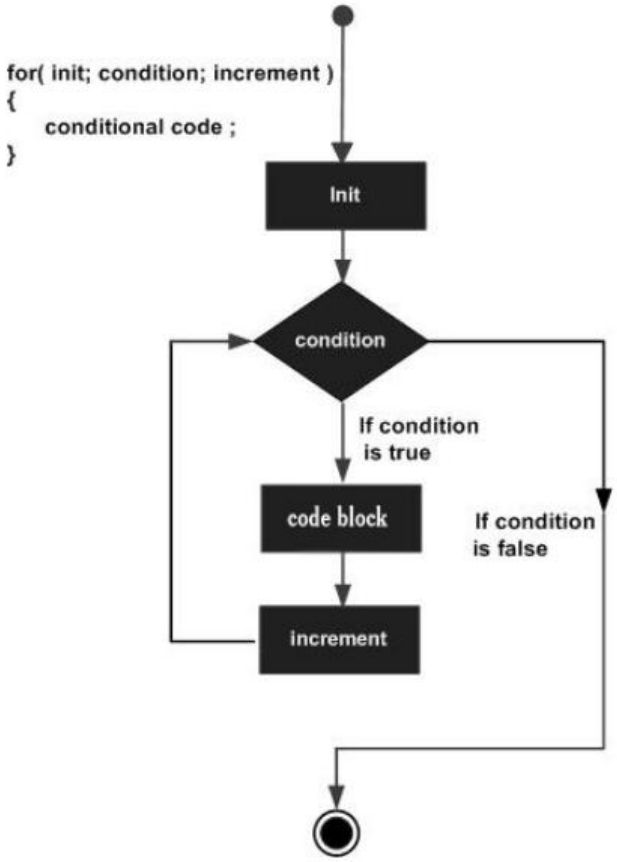
```
do
{
 // Tiếp tục thực thi xử lý nếu condition là true;
} while (condition);
```

Ví dụ:

```
Console.WriteLine("Do...While statement example");
int n;
do
{
  Console.Write("Please input your number: ");
  n = Convert.ToInt32(Console.ReadLine());
} while (n <= 0);
Console.Write("Alright");
```

Vòng lặp for

Tương tự như vòng lặp while, những câu lệnh bên trong vòng lặp for sẽ được thực thi nếu biểu thức điều kiện là đúng (true).



Cú pháp:

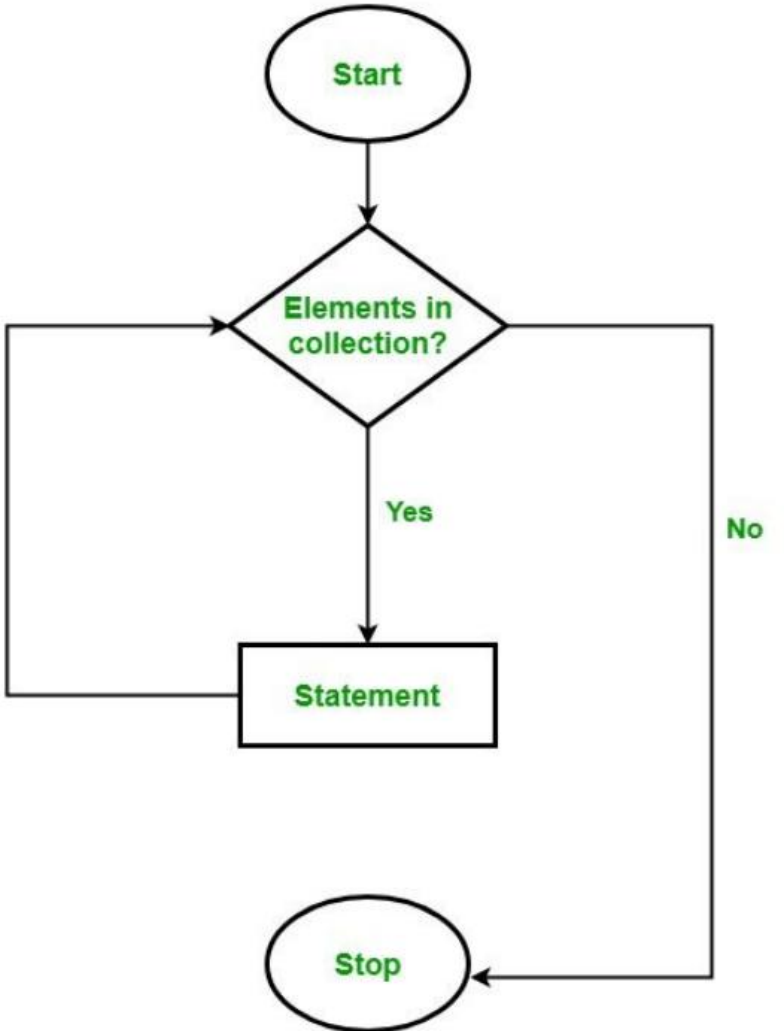
```
for (initialization; condition; increment/decrement)
{
  // Xử lý sẽ được thực thi nếu condition là true
}
```

Ví dụ:

```
Console.WriteLine("For statement example");
// int idx = 10 là initialization
// idx > 0 là condition
// idx-- là decrement
for (int idx = 10; idx > 0; idx--)
{
 Console.Write(idx + " ");
}
```

Vòng lặp foreach

Vòng lặp foreach thường được sử dụng để xử lý trên mảng hoặc trên collection để truy cập giá trị của các phần tử trong mảng hoặc collection.



Cú pháp:

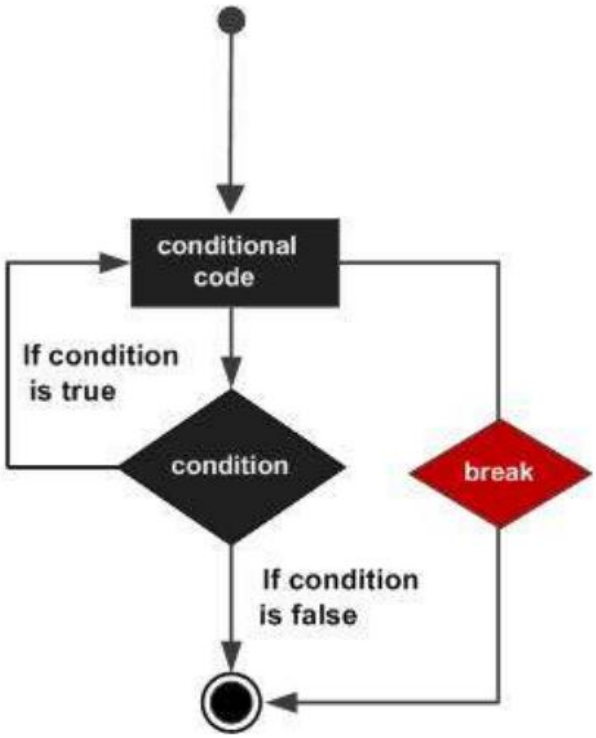
```
foreach (val in Array/Collection)
{
 // Xử lý trong khi chưa hết các phần tử trong mảng hoặc collection
}
```

Ví dụ:

```
Console.WriteLine("Foreach statement example");
int[] intArr = new int[10];
Random r = new Random();
// Khởi tạo giá trị cho các phần tử của mảng
for (int idx = 0; idx < 10; idx++)
{
 intArr[idx] = r.Next(1, 10);
}
// Hiển thị giá trị của các phần tử sử dụng foreach
Console.Write("Value of element: ");
foreach (int val in intArr)
{
 Console.Write(val + " ");
}
```

Câu lệnh break

Như cấc bạn đã biết, câu lệnh break được sử dụng trong switch…case để kết thúc switch…case. Trong vòng lặp câu lệnh break được sử dụng để kết thúc vòng lặp.

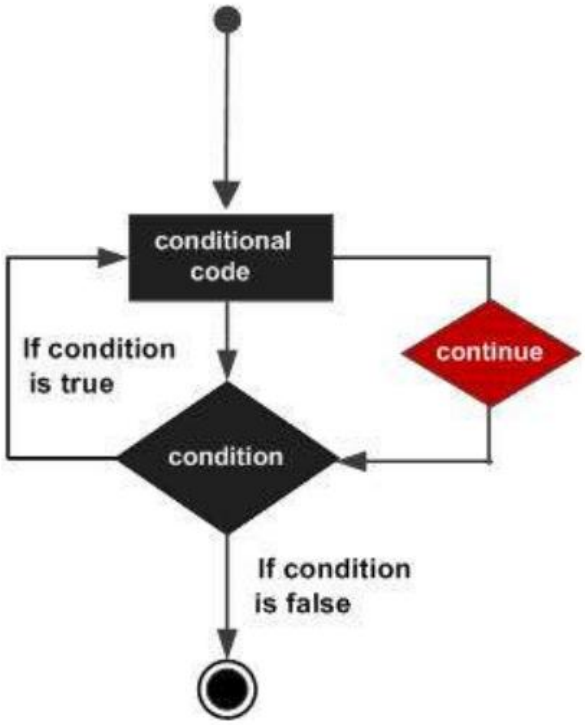


Ví dụ sử dụng câu lệnh break để kết thúc vòng lặp nếu có một phần tử nào đó chia hết cho 2:

```
Console.WriteLine("Break statement example");
int[] intArray = new int[10];
Random random = new Random();
// Khởi tạo mảng
for (int idx = 0; idx < 10; idx++)
{
 intArray[idx] = random.Next(1, 20);
 Console.Write(intArray[idx] + " ");
}
Console.Write("\nValue of element: ");
foreach (int e in intArray)
{
 if (e % 2 == 0)
 {
   break;
 }
 Console.Write(e + " ");
}
```

Câu lệnh continue

Nếu như câu lệnh break sẽ kết thúc vòng lặp thì câu lệnh continue sẽ bỏ qua những xử lý ở sau câu lệnh continue.



Ví dụ sử dụng câu lệnh continue để hiển thị các số lẻ từ 1 đến 10:

```
Console.WriteLine("Continue statement example");
for (int idx = 1; idx <= 10; idx++)
{
 if (idx % 2 == 0)
 {
  continue;
 }
 Console.Write(idx + " ");
}
```

Bài 20: Bài tập kết thúc chương 2

Bài tập 1: Viết chương trình giải phương trình bậc nhất (ax+b=0) với a,b nhập vào từ bàn phím.

Bài tập 2: Viết phương trình giải phương trình bậc 2 ($ax^2 + bx + c = 0$) với a,b,c nhập vào từ bàn phím.

Bài tập 3: Tính giai thừa của một số tự nhiên bất kỳ nhập vào từ bàn phím.

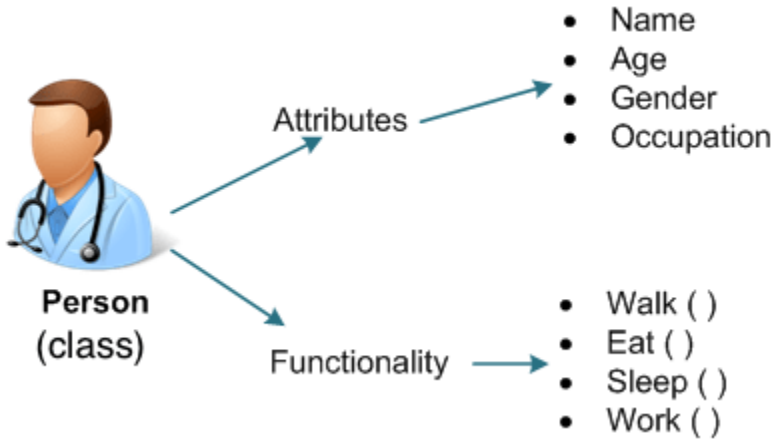
Bài tập 4: Tìm số lượng số nguyên tố từ 1 đến n. Với n nhập từ bàn phím

Bài tập 5: Tìm dãy số fibonancy thứ n, với n nhập từ bàn phím.

Bài 21: Lập trình hướng đối tượng

Lập trình hướng đối tượng (Object Oriented Programing) hay còn gọi là OOP. Là một kỹ thuật lập trình cho phép các lập trình viên có thể ánh xạ các thực thể bên ngoài đời thực và trừu tượng hoá thành các class và object trong mã nguồn.

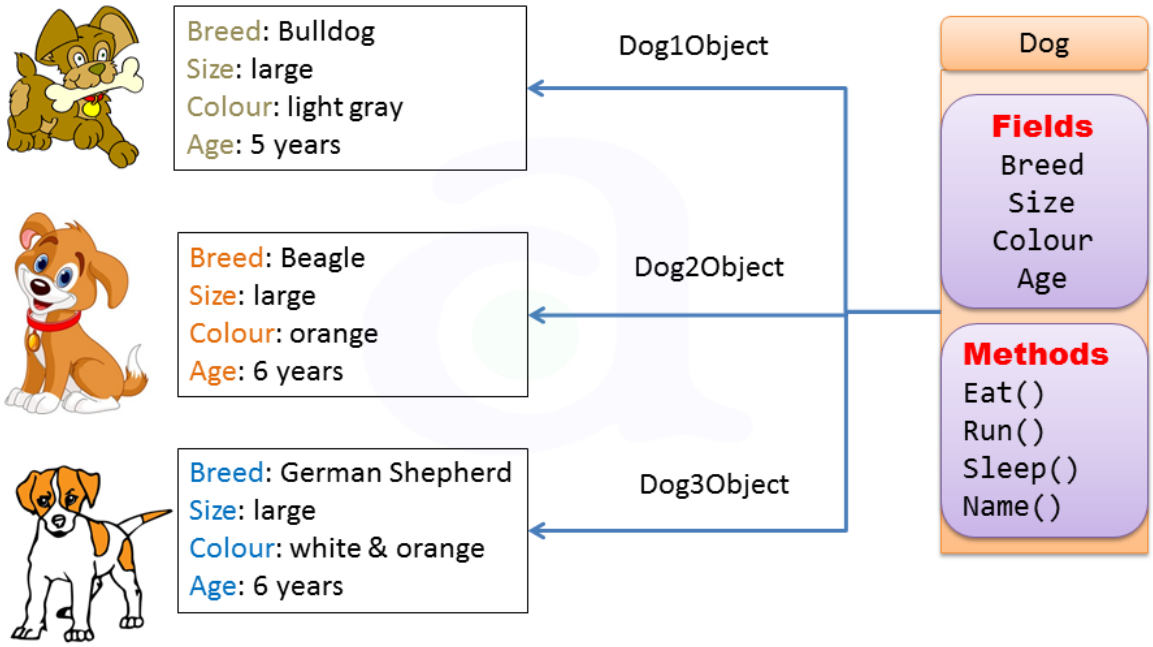
Trong đó: Mỗi thực thể được ánh xạ thành các class có chứa các thông tin mô tả của thực thể đó (gọi là thuộc tính) và các hành động của thực thể (gọi là các phương thức).



Giới thiệu về Class và Object

- Một Class là một Blueprint (kế hoạch) hay Prototype (nguyên mẫu) xác định biến và các phương thức (hay function) chung với tất cả các đối tượng cùng loại.
- Một Object (đối tượng) là một cụ thể, thể hiện của một Class.

Các đối tượng thường được dùng để mô tả đối tượng trong thế giới thực mà bạn thấy hàng ngày.



Cú pháp khai báo lớp:

```
<Access Modifiers> class Class_Name {
    // khai báo các thành viên dữ  liệu (thuộc tính, biến trường dữ liệu)
    // khai báo các thành viên  hàm (phương thức)
}
```

Sử dụng Constructor (1 constructor và nhiều constructor)

Phương thức khởi tạo (Constructor) là những phương thức đặc biệt được gọi đến ngay khi khởi tạo 1 đối tượng nào đó.

Đặc điểm

- Có tên trùng với tên lớp.
- Không có kiểu trả về.
- Được tự động gọi khi 1 đối tượng thuộc lớp được khởi tạo.
- Nếu như bạn không khai báo bất kỳ phương thức khởi tạo nào thì hệ thống sẽ tự tạo ra phương thức khởi tạo mặc định không đối số và không có nội dung gì.
- Có thể có nhiều constructor bên trong 1 lớp.

Phạm vi truy cập (Access Modifiers)

Phạm vi truy cập là cách mà người lập trình quy định về quyền được truy xuất đến các thành phần của lớp.

Trong C# có 5 phạm vi truy cập:

1. public: không giới hạn phạm vi truy cập
2. protected: chỉ truy cập trong nội bộ lớp hay các lớp kế thừa
3. private: chỉ truy cập được từ các thành viên của lớp chứa nó
4. internal: chỉ truy cập được trong cùng assembly (dll, exe)
5. protected internal: truy cập được khi cùng assembly hoặc lớp kế thừa

Lưu ý:

- Nếu khai báo lớp mà không chỉ ra phạm vi truy cập thì mặc định là **internal**
- Nếu khai báo thành phần bên trong lớp mà không chỉ ra phạm vi cụ thể thì mặc định là **private**

Tạo và sử dụng đối tượng

```
// Khai báo và khởi tạo đối tượng luôn
var ob1 = new ClassName();

// Khai báo, sau đó khởi tạo
ClassName ob2;
ob2 = new ClassName();
```

Sau khi đối tượng lớp (object) được tạo, bạn có thể truy cập đến các thuộc tính, trường dữ liệu và phương thức của đối tượng đó bằng ký hiệu . theo quy tắc **object.tên_thuộc_tính** hay **object.tên_phương_thức**

Từ khoá this

Từ khóa **this** dùng trong các phương thức của lớp, nó tham chiếu đến đối tượng hiện tại sinh ra từ lớp. Sử dụng **this** để tường minh, tránh sự không rõ ràng khi truy cập thuộc tính, phương thức hoặc để lấy đối tượng lớp làm tham số cho các thành phần khác ...

Thuộc tính (Properties)

Trường dữ liệu của lớp

Trường dữ liệu - khai báo như biến trong lớp, nó là thành viên của lớp, nó là biến. Trường dữ liệu có thể sử dụng bởi các phương thức trong lớp, hoặc nếu là public nó có thể truy cập từ bên ngoài, nhưng cách hay hơn để đảm bảo tính đóng gói khi cần truy cập thuộc tính hãy sử dụng phương thức, còn bản thân thuộc tính là private. Chúng ta đã sử dụng các trường dữ liệu ở những ví dụ trên.

Thuộc tính, bộ truy cập accessor setter/getter

Ngoài cách sử dụng trường dữ liệu, khai báo như biến ở phần trước, khai báo THUỘC TÍNH tương tự nhưng nó có cơ chế accessor (bộ truy cập), một cơ chế hết sức linh hoạt khi bạn đọc / ghi dữ liệu vào thuộc tính. Hãy tìm hiểu qua một ví dụ sau:

```
class Student
{
    private string name; // đây là trường dữ liệu
}
```

Lớp này có một trường dữ liệu private là **name**. Giờ ta sẽ khai báo một thuộc tính có tên **Name** với modify là **public**, thuộc tính này khi đọc sẽ thi hành một đoạn code gọi là **get**, khi ghi (gán) dữ liệu nó thi hành đoạn code gọi là **set**, thuộc tính **Name** sẽ phối hợp cùng trường dữ liệu **name**

```
class Student
{
    private string name;     // Đây là trường dữ liệu

    public string Name       // Đây là thuộc tính
    {
        // set thi hành khi gán, write
        // dữ liệu gán là value
        set
        {
            Console.WriteLine("Ghi dữ liệu <--" + value);
            name = value;
        }

        //get thi hành ghi đọc dữ liệu
        get {
            return "Tên là: " + name;
        }
    }
}
```

Khi thực hiện:

```
var s = new Student();
s.Name = "XYZ";                                          // set thi hành
// In ra: Ghi dữ liệu <--XYZ
// Và trường name giờ bằng XYZ

Console.WriteLine(s.Name);                               // get được thi hành
// In ra: Tên là: XYZ
```

Trong C#, phương thức truy xuất và phương thức cập nhật đã được nâng cấp lên thành 1 cấu trúc mới ngắn gọn hơn và tiện dụng hơn đó là **property**.

Sử dụng property giúp ta có thể thao tác dữ liệu tự nhiên hơn nhưng vẫn đảm bảo tính đóng gói của lập trình hướng đối tượng.

Lưu ý:

- Người ta dùng Property thay cho phương thức truy vấn, phương thức cập nhật vì thế tên property thường phải làm gợi nhớ đến tên thuộc tính private bên trong lớp.
- Tuỳ theo nhu cầu và tính bảo mật mà người lập trình có thể ngăn không cho gán giá trị hoặc ngăn không cho lấy dữ liệu bằng cách bỏ đi từ khoá tương ứng.
- Thuộc tính accessor có thể khai báo thiếu **set** hoặc **get**, nếu thiếu **set** nó trở thành loại chỉ đọc (readonly). Sử dụng **set** rất tiện lợi cho thao tác kiểm tra tính hợp lệ của dữ liệu khi gán, hoặc tự động thực hiện một số tác vụ mỗi khi dữ liệu được gán.

Tìm hiểu tính đóng gói lập trình hướng đối tượng

Tính đóng gói mục đích hạn chế tối đa việc can thiệp trực tiếp vào dữ liệu, hoặc thi hành các tác vụ nội bổ của đối tượng. Nói cách khác, một đối tượng là hộp đen đối với các thành phần bên ngoài, nó chỉ cho phép bên ngoài tương tác với nó ở một số phương thức, thuộc tính, trường dữ liệu nhất định - hạn chế.

C# triển khai tính đóng gói này chính là sử dụng các Access Modifiers: public private protected internal khi khai báo lớp, phương thức, thuộc tính, trường dữ liệu (biến).

- **public** thành viên có thể truy cập được bởi code bất kỳ đâu, ngoài đối tượng, không có hạn chế truy cập nào.
- **private** phương thức, thuộc tính, trường khai báo với private chỉ có thể truy cập, gọi bởi các dòng code cùng lớp.
- **protected** phương thức, thuộc tính, trường chỉ có thể truy cập, gọi bởi các dòng code cùng lớp hoặc các lớp kế thừa nó.
- **internal** truy cập được bởi code ở cùng assembly (file).
- **protected internal** truy cập được từ code assembly, hoặc lớp kế thừa nó ở assembly khác.

Bài 22: Sử dụng mảng (Arrays)

Giới thiệu về mảng

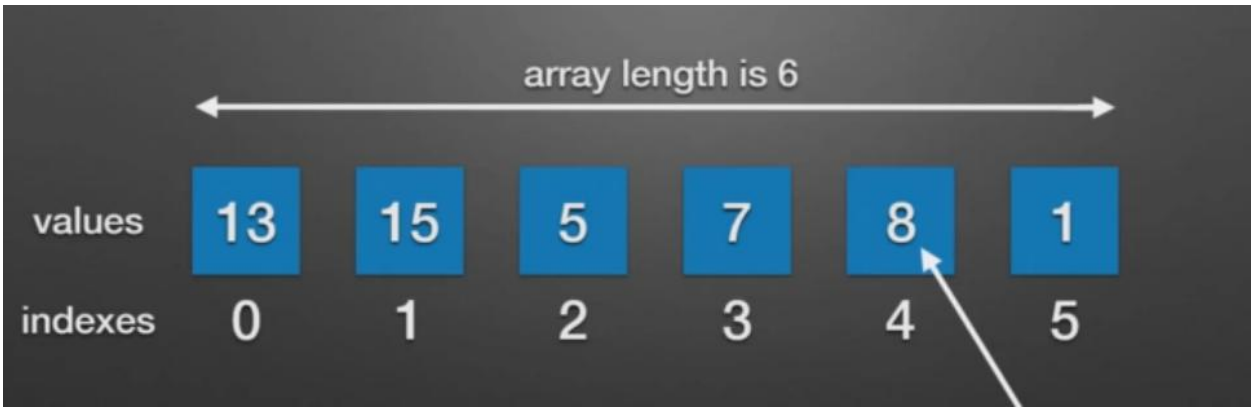
Mảng trong C# là một cấu trúc dữ liệu lưu trữ một dãy các phần tử có bộ nhớ nằm liên tiếp nhau và có kích thước cố đinh.

Mỗi phần tử trong mảng được truy cập thông qua chỉ số. Phần tử đầu tiên có chỉ số là 0 và phần tử cuối cùng có chỉ số n – 1 (trong đó n là số lượng phần tử có trong mảng).

Dựa vào cách mảng lưu trữ các phần tử, mảng có thể được phân thành 2 loại là mảng một chiều (Single-dimensional Arrays) và mảng đa chiều (Multi-dimensional Arrays).

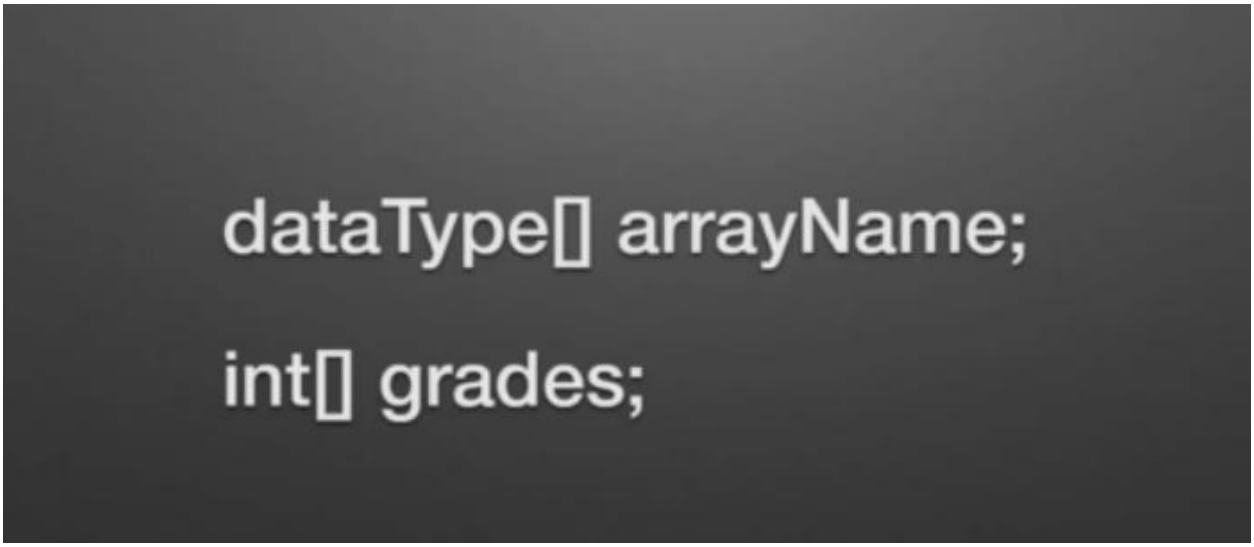
Tính chất:

- Độ dài cố định
- Chỉ lưu các phần tử cùng kiểu dữ liệu
- Lưu trữ và cấp phát các ô nhớ liên tiếp nhau

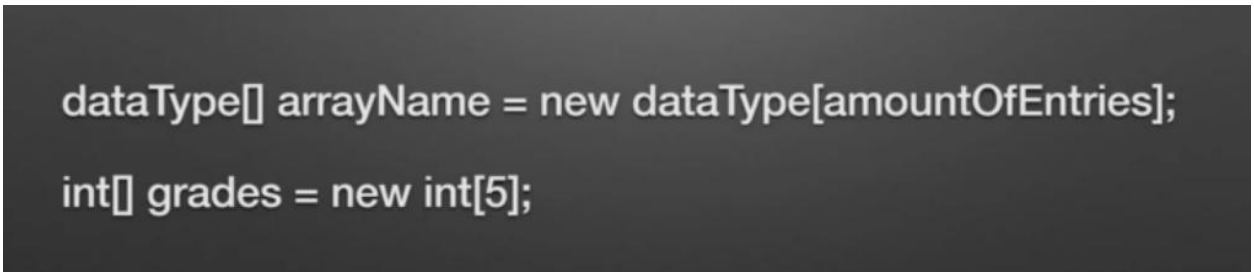


Khai báo và khởi tạo mảng

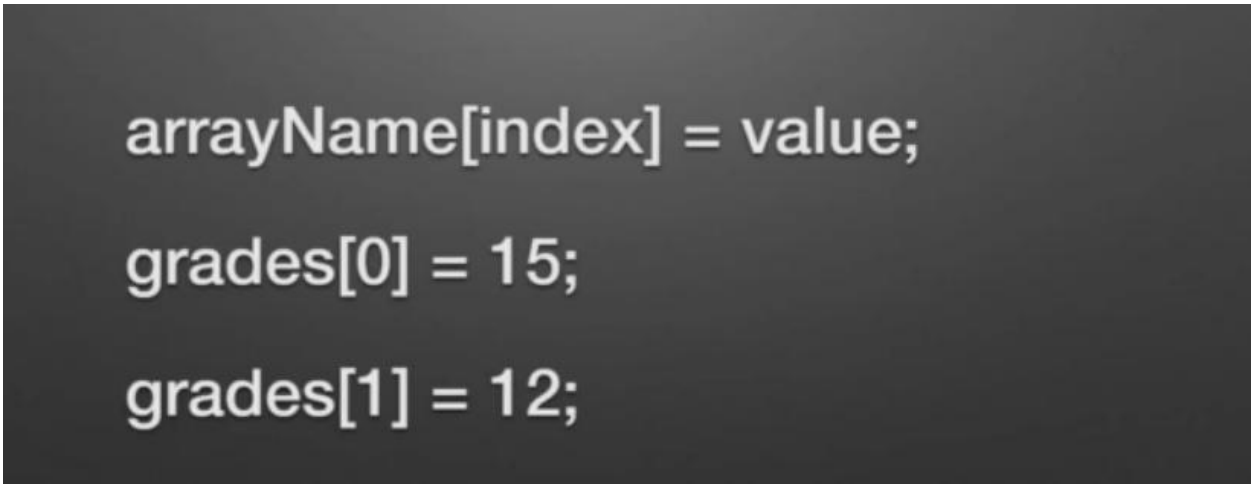
Khai báo



Khai báo và khởi tạo



Gán giá trị cho một mảng



Thực hành tạo mảng và truy xuất giá trị

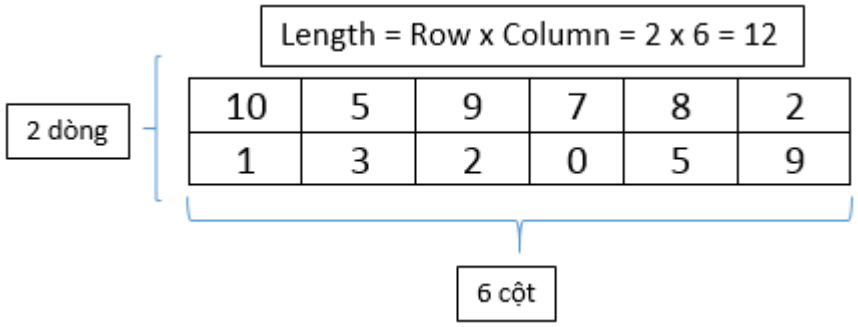
Lặp qua mảng

- Sử dụng vòng lặp for hoặc foreach

```
foreach(datatype variable in arrayName)
{
  // Xử lý
}
```

Mảng 2 chiều

Không giống như mảng một chiều, mảng đa chiều cho phép chúng ta lưu trữ dữ liệu trên nhiều dòng. Kích thước của mảng được xác định dựa vào số dòng và số cột tương tự như một sheet trong Microsoft Excel và được phân làm 2 loại như sau



Khai báo mảng 2 chiều:

```
datatype[,] arrayName = new datatype[rows , columns];
```

Jagged Array

Jagged Array tương tự Rectangular Array ngoại trừ số cột trên mỗi dòng có thể khác nhau. Hình bên dưới là một ví dụ

10	5	9	7	8	2
1	3	2	0	5	9
3	1	2			

Bài 23: Tổng quan về Generic và Non-Generic Collection

Mục tiêu bài học

- Hiểu khái niệm **Generic** và **Non-Generic Collection**.
- Phân biệt giữa Generic Collection và Non-Generic Collection.
- Làm quen với các Collection thường dùng trong C# như List<T>, Dictionary<TKey, TValue>, ArrayList, và Hashtable.
- Thực hành sử dụng Generic và Non-Generic Collection để giải quyết bài toán thực tế.

Phần 1: Tổng quan về Collection trong C#

1. **Collection là gì?**
    - Collection là các cấu trúc dữ liệu được sử dụng để lưu trữ và quản lý tập hợp các đối tượng.
    - Ưu điểm:
        - Quản lý dữ liệu linh hoạt hơn mảng.
        - Hỗ trợ nhiều phương thức và tính năng sẵn có.
2. **Phân loại Collection**
    - **Generic Collection**: Cho phép làm việc với các kiểu dữ liệu cụ thể. (e.g., List<T>, Dictionary<TKey, TValue>).
    - **Non-Generic Collection**: Làm việc với kiểu dữ liệu không cụ thể, thường là object. (e.g., ArrayList, Hashtable).

Phần 2: Generic Collection

1. **Tại sao sử dụng Generic Collection?**
    - Giúp tránh lỗi runtime do kiểu dữ liệu không khớp.
    - Cải thiện hiệu suất vì không cần ép kiểu (boxing/unboxing).
    - Tăng tính an toàn về kiểu dữ liệu tại compile-time.
2. **Một số Generic Collection thường dùng**
    - **List<T>**:
        - Dùng để lưu trữ danh sách các phần tử có thứ tự.
        - Hỗ trợ thao tác thêm, xóa, tìm kiếm.
        - Ví dụ

```
List<int> numbers = new List<int> { 1, 2, 3 };
numbers.Add(4);
Console.WriteLine(string.Join(", ", numbers));
```

    - **Dictionary<TKey, TValue>**:
        - Lưu trữ các cặp khóa-giá trị.
        - Dễ dàng truy cập giá trị thông qua khóa.
        - Ví dụ

```
Dictionary<string, int> studentScores = new Dictionary<string,
int>
{
    { "Alice", 90 },
    { "Bob", 85 }
};
Console.WriteLine(studentScores["Alice"]);
```

Phần 3: Non-Generic Collection

1. **Hạn chế của Non-Generic Collection**
    - Không đảm bảo an toàn về kiểu dữ liệu.
    - Hiệu suất thấp hơn do cần thực hiện boxing/unboxing.
2. **Một số Non-Generic Collection thường dùng**
    - **ArrayList**:
        - Lưu trữ danh sách các phần tử có thứ tự, nhưng không yêu cầu kiểu dữ liệu cụ thể.
        - Ví dụ

```
using System.Collections;

ArrayList list = new ArrayList { 1, "Hello", true };
list.Add(3.14);
Console.WriteLine(string.Join(", ", list.Cast<object>()));
```

- **Hashtable**:
  - Lưu trữ các cặp khóa-giá trị, khóa và giá trị đều là kiểu object.
  - Ví dụ

```
using System.Collections;

Hashtable table = new Hashtable
{
    { "Alice", 90 },
    { "Bob", 85 }
};
Console.WriteLine(table["Alice"]);
```

Phần 4: So sánh Generic và Non-Generic Collection

Đặc điểm	Generic Collection	Non-Generic Collection
Kiểu dữ liệu	Cụ thể (type-safe)	Không cụ thể (object)
Hiệu suất	Cao hơn	Thấp hơn
Tính an toàn kiểu	Đảm bảo	Không đảm bảo
Boxing/Unboxing	Không cần thiết	Cần boxing/unboxing

Phần 5: Bài tập thực hành

1. **Bài tập 1: Sử dụng List<T> để quản lý danh sách sinh viên.**
   - Tạo một danh sách lưu trữ tên các sinh viên.
   - Thêm, xóa, và tìm kiếm tên trong danh sách.
2. **Bài tập 2: Sử dụng Dictionary<TKey, TValue> để quản lý điểm thi.**
   - Lưu trữ điểm thi của sinh viên với khóa là tên và giá trị là điểm.
   - In danh sách sinh viên có điểm > 8.
3. **Bài tập 3: So sánh hiệu năng giữa ArrayList và List<T>.**
   - Tạo 1000 phần tử số nguyên trong ArrayList và List<int>.
   - Đo thời gian truy xuất từng phần tử.

Phần 6: Tổng kết

- Generic Collection mang lại hiệu suất và tính an toàn cao hơn so với Non-Generic Collection.
- Nên ưu tiên sử dụng Generic Collection khi làm việc với dữ liệu cụ thể.
- Non-Generic Collection chỉ nên dùng trong các trường hợp đặc biệt, khi không xác định được kiểu dữ liệu từ đầu.

Bài 24: Tìm hiểu về các loại Collection trong C#

Trong C#, **collections** là các cấu trúc dữ liệu được sử dụng để lưu trữ và quản lý các nhóm đối tượng. C# cung cấp nhiều loại collections khác nhau để phù hợp với các yêu cầu cụ thể của lập trình viên. Dưới đây là phân loại và giới thiệu chi tiết về các loại collections:

---

1. Non-Generic Collections

Được định nghĩa trong namespace System.Collections. Những collection này không yêu cầu kiểu dữ liệu cụ thể cho các phần tử mà chúng lưu trữ.

Các loại phổ biến:

- **ArrayList**: Lưu trữ các phần tử không có kiểu cụ thể. Kích thước có thể thay đổi linh hoạt.
    - Ưu điểm: Dễ sử dụng khi không biết kiểu dữ liệu chính xác.
    - Nhược điểm: Hiệu suất kém hơn so với collections generic và không an toàn kiểu dữ liệu.
- **Hashtable**: Lưu trữ các cặp **key-value**. Các khóa không thể trùng lặp.
- **Queue**: Thực hiện theo cơ chế **FIFO (First In First Out)**.
- **Stack**: Thực hiện theo cơ chế **LIFO (Last In First Out)**.

---

2. Generic Collections

Được định nghĩa trong namespace System.Collections.Generic. Các collection này yêu cầu xác định kiểu dữ liệu ngay khi khai báo, giúp an toàn kiểu và hiệu suất cao hơn.

Các loại phổ biến:

- **List<T>**:
    - Lưu trữ danh sách các phần tử có kiểu xác định.
    - Tương tự như ArrayList nhưng an toàn kiểu.

```
List<int> numbers = new List<int> { 1, 2, 3 };
numbers.Add(4); // Thêm phần tử
```

- **Dictionary<TKey, TValue>**:
  - Lưu trữ các cặp **key-value** với kiểu xác định.
  - Các key là duy nhất.

```
Dictionary<int, string> students = new Dictionary<int, string>();
students.Add(1, "Alice");
```

- **Queue<T>**:
  - Queue generic với kiểu dữ liệu xác định.
- **Stack<T>**:
  - Stack generic với kiểu dữ liệu xác định.
- **HashSet<T>**:
  - Một tập hợp không chứa các phần tử trùng lặp.

---

3. Concurrent Collections

Được định nghĩa trong namespace System.Collections.Concurrent. Những collection này được thiết kế để hoạt động tốt trong môi trường đa luồng (**multi-threading**).

Các loại phổ biến:

- **ConcurrentBag<T>**: Hỗ trợ thêm/xóa các phần tử không có thứ tự cụ thể.
- **ConcurrentDictionary<TKey, TValue>**: Dictionary hỗ trợ thread-safe.
- **BlockingCollection<T>**: Được sử dụng để điều phối sản xuất và tiêu thụ trong các ứng dụng đa luồng.

---

4. Specialized Collections

Được định nghĩa trong namespace System.Collections.Specialized. Được sử dụng trong các trường hợp đặc biệt.

Các loại phổ biến:

- **StringCollection**: Một collection chỉ lưu trữ các chuỗi.
- **NameValueCollection**: Lưu trữ các cặp **key-value** nhưng hỗ trợ nhiều giá trị cho một key.

---

5. Immutable Collections

Được định nghĩa trong namespace System.Collections.Immutable. Các collection này không thể thay đổi sau khi được tạo ra, rất hữu ích trong các ứng dụng không cho phép chỉnh sửa dữ liệu (immutable).

Các loại phổ biến:

- **ImmutableList<T>**
- **ImmutableDictionary<TKey, TValue>**
- **ImmutableArray<T>**

So sánh Non-Generic và Generic

Tiêu chí	**Non-Generic**	**Generic**
An toàn kiểu	Không an toàn	An toàn
Hiệu suất	Thấp hơn	Cao hơn
Kiểu dữ liệu	Không yêu cầu cụ thể	Phải xác định
Tính phổ biến	Ít phổ biến hơn	Phổ biến hơn

Khi nào nên dùng loại nào?

- **Non-Generic**: Khi cần làm việc với kiểu dữ liệu hỗn hợp (hiếm gặp).
- **Generic**: Nên ưu tiên sử dụng vì an toàn và hiệu suất cao.
- **Concurrent**: Khi làm việc trong môi trường đa luồng.
- **Immutable**: Khi cần đảm bảo tính bất biến của dữ liệu.

Bài tập 1:

Đề bài: Viết chương trình sử dụng Queue<T> để quản lý hàng đợi khách hàng tại ngân hàng.
Các chức năng:

- Thêm khách hàng mới vào hàng đợi.
- Gọi khách hàng tiếp theo.
- Hiển thị danh sách khách hàng trong hàng đợi.

Bài tập 2: Đảo ngược chuỗi bằng Stack

Đề bài: Viết chương trình sử dụng Stack để đảo ngược một chuỗi bất kỳ do người dùng nhập vào.
Ví dụ:

- Nhập: "Hello"

- Kết quả: "olleH"

Tham khảo: https://tedu.com.vn/lap-trinh-c/cnet-can-ban-hieu-biet-ve-cac-collection-trong-net-framework-90.html

Bài 25: Cách debug ứng dụng C#

1. Chuẩn bị môi trường

- Sử dụng **Visual Studio** hoặc **Visual Studio Code** với các extension như **C# for Visual Studio Code**.
- Đảm bảo cài đặt .NET SDK phù hợp với dự án.
- Cài đặt **C# Dev Kit** Extensions cho Visual Studio Code

---

2. Chạy ứng dụng trong chế độ Debug

- **Visual Studio**: Nhấn F5 hoặc chọn **Debug > Start Debugging**.
- **Visual Studio Code**:
  - Cấu hình file launch.json (thường được tạo sẵn khi bạn chạy lần đầu).
  - Nhấn F5 để khởi động chế độ debug.

---

3. Đặt Breakpoints

Breakpoints cho phép dừng chương trình tại một dòng mã cụ thể:

- Nhấn vào lề bên trái của dòng mã trong trình soạn thảo (hoặc nhấn F9 trong Visual Studio).
- Khi chương trình chạy tới dòng đó, nó sẽ tạm dừng.

---

4. Kiểm tra giá trị biến

Khi chương trình dừng tại breakpoint:

- Di chuột qua biến để xem giá trị.
- Sử dụng **Watch Window** để thêm biến cần theo dõi.
- Dùng **Immediate Window** để thực thi lệnh và kiểm tra giá trị.

---

5. Sử dụng công cụ Step

Các lệnh step giúp bạn kiểm soát luồng chạy:

- **Step Over (F10)**: Chạy qua một dòng mã (không vào phương thức con).

- **Step Into (F11)**: Đi vào phương thức con để xem chi tiết bên trong.
- **Step Out (Shift + F11)**: Thoát khỏi phương thức hiện tại.

Bài 26: Tính chất kế thừa (Inheritance) và đa hình (polymorphism)

I. Tính Chất Kế Thừa (Inheritance)

1. **Khái Niệm:**
   - Kế thừa là cơ chế cho phép một lớp (class) kế thừa các thuộc tính và phương thức từ một lớp khác.
   - Lớp kế thừa gọi là **lớp con (child class)**, lớp được kế thừa gọi là **lớp cha (parent class)**.
2. **Mục Đích:**
   - Tái sử dụng mã nguồn (code reuse).
   - Tổ chức và phân cấp hệ thống, giúp dễ quản lý và mở rộng.
3. **Cách Sử Dụng:**

```
class ParentClass
{
    public string Name { get; set; }
    public void Display()
    {
        Console.WriteLine($"Name: {Name}");
    }
}

class ChildClass : ParentClass
{
    public void ShowMessage()
    {
        Console.WriteLine("This is a child class.");
    }
}

ChildClass child = new ChildClass();
child.Name = "John";
child.Display(); // Từ lớp cha
child.ShowMessage(); // Từ lớp con
```

4. **Ưu Điểm:**
   - Tăng khả năng tái sử dụng mã.
   - Giảm sự trùng lặp mã.
5. **Lưu Ý:**

- Kế thừa đơn trong C#: Một lớp chỉ có thể kế thừa từ **một lớp cha**.
- Sử dụng từ khóa **base** để truy cập các thành phần của lớp cha.

II. Tính Chất Đa Hình (Polymorphism)

1. **Khái Niệm:**
   - Đa hình là khả năng thực thi một hành vi (phương thức) theo nhiều cách khác nhau.
   - Có hai loại:
     - **Đa hình tại thời điểm biên dịch (Compile-time polymorphism):** Overloading (nạp chồng phương thức).
     - **Đa hình tại thời điểm chạy (Runtime polymorphism):** Overriding (ghi đè phương thức).
2. **Nạp Chồng Phương Thức (Method Overloading):**
   - Cùng tên phương thức nhưng khác tham số (số lượng hoặc kiểu dữ liệu).
   - **Ví dụ**

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

    public double Add(double a, double b)
    {
        return a + b;
    }
}

// Sử dụng
Calculator calc = new Calculator();
Console.WriteLine(calc.Add(3, 5)); // 8
Console.WriteLine(calc.Add(2.5, 4.5)); // 7.0
```

3. **Ghi Đè Phương Thức (Method Overriding):**
   - Cho phép lớp con thay đổi cách triển khai phương thức của lớp cha.
   - Sử dụng từ khóa virtual ở lớp cha và override ở lớp con.

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

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

// Sử dụng
Animal myDog = new Dog();
myDog.Speak(); // The dog barks.
```

4. **Ưu Điểm:**
    - Linh hoạt trong thiết kế phần mềm.
    - Tăng khả năng mở rộng (open for extension).
5. **Lưu Ý:**
    - Phải có từ khóa virtual và override để ghi đè phương thức.
    - Đa hình runtime chỉ áp dụng khi sử dụng tham chiếu kiểu lớp cha trỏ đến đối tượng lớp con.

III. So Sánh Kế Thừa và Đa Hình

Đặc Điểm	**Kế Thừa**	**Đa Hình**
Khái Niệm	Chia sẻ và tái sử dụng mã từ lớp cha.	Hành vi khác nhau với cùng một tên.
Mục Tiêu	Tái sử dụng và tổ chức mã.	Linh hoạt và mở rộng chức năng.
Loại	Tương tác giữa lớp cha và lớp con.	Compile-time hoặc Runtime.
Từ Khóa	: để chỉ định kế thừa.	virtual, override, new.

IV. Bài Tập Thực Hành

1. Tạo lớp cha Vehicle với phương thức Move(). Tạo các lớp con như Car, Bike kế thừa từ Vehicle và ghi đè phương thức Move().
2. Bài tập quản lý sinh viên:

    - Tạo một hệ thống quản lý nhân viên với các loại nhân viên khác nhau: FullTimeEmployee, PartTimeEmployee, và Intern.

    - Lớp cha Employee chứa thông tin chung như Name và CalculateSalary().

    - Các lớp con sẽ ghi đè phương thức CalculateSalary() để tính lương theo quy tắc riêng:

- **Nhân viên toàn thời gian (FullTimeEmployee):** Lương cố định.
- **Nhân viên bán thời gian (PartTimeEmployee):** Tính lương theo giờ làm việc.
- **Thực tập sinh (Intern):** Không nhận lương, nhưng in thông báo.

- Sử dụng đa hình để gọi phương thức CalculateSalary() cho các loại nhân viên khác nhau.

V. Kết Luận

- **Kế thừa** giúp tổ chức mã nguồn rõ ràng, dễ bảo trì.
- **Đa hình** mang lại sự linh hoạt khi xử lý hành vi cho các đối tượng khác nhau.
- Hai tính chất này kết hợp tạo nên sức mạnh cho lập trình hướng đối tượng, giúp xây dựng các hệ thống phức tạp một cách dễ dàng.

Bài 27: Abstract classes and interfaces

1. Tính trừu tượng là gì?

Tính trừu tượng (Abstraction) trong OOP là kỹ thuật **ẩn đi các chi tiết triển khai** và chỉ hiển thị cho người dùng những **chức năng cần thiết**. Điều này giúp bạn tập trung vào **"cái gì"** một đối tượng có thể làm, thay vì **"làm như thế nào"**.

Ví dụ: Khi bạn lái xe, bạn chỉ cần biết cách **điều khiển vô lăng**, **đạp ga/phanh**, mà không cần biết chi tiết bên trong động cơ hoạt động như thế nào.

---

2. Abstract Classes (Lớp trừu tượng)

Định nghĩa:

- **Abstract class** là một lớp không thể khởi tạo trực tiếp, chỉ có thể được kế thừa bởi lớp con.
- Một lớp trừu tượng có thể chứa cả:
  - **Phương thức trừu tượng** (không có thân hàm, chỉ định nghĩa).
  - **Phương thức bình thường** (có thân hàm).

Cách sử dụng:

- Khi bạn muốn cung cấp **cơ chế mặc định** cho các lớp con, nhưng cũng yêu cầu các lớp con phải triển khai một số phương thức cụ thể.

```
public abstract class Animal
{
    // Phương thức trừu tượng
    public abstract void Speak(); // Lớp con bắt buộc phải override.
```

```
    // Phương thức thông thường
    public void Eat()
    {
        Console.WriteLine("Animal is eating...");
    }
}

// Lớp kế thừa Abstract Class
public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks: Woof Woof!");
    }
}
```

Đặc điểm:

- **Không thể tạo đối tượng từ abstract class.**
- **Có thể chứa thuộc tính, phương thức tĩnh.**

---

3. Interfaces (Giao diện)

Định nghĩa:

- **Interface** là một tập hợp các phương thức, thuộc tính mà các lớp thực thi phải triển khai.
- Chỉ định nghĩa **cái gì cần làm** (không cung cấp code triển khai).
- Một lớp có thể thực thi **nhiều interfaces** (đa kế thừa).

```
public interface IAnimal
{
    void Speak(); // Phương thức không có phần thân.
    void Move();
}

public class Bird : IAnimal
{
    public void Speak()
    {
        Console.WriteLine("Bird chirps: Tweet Tweet!");
    }

    public void Move()
    {
        Console.WriteLine("Bird flies!");
    }
}
```

Đặc điểm:

- **Tất cả các phương thức mặc định là public.**
- Không chứa code triển khai (trừ khi dùng default implementation - từ C# 8.0 trở đi).

- **Hỗ trợ đa kế thừa** (một lớp có thể implement nhiều interfaces).

---

4. So sánh Abstract Classes và Interfaces

Tiêu chí	Abstract Class	Interface
Kế thừa	Một lớp chỉ kế thừa **một abstract class**.	Một lớp có thể thực thi **nhiều interfaces**.
Phương thức	Có thể chứa cả phương thức có thân hàm và trừu tượng.	Chỉ chứa phương thức không có thân hàm (trừ default implementation từ C# 8.0).
Thuộc tính/Tiêu đề	Hỗ trợ thuộc tính, hằng số, phương thức tĩnh.	Không hỗ trợ thuộc tính hoặc phương thức tĩnh (trừ C# 8.0).
Mục đích sử dụng	Khi các lớp con cần chia sẻ code chung.	Khi muốn xác định hành vi mà các lớp phải tuân theo.

---

5. Khi nào dùng Abstract Class và Interface?

- **Abstract Class**:
  - Khi các lớp con cần kế thừa **một logic chung**.
  - Khi bạn cần một **base class** với cả phương thức trừu tượng và không trừu tượng.
  - Ví dụ: Hệ thống động vật với logic chung như Eat().
- **Interface**:
  - Khi các lớp không liên quan cần chia sẻ **hành vi**.
  - Khi bạn cần **đa kế thừa**.
  - Ví dụ: Các đối tượng Bird, Fish, và Plane đều có hành vi Move() nhưng không liên quan logic.

---

6. Ví dụ kết hợp Abstract Class và Interface

```
public abstract class Animal
{
    public abstract void Speak();
    public void Eat()
    {
        Console.WriteLine("Animal is eating...");
    }
}
```

```
public interface ISwimmable
{
    void Swim();
}

public class Dolphin : Animal, ISwimmable
{
    public override void Speak()
    {
        Console.WriteLine("Dolphin clicks: Click Click!");
    }

    public void Swim()
    {
        Console.WriteLine("Dolphin swims in the ocean!");
    }
}
```

Kết quả khi chạy:

```
Dolphin dolphin = new Dolphin();
dolphin.Speak(); // Dolphin clicks: Click Click!
dolphin.Eat();   // Animal is eating...
dolphin.Swim();  // Dolphin swims in the ocean!
```

---

7. Bài tập thực hành

Mô tả:
Kết hợp cả Abstract Class và Interface để quản lý một hệ thống.

1. Tạo abstract class Device với:
    - Phương thức GetDeviceInfo(): Hiển thị thông tin thiết bị.
2. Tạo interface INetworkConnectable với:
    - Phương thức ConnectToNetwork().
3. Tạo các lớp kế thừa Device và thực thi INetworkConnectable:
    - SmartPhone.
    - Laptop.
4. Viết chương trình tạo danh sách thiết bị và thực hiện kết nối mạng.

Bài 28: Tính đóng gói (Encapsulation) và best practices trong OOP

1. Giới thiệu về Encapsulation

- **Encapsulation** (Tính đóng gói) là một trong những nguyên tắc cơ bản của lập trình hướng đối tượng (OOP).
- Nó được sử dụng để ẩn đi chi tiết triển khai nội bộ của một lớp, chỉ cho phép truy cập và sửa đổi thông qua các phương thức hoặc thuộc tính công khai (public).
- **Lợi ích chính**:
    - **Bảo mật dữ liệu**: Ngăn chặn truy cập trái phép vào các thành phần bên trong.
    - **Tăng tính linh hoạt**: Dễ dàng thay đổi logic bên trong mà không ảnh hưởng đến mã bên ngoài.
    - **Dễ bảo trì**: Cải thiện khả năng quản lý mã nguồn.

---

2. Các thành phần của Encapsulation

- **Access Modifiers** (Các phạm vi truy cập):
    - private: Chỉ có thể truy cập từ bên trong lớp.
    - protected: Truy cập từ bên trong lớp và các lớp dẫn xuất.
    - public: Truy cập từ bất kỳ đâu.
    - internal (trong C#) / default (trong Java): Chỉ truy cập được trong cùng một package/assembly.
- **Getters và Setters**:
    - Được sử dụng để truy cập và cập nhật giá trị của các trường (fields) được ẩn.

---

3. Ví dụ về Encapsulation

```
public class BankAccount
{
    private decimal balance; // Biến private để bảo vệ dữ liệu

    public decimal Balance
    {
        get { return balance; } // Chỉ cho phép đọc giá trị
        private set
        {
            if (value >= 0)
                balance = value; // Chỉ cho phép cập nhật giá trị hợp lệ
        }
    }

    public BankAccount(decimal initialBalance)
    {
        Balance = initialBalance;
    }
```

```
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            Balance += amount;
        }
    }

    public bool Withdraw(decimal amount)
    {
        if (amount > 0 && Balance >= amount)
        {
            Balance -= amount;
            return true;
        }
        return false;
    }
}
```

4. Best Practices (Thực hành tốt nhất)

1. **Giữ trạng thái (fields) ở chế độ private hoặc protected**:
    - Tránh để trạng thái (fields) ở chế độ public vì nó làm lộ dữ liệu và giảm tính bảo mật.
    - Sử dụng các phương thức get và set để kiểm soát quyền truy cập.
2. **Chỉ mở ra những gì cần thiết**:
    - Các phương thức và thuộc tính nên chỉ public nếu thực sự cần thiết.
3. **Sử dụng các phương thức hợp lệ để kiểm soát dữ liệu**:
    - Ví dụ: kiểm tra giá trị trước khi gán, tránh trạng thái không hợp lệ.
4. **Tách biệt giữa logic bên trong và giao diện bên ngoài**:
    - Tạo ra một API rõ ràng để giao tiếp với lớp, ẩn đi các chi tiết nội bộ.
5. **Tuân thủ nguyên tắc SRP (Single Responsibility Principle)**:
    - Mỗi lớp nên chỉ chịu trách nhiệm duy nhất về một phần logic trong ứng dụng.
6. **Sử dụng tính bất biến (immutability) khi cần thiết**:
    - Tạo các đối tượng không thay đổi trạng thái sau khi khởi tạo.

---

5. Tích hợp Encapsulation trong Thiết Kế OOP

- Kết hợp Encapsulation với các nguyên tắc SOLID khác (ví dụ: Dependency Injection, Open-Closed Principle) để thiết kế hệ thống linh hoạt và dễ mở rộng.
- Trong các hệ thống lớn, sử dụng Encapsulation để giảm sự phụ thuộc giữa các module.

6. Tóm tắt

- **Encapsulation** giúp bảo vệ dữ liệu, giảm độ phức tạp của hệ thống, và cải thiện khả năng bảo trì.
- Thực hành tốt bao gồm ẩn các chi tiết nội bộ, sử dụng các phương thức getter và setter hợp lý, và thiết kế API rõ ràng.
- Làm chủ Encapsulation là bước đầu tiên để trở thành một nhà phát triển giỏi về thiết kế hướng đối tượng.

Bài 29: Sử dụng kiểu tập hợp (Enum)

1. Enum là gì?

Enum (viết tắt của *Enumeration*) trong C# là một kiểu dữ liệu đặc biệt cho phép bạn định nghĩa một tập hợp các hằng số có tên. Enum thường được sử dụng khi bạn có một tập hợp các giá trị cố định và muốn làm cho mã dễ đọc hơn.

Ví dụ, bạn muốn biểu diễn các ngày trong tuần: Monday, Tuesday, ..., Sunday.

2. Cách khai báo Enum

```
enum TenEnum
{
    GiaTri1,
    GiaTri2,
    ...
}
```

- Ví dụ

```
enum DayOfWeek
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

3. Giá trị mặc định của Enum

- Các giá trị trong enum mặc định bắt đầu từ 0 và tăng dần.
- Bạn có thể gán giá trị cụ thể cho từng phần tử:

```
enum DayOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
```

```
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}
```

4. Sử dụng Enum trong C#

- **Khai báo biến kiểu Enum:**

```
DayOfWeek today = DayOfWeek.Friday;
Console.WriteLine(today); // Output: Friday
```

- **Ép kiểu Enum sang số nguyên:**

```
int dayValue = (int)DayOfWeek.Friday;
Console.WriteLine(dayValue); // Output: 5
```

- **Ép kiểu số nguyên sang Enum:**

```
DayOfWeek day = (DayOfWeek)3;
Console.WriteLine(day); // Output: Wednesday
```

5. Enum và vòng lặp

Bạn có thể duyệt qua các giá trị của enum bằng cách sử dụng phương thức Enum.GetValues():

```
foreach (DayOfWeek day in Enum.GetValues(typeof(DayOfWeek)))
{
    Console.WriteLine(day);
}
```

6. Thêm mô tả cho Enum

Trong C#, bạn có thể thêm mô tả cho từng giá trị của Enum bằng cách sử dụng thuộc tính Description trong thư viện System.ComponentModel.

- Bạn có thể viết một phương thức tiện ích để lấy mô tả của một giá trị Enum.
- Bạn cũng có thể lấy toàn bộ danh sách các giá trị và mô tả của một Enum:

7. Lợi ích của Enum

- Giúp mã dễ đọc và bảo trì hơn.
- Tránh sử dụng các giá trị "magic numbers".
- Dễ dàng xử lý các nhóm giá trị cố định.

8. Thực hành: Viết chương trình đơn giản

Đề bài: Viết chương trình nhận vào một số nguyên từ 1 đến 7 và in ra ngày tương ứng trong tuần.

Gợi ý:

```
using System;
```

```
enum DayOfWeek
{
    Monday = 1,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

class Program
{
    static void Main()
    {
        Console.Write("Nhập số từ 1 đến 7: ");
        int dayNumber = int.Parse(Console.ReadLine());

        if (Enum.IsDefined(typeof(DayOfWeek), dayNumber))
        {
            DayOfWeek day = (DayOfWeek)dayNumber;
            Console.WriteLine($"Ngày tương ứng: {day}");
        }
        else
        {
            Console.WriteLine("Số nhập không hợp lệ!");
        }
    }
}
```

9. Kết luận

`Enum` là một công cụ hữu ích trong C# giúp bạn quản lý các giá trị cố định một cách hiệu quả và rõ ràng. Việc sử dụng `enum` không chỉ làm cho mã nguồn dễ hiểu mà còn giúp giảm thiểu lỗi do việc sử dụng sai giá trị.

Bài 30: Sử dụng Math class

Trong C#, **Math class** là một lớp tiện ích chứa các phương thức và hằng số dùng để thực hiện các phép toán học. Lớp này thuộc namespace System và không cần khởi tạo đối tượng vì tất cả các thành viên đều là **static**.

1. Các phương thức và hằng số phổ biến của Math Class

1. **Hằng số toán học**:
    - Math.PI: Trả về giá trị của số pi (≈ 3.14159).
    - Math.E: Trả về cơ số của lôgarit tự nhiên (≈ 2.71828).
2. **Phương thức toán học cơ bản**:

- Math.Abs(x): Trả về giá trị tuyệt đối của x.
- Math.Ceiling(x): Làm tròn lên đến số nguyên gần nhất.
- Math.Floor(x): Làm tròn xuống đến số nguyên gần nhất.
- Math.Round(x): Làm tròn số đến số nguyên gần nhất hoặc số thập phân được chỉ định.

3. **Lũy thừa và căn bậc hai**:
   - Math.Pow(x, y): Trả về x mũ y.
   - Math.Sqrt(x): Trả về căn bậc hai của x.
4. **Phương thức logarit và mũ**:
   - Math.Log(x): Trả về lôgarit tự nhiên (cơ số e) của x.
   - Math.Log10(x): Trả về lôgarit cơ số 10 của x.
   - Math.Exp(x): Trả về e^x.
5. **Lượng giác**:
   - Math.Sin(x): Trả về sin của x (đơn vị radian).
   - Math.Cos(x): Trả về cos của x (đơn vị radian).
   - Math.Tan(x): Trả về tan của x (đơn vị radian).
6. **Số ngẫu nhiên**:
   - Sử dụng Random class thay vì Math để tạo số ngẫu nhiên.

7 . Bài tập thực hành

1. Viết chương trình tính diện tích hình tròn với bán kính nhập từ bàn phím.
2. Tạo chương trình nhập một số nguyên từ bàn phím và in ra giá trị tuyệt đối, bình phương, và căn bậc hai của số đó.
3. Viết hàm kiểm tra xem một góc (nhập bằng độ) có nằm trong góc phần tư thứ nhất hay không (sử dụng sin và cos).

Bài 31: Sử dụng DateTime

1. DateTime là gì?

DateTime là một kiểu dữ liệu trong C# được sử dụng để biểu diễn ngày, giờ và các thao tác liên quan.

Nó nằm trong **System namespace** và rất hữu ích khi làm việc với ngày tháng.

2. Cách tạo một đối tượng DateTime

Có nhiều cách để khởi tạo một đối tượng DateTime:

Lấy ngày giờ hiện tại:

```
DateTime now = DateTime.Now;
Console.WriteLine("Ngày giờ hiện tại: " + now);
```

Lấy giờ UTC:

```
DateTime utcNow = DateTime.UtcNow;
Console.WriteLine("UTC hiện tại: " + utcNow);
```

Khởi tạo một ngày giờ cụ thể:

```
DateTime specificDate = new DateTime(2024, 11, 26, 14, 30, 0);
Console.WriteLine("Ngày giờ cụ thể: " + specificDate);
```

Chỉ lấy ngày hôm nay:

```
DateTime today = DateTime.Today;
Console.WriteLine("Ngày hôm nay: " + today.ToShortDateString());
```

3. Các định dạng ngày giờ

- **Chuyển đổi thành chuỗi**:

```
DateTime now = DateTime.Now;
Console.WriteLine("Định dạng mặc định: " + now.ToString());
Console.WriteLine("Ngày: " + now.ToShortDateString());
Console.WriteLine("Giờ: " + now.ToShortTimeString());
Console.WriteLine("Tùy chỉnh: " + now.ToString("dd/MM/yyyy HH:mm:ss"));
```

Phân tích chuỗi thành DateTime:

```
string dateString = "26/11/2024 14:30";
DateTime parsedDate = DateTime.ParseExact(dateString, "dd/MM/yyyy HH:mm", null);
Console.WriteLine("Chuỗi đã phân tích: " + parsedDate);
```

4. Các phương thức và thuộc tính hữu ích

- **Thêm hoặc bớt thời gian**

```
DateTime now = DateTime.Now;
DateTime tomorrow = now.AddDays(1);
DateTime lastWeek = now.AddDays(-7);
Console.WriteLine("Ngày mai: " + tomorrow);
Console.WriteLine("Tuần trước: " + lastWeek);
```

Lấy các thành phần ngày giờ:

```
DateTime now = DateTime.Now;
Console.WriteLine("Năm: " + now.Year);
Console.WriteLine("Tháng: " + now.Month);
```

```
Console.WriteLine("Ngày: " + now.Day);
Console.WriteLine("Giờ: " + now.Hour);
Console.WriteLine("Phút: " + now.Minute);
Console.WriteLine("Giây: " + now.Second);
```

Kiểm tra ngày hợp lệ:

```
bool isValidDate = DateTime.TryParse("30/02/2024", out DateTime result);
Console.WriteLine(isValidDate ? "Hợp lệ" : "Không hợp lệ");
```

Dưới đây là bài giảng về cách sử dụng **DateTime** trong C#, phù hợp cho người mới bắt đầu:

---

5. Sử dụng TimeSpan

TimeSpan được sử dụng để biểu diễn khoảng cách giữa hai thời điểm:

```
DateTime start = new DateTime(2024, 11, 1);
DateTime end = new DateTime(2024, 11, 26);
TimeSpan duration = end - start;
Console.WriteLine("Số ngày giữa hai ngày: " + duration.Days);
```

6. Ứng dụng thực tế

- **Tạo bộ đếm thời gian**:

  ```
  DateTime startTime = DateTime.Now;
  // Một tác vụ mất thời gian
  System.Threading.Thread.Sleep(2000);
  DateTime endTime = DateTime.Now;
  TimeSpan elapsedTime = endTime - startTime;
  Console.WriteLine("Thời gian thực hiện: " + elapsedTime.TotalSeconds + "
  giây");
  ```

- **Hiển thị ngày giờ theo múi giờ khác**:

```
DateTime utcNow = DateTime.UtcNow;
TimeZoneInfo vietnamZone = TimeZoneInfo.FindSystemTimeZoneById("SE Asia Standard
Time");
DateTime vietnamTime = TimeZoneInfo.ConvertTimeFromUtc(utcNow, vietnamZone);
Console.WriteLine("Giờ Việt Nam: " + vietnamTime);
```

7. Bài tập thực hành

1. Viết chương trình nhập ngày sinh của bạn và tính số ngày bạn đã sống.
2. Tạo ứng dụng hiển thị đồng hồ thời gian thực.
3. Viết chương trình đếm ngược đến ngày cuối năm.

Bài 32: Kiểu Nullable

Nullable trong C#

1. Nullable là gì?

- Nullable cho phép một biến kiểu giá trị (value type) có thể nhận giá trị null.
- Ví dụ, thông thường kiểu int không thể nhận giá trị null, nhưng với Nullable<int>, điều này trở nên khả thi.

---

2. Cú pháp

Có hai cách để khai báo một biến nullable:

1. Sử dụng Nullable<T>:

   Nullable<int> x = null;

2. Sử dụng dấu ? (cách viết tắt):

   int? x = null;

---

3. Cách sử dụng

- **Gán giá trị null:**

  ```
  int? age = null;
  ```

- **Kiểm tra giá trị null:**

  ```
  if (age.HasValue)
  {
      Console.WriteLine($"Tuổi là: {age.Value}");
  }
  else
  {
      Console.WriteLine("Tuổi chưa được gán giá trị.");
  }
  ```

- **Sử dụng giá trị mặc định nếu là null:**

  ```
  int defaultAge = age ?? 18; // Nếu age là null, thì defaultAge = 18
  ```

---

4. Null-Coalescing Operators

C# cung cấp các toán tử để xử lý nullable một cách ngắn gọn:

1. **Toán tử ??:**

```
int? a = null;
int b = a ?? 10; // Nếu a là null, b sẽ nhận giá trị 10
```

2. **Toán tử ?. (Null-conditional operator):** Dùng để tránh lỗi NullReferenceException khi truy cập thuộc tính hoặc phương thức của một đối tượng nullable.

```
string? name = null;
int? length = name?.Length; // Không gây lỗi, trả về null nếu name là null
```

---

5. Lợi ích của Nullable

- Giảm thiểu lỗi NullReferenceException.
- Làm rõ ý nghĩa code: Biến nào có thể null, biến nào không.
- Tăng cường tính an toàn khi làm việc với dữ liệu không đầy đủ (ví dụ: dữ liệu từ cơ sở dữ liệu).

---

6. Thực hành

Ví dụ: Tính tổng giá trị nullable:

```
int? a = 5;
int? b = null;
int result = (a ?? 0) + (b ?? 0);
Console.WriteLine($"Kết quả: {result}"); // Output: 5
```

Bài 33: Tìm về delegates and anonymous methods

1. Giới thiệu về Delegate

- **Delegate là gì?**
  - Delegate trong C# là một kiểu dữ liệu đại diện cho các phương thức có chữ ký (signature) tương ứng.
  - Delegate giống như một con trỏ hàm trong các ngôn ngữ lập trình khác nhưng an toàn hơn.
- **Tại sao cần sử dụng Delegate?**
  - Tăng tính linh hoạt và tái sử dụng mã nguồn.
  - Thay đổi cách gọi phương thức mà không cần sửa đổi nhiều code.
  - Sử dụng trong lập trình sự kiện (event-driven programming).

---

2. Cách khai báo và sử dụng Delegate

- **Khai báo Delegate:**

```
public delegate void ShowMessage(string message);
```

- **Khởi tạo Delegate:**

```
ShowMessage display = new ShowMessage(Console.WriteLine);
```

- **Gọi phương thức thông qua Delegate:**

```
display("Hello, Delegates!");
```

---

3. Anonymous Methods (Phương thức ẩn danh)

- **Phương thức ẩn danh là gì?**
  - Một cách khai báo phương thức trực tiếp tại nơi sử dụng, không cần đặt tên.
- **Ví dụ sử dụng phương thức ẩn danh với Delegate:**

```
ShowMessage show = delegate (string msg) {
   Console.WriteLine("Anonymous: " + msg);
};
show("Xin chào các bạn!");
```

- **Ưu điểm:**
  - Giảm số dòng code phải viết khi phương thức không được tái sử dụng.
  - Rất hữu ích trong các tình huống callback hoặc sự kiện.

---

4. So sánh Delegate và Anonymous Methods

Delegate	**Anonymous Methods**
Được định nghĩa riêng biệt, có thể tái sử dụng.	Được khai báo tại chỗ, không tái sử dụng.
Yêu cầu khai báo phương thức trước khi sử dụng.	Không yêu cầu phương thức rõ ràng.
Có thể chứa nhiều phương thức trong danh sách.	Chủ yếu dùng cho các tác vụ nhanh gọn, đơn giản.

---

5. Lý thuyết và Thực hành

- **Lý thuyết:**
  - Delegate hoạt động như cầu nối để truyền hàm làm tham số.
  - Anonymous methods thường được dùng khi viết code nhanh hoặc xử lý sự kiện.

6. Mở rộng

- **Lambdas (Biểu thức Lambda):**
    - Là một cách viết tắt gọn gàng hơn của anonymous methods.

      ShowMessage show = (msg) => Console.WriteLine("Lambda: " + msg);

      show("Hello, Lambdas!");
    - Đây là bước đệm quan trọng để tìm hiểu LINQ và lập trình functional trong C#.
- **Ứng dụng thực tế:**
    - Delegate thường được dùng trong lập trình sự kiện và thiết kế callback.
    - Anonymous methods hay Lambda expressions phổ biến trong việc xử lý dữ liệu với LINQ.

  Ví dụ 1: Delegate - Tính toán với các phép toán khác nhau

  Ví dụ 2: Anonymous Methods - Lọc danh sách số chẵn

Bài 34: Cơ bản về LINQ và biểu thức Lambda

1. LINQ là gì?

- LINQ (Language Integrated Query) là một tính năng mạnh mẽ trong C# giúp thực hiện các truy vấn dữ liệu một cách đơn giản và trực quan.
- LINQ có thể làm việc với nhiều nguồn dữ liệu khác nhau:
    - **LINQ to Objects**: Dữ liệu từ các collections như List<T>, Array, v.v.
    - **LINQ to SQL**: Làm việc với cơ sở dữ liệu SQL Server.
    - **LINQ to XML**: Xử lý dữ liệu XML.
    - **LINQ to Entities**: Truy vấn dữ liệu qua Entity Framework.

---

2. Biểu thức Lambda là gì?

- Biểu thức Lambda là một cách viết ngắn gọn của các hàm vô danh (anonymous functions) trong C#.

  (tham_số) => biểu_thức

  Ví dụ:

  x => x * x // Hàm nhận vào x, trả về x bình phương

Kết hợp Lambda với LINQ:

Lambda thường được sử dụng trong các phương thức LINQ như: Where, Select, OrderBy, v.v.

---

3. Cách sử dụng LINQ

LINQ hoạt động thông qua 2 dạng chính:

1. **Query Syntax**: Sử dụng cú pháp gần giống SQL.

   var results = from item in list

   where item > 5

   select item;

2. **Method Syntax**: Sử dụng phương thức mở rộng (Extension Methods).

   var results = list.Where(item => item > 5).Select(item => item);

---

4. Ví dụ cụ thể

Ví dụ 1: LINQ to Objects

```
using System;
using System.Linq;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

        // Query Syntax
        var evenNumbersQuery = from number in numbers
                               where number % 2 == 0
                               select number;

        // Method Syntax
        var evenNumbersMethod = numbers.Where(number => number % 2 == 0);

        Console.WriteLine("Even Numbers (Query): " + string.Join(", ",
evenNumbersQuery));
        Console.WriteLine("Even Numbers (Method): " + string.Join(", ",
evenNumbersMethod));
    }
}
```

Ví dụ 2: LINQ với Biểu thức Lambda

```
using System;
using System.Linq;

class Program
{
    static void Main()
```

```
    {
        string[] fruits = { "Apple", "Banana", "Cherry", "Date" };

        // Lọc các từ có độ dài > 5
        var longFruits = fruits.Where(fruit => fruit.Length > 5);

        Console.WriteLine("Fruits with more than 5 characters: " + string.Join(", ",
longFruits));
    }
}
```

---

5. Một số phương thức phổ biến của LINQ

- **Where:** Lọc dữ liệu theo điều kiện.

  var result = list.Where(x => x > 5);

- **Select:** Biến đổi dữ liệu.

  var result = list.Select(x => x * 2);

- **OrderBy:** Sắp xếp tăng dần.

  var result = list.OrderBy(x => x);

- **OrderByDescending:** Sắp xếp giảm dần.

  var result = list.OrderByDescending(x => x);

- **GroupBy:** Nhóm các phần tử.

  var result = list.GroupBy(x => x % 2);

- **First, FirstOrDefault:** Lấy phần tử đầu tiên (hoặc giá trị mặc định nếu không có).

  var result = list.FirstOrDefault(x => x > 10);

- **Count:** Đếm số phần tử.

  var count = list.Count(x => x > 5);

---

6. Lợi ích của LINQ và Lambda

- **Mã nguồn ngắn gọn:** LINQ kết hợp Lambda giúp giảm thiểu số dòng mã so với cách viết truyền thống.
- **Đọc dễ dàng:** Các câu lệnh LINQ mang tính tự nhiên, dễ đọc hiểu.
- **Tái sử dụng:** LINQ có thể áp dụng cho nhiều nguồn dữ liệu khác nhau.

---

7. Bài tập thực hành

1. Viết chương trình tìm tất cả các số lẻ trong một danh sách các số nguyên.
2. Sử dụng LINQ để lấy danh sách các chuỗi có độ dài > 3 ký tự từ một mảng chuỗi.
3. Tạo một mảng số nguyên, nhân đôi các giá trị và sắp xếp chúng giảm dần.

Bài 35: Sử dụng LINQ với collections

1. Các phương thức LINQ cơ bản với Collections

a. Where

Lọc các phần tử trong một collection thỏa mãn điều kiện.

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };
var evenNumbers = numbers.Where(n => n % 2 == 0);
foreach (var num in evenNumbers)
{
    Console.WriteLine(num); // Output: 2, 4, 6
}
```

b. Select

Dùng để chiếu (transform) dữ liệu thành một dạng khác.

```
var names = new List<string> { "Alice", "Bob", "Charlie" };
var upperNames = names.Select(name => name.ToUpper());
foreach (var name in upperNames)
{
    Console.WriteLine(name); // Output: ALICE, BOB, CHARLIE
}
```

c. OrderBy và OrderByDescending

Sắp xếp các phần tử theo thứ tự tăng hoặc giảm.

```
var numbers = new List<int> { 5, 3, 8, 1 };
var sortedNumbers = numbers.OrderBy(n => n);

foreach (var num in sortedNumbers)
{
    Console.WriteLine(num); // Output: 1, 3, 5, 8
}
```

d. GroupBy

Nhóm các phần tử theo một tiêu chí.

```
var students = new List<string> { "Alice", "Bob", "Charlie", "Anna" };
var groupedStudents = students.GroupBy(s => s[0]);
foreach (var group in groupedStudents)
{
    Console.WriteLine($"Key: {group.Key}");
    foreach (var student in group)
    {
        Console.WriteLine(student);
    }
}
// Output:
```

```
// Key: A

// Alice, Anna

// Key: B

// Bob

// Key: C

// Charlie
```

e. First, FirstOrDefault, Last, LastOrDefault

Lấy phần tử đầu tiên, phần tử cuối cùng, hoặc giá trị mặc định nếu không tìm thấy.

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };
Console.WriteLine(numbers.First()); // Output: 1
Console.WriteLine(numbers.Last()); // Output: 5
Console.WriteLine(numbers.FirstOrDefault(n => n > 5)); // Output: 0 (mặc định)
```

f. Aggregate

Thực hiện các phép tính tổng hợp trên collection.

```
var numbers = new List<int> { 1, 2, 3, 4 };
var sum = numbers.Aggregate((total, next) => total + next);
Console.WriteLine(sum); // Output: 10
```

---

4. Ứng dụng LINQ trong thực tế

a. Tìm kiếm

```
var products = new List<string> { "Apple", "Banana", "Cherry" };
var searchResult = products.Where(p => p.Contains("a"));
foreach (var product in searchResult)
{
    Console.WriteLine(product); // Output: Banana, Cherry
}
```

b. Phân nhóm

```
var orders = new List<int> { 1001, 1002, 2001, 2002, 3001 };
var groupedOrders = orders.GroupBy(order => order / 1000);
foreach (var group in groupedOrders)
{
    Console.WriteLine($"Group: {group.Key}");
    foreach (var order in group)
    {
        Console.WriteLine(order);
    }
}
// Output:

// Group: 1

// 1001, 1002
```

```
// Group: 2

// 2001, 2002

// Group: 3

// 3001
```

c. Phân trang

```
var items = Enumerable.Range(1, 20);
int pageSize = 5;
int page = 2;
var pagedItems = items.Skip((page - 1) * pageSize).Take(pageSize);
foreach (var item in pagedItems)
{
    Console.WriteLine(item); // Output: 6, 7, 8, 9, 10
}
```

---

5. Thực hành

1. Viết chương trình lọc các số chia hết cho 3 trong một danh sách.
2. Viết chương trình nhóm tên sinh viên theo ký tự đầu tiên.
3. Tạo danh sách sản phẩm, sắp xếp theo giá từ cao đến thấp.

---

6. Kết luận

LINQ giúp code trở nên gọn gàng, dễ hiểu và mạnh mẽ hơn. Bằng cách nắm vững LINQ, bạn có thể xử lý dữ liệu hiệu quả hơn trong các dự án thực tế.

Bài 36: Events and event-driven programming

1. Tổng quan về Events và Event-driven Programming

- **Events** là cơ chế giao tiếp giữa các đối tượng trong lập trình C#.
- **Event-driven programming** (lập trình hướng sự kiện) là một mô hình lập trình dựa trên việc xử lý các sự kiện, chẳng hạn như nhấp chuột, gõ phím, hoặc thay đổi trạng thái.

2. Events trong C#

- **Event** là một **delegate** đặc biệt dùng để thông báo khi một hành động cụ thể xảy ra.
- Cú pháp khai báo:

```
public event EventHandler EventName;
```

- EventHandler là một delegate được định nghĩa sẵn trong .NET.
- EventName là tên của sự kiện.

- **Delegate cơ bản**:

```
public delegate void MyEventHandler(object sender, EventArgs e);
```

3. Cách sử dụng Events

Bước 1: Khai báo event.
Bước 2: Gắn event với một phương thức xử lý (event handler).
Bước 3: Kích hoạt event khi điều kiện được thỏa mãn.

Ví dụ:

```
class Program
{
    // Bước 1: Khai báo event
    public event EventHandler MyEvent;

    public void TriggerEvent()
    {
        // Bước 3: Kích hoạt event
        MyEvent?.Invoke(this, EventArgs.Empty);
    }

    static void Main()
    {
        Program program = new Program();

        // Bước 2: Gắn event với phương thức xử lý
        program.MyEvent += Program_MyEvent;

        // Gọi event
        program.TriggerEvent();
    }

    static void Program_MyEvent(object sender, EventArgs e)
    {
        Console.WriteLine("Event được kích hoạt!");
    }
}
```

4. Các thành phần chính trong Event-driven Programming

- **Publisher**: Là đối tượng tạo và kích hoạt event.
- **Subscriber**: Là đối tượng lắng nghe và xử lý event.
- **Event Handler**: Là phương thức xử lý sự kiện, thường có kiểu delegate tương thích.

5. Tùy chỉnh EventArgs

Nếu sự kiện cần truyền thêm thông tin, ta có thể tạo một lớp kế thừa từ EventArgs.

Ví dụ:

```
public class CustomEventArgs : EventArgs
{
    public string Message { get; set; }
}

class Program
{
    public event EventHandler<CustomEventArgs> MyEvent;

    public void TriggerEvent(string message)
    {
        MyEvent?.Invoke(this, new CustomEventArgs { Message = message });
    }

    static void Main()
    {
        Program program = new Program();
        program.MyEvent += Program_MyEvent;

        program.TriggerEvent("Xin chào, đây là sự kiện tuỳ chỉnh!");
    }

    static void Program_MyEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"Event nhận được: {e.Message}");
    }
}
```

6. Ứng dụng thực tế

- Xử lý sự kiện trong giao diện người dùng (Windows Forms, WPF).
- Quản lý các tác vụ bất đồng bộ (asynchronous programming).
- Hệ thống thông báo trong các ứng dụng lớn (Pub-Sub model).

7. Lưu ý khi sử dụng Events

- **Unsubscribe sự kiện**: Luôn đảm bảo hủy đăng ký sự kiện khi không cần thiết để tránh rò rỉ bộ nhớ.

  ```
  program.MyEvent -= Program_MyEvent;
  ```

- Sử dụng ?. để kiểm tra event có null trước khi gọi:

  ```
  MyEvent?.Invoke(this, EventArgs.Empty);
  ```

8. Bài tập thực hành

1. Tạo một chương trình quản lý danh sách công việc, nơi mỗi khi thêm công việc mới, hệ thống thông báo cho người dùng.
2. Mô phỏng hệ thống chuông báo thức, nơi khi đến giờ, chuông sẽ kêu và hiển thị thông báo.

Bài 37: Đọc và ghi file/folder sử dụng I/O

Mục tiêu:

- Hiểu các lớp cơ bản trong **System.IO** dùng để làm việc với file.
- Biết cách sử dụng **File**, **StreamReader/StreamWriter**, **FileStream**, và **BinaryReader/BinaryWriter**.
- Nắm được sự khác nhau giữa các lớp này và khi nào nên sử dụng.

---

1. Giới thiệu về System.IO

System.IO cung cấp các lớp và phương thức để thao tác với file và thư mục, gồm:

- **File**: Cung cấp các phương thức tĩnh để thao tác file.
- **StreamReader/StreamWriter**: Làm việc với chuỗi (text).
- **FileStream**: Làm việc với dữ liệu nhị phân (binary) hoặc text.
- **BinaryReader/BinaryWriter**: Đọc/ghi dữ liệu nhị phân cấp cao hơn.

---

2. So sánh các lớp

Lớp	**Loại dữ liệu**	**Đặc điểm**	**Khi nào sử dụng**
File	Text	- Dễ sử dụng, gồm các phương thức tĩnh như File.ReadAllText, File.WriteAllText, File.ReadAllLines.	- Khi cần đọc/ghi file nhỏ một cách nhanh chóng mà không cần quản lý nhiều về hiệu năng hoặc tài nguyên.
StreamReader/Writer	Text	- Hoạt động theo dòng, tối ưu khi xử lý file lớn (không cần đọc toàn bộ vào bộ nhớ).	- Khi cần đọc/ghi dữ liệu dạng text dòng theo dòng.
FileStream	Binary/Text	- Cấp thấp, dùng để làm việc với file ở mức byte hoặc kết hợp với lớp khác (như StreamReader).	- Khi cần kiểm soát chi tiết dữ liệu đọc/ghi, hoặc xử lý file nhị phân (như hình ảnh).
BinaryReader/Writer	Binary	- Đọc/Ghi dữ liệu nhị phân (số nguyên, số thực, mảng byte, v.v.) dễ dàng hơn.	- Khi làm việc với dữ liệu nhị phân có cấu trúc, ví dụ file lưu trữ trạng thái hoặc cấu trúc phức tạp.

---

3. Ví dụ minh họa

3.1. Sử dụng File

```
// Ghi toàn bộ text vào file
File.WriteAllText("example.txt", "Hello, File!");

// Đọc toàn bộ nội dung từ file
string content = File.ReadAllText("example.txt");
Console.WriteLine(content);
```

Ưu điểm: Nhanh chóng, tiện lợi.
Nhược điểm: Không phù hợp với file lớn (sẽ đọc toàn bộ vào bộ nhớ).

3.2. Sử dụng StreamReader/StreamWriter

```
// Ghi file theo dòng
using (StreamWriter writer = new StreamWriter("example.txt"))
{
    writer.WriteLine("Line 1");
    writer.WriteLine("Line 2");
}

// Đọc file theo dòng
using (StreamReader reader = new StreamReader("example.txt"))
{
    string line;
    while ((line = reader.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
}
```

Ưu điểm: Tối ưu bộ nhớ, phù hợp với file lớn.
Nhược điểm: Chỉ dùng với text.

3.3. Sử dụng FileStream

```
// Ghi dữ liệu nhị phân
using (FileStream fs = new FileStream("example.bin", FileMode.Create))
{
    byte[] data = { 1, 2, 3, 4, 5 };
    fs.Write(data, 0, data.Length);
}

// Đọc dữ liệu nhị phân
using (FileStream fs = new FileStream("example.bin", FileMode.Open))
{
    byte[] data = new byte[fs.Length];
    fs.Read(data, 0, data.Length);
    Console.WriteLine(string.Join(", ", data));
}
```

Ưu điểm: Kiểm soát chi tiết dữ liệu.
Nhược điểm: Phức tạp hơn khi xử lý text.

3.4. Sử dụng BinaryReader/BinaryWriter

```
// Ghi dữ liệu nhị phân
using (BinaryWriter writer = new BinaryWriter(File.Open("example.dat",
FileMode.Create)))
{
    writer.Write(123);    // Ghi số nguyên
    writer.Write(45.67);  // Ghi số thực
    writer.Write("Hello"); // Ghi chuỗi
}

// Đọc dữ liệu nhị phân
using (BinaryReader reader = new BinaryReader(File.Open("example.dat",
FileMode.Open)))
{
    int number = reader.ReadInt32();
    double value = reader.ReadDouble();
    string text = reader.ReadString();

    Console.WriteLine($"{number}, {value}, {text}");
}
```

Ưu điểm: Hỗ trợ nhiều kiểu dữ liệu có cấu trúc.
Nhược điểm: Chỉ làm việc với dữ liệu nhị phân.

---

4. Khi nào sử dụng cái gì?

- **File**:
  Sử dụng khi cần thao tác nhanh với file nhỏ và không cần kiểm soát chi tiết.
  Ví dụ: Đọc/Ghi cấu hình, file log ngắn.
- **StreamReader/StreamWriter**:
  Dùng cho file text lớn, khi cần đọc/ghi theo dòng.
  Ví dụ: Đọc file CSV, file nhật ký (log).
- **FileStream**:
  Dùng khi cần thao tác chi tiết với dữ liệu ở mức byte.
  Ví dụ: Làm việc với file hình ảnh, âm thanh, hoặc tạo lớp wrapper riêng.
- **BinaryReader/BinaryWriter**:
  Dùng khi đọc/ghi dữ liệu nhị phân có cấu trúc cụ thể.
  Ví dụ: File lưu trạng thái game, file nhị phân nén.

---

5. Tổng kết

- Mỗi lớp trong **System.IO** phục vụ một mục đích khác nhau, tùy thuộc vào nhu cầu xử lý file (text hay nhị phân) và quy mô file (nhỏ hay lớn).
- Lựa chọn đúng công cụ sẽ giúp tối ưu hiệu năng và đơn giản hóa công việc.

Làm việc với Directory trong C# (Namespace System.IO)

Ngoài việc thao tác với file, C# cung cấp các lớp hỗ trợ quản lý thư mục (**Directory**) và các đối tượng liên quan. Hai lớp chính dùng để làm việc với thư mục là:

- **Directory**: Cung cấp các phương thức tĩnh để làm việc với thư mục.
- **DirectoryInfo**: Cung cấp các phương thức và thuộc tính cho các thao tác chi tiết hơn.

---

1. Sử dụng lớp Directory

Lớp Directory gồm các phương thức tĩnh để thực hiện các thao tác cơ bản với thư mục.

1.1. Tạo thư mục

```
string folderPath = "TestDirectory";

// Tạo thư mục nếu chưa tồn tại
if (!Directory.Exists(folderPath))
{
    Directory.CreateDirectory(folderPath);
    Console.WriteLine($"Thư mục '{folderPath}' đã được tạo.");
}
else
{
    Console.WriteLine($"Thư mục '{folderPath}' đã tồn tại.");
}
```

1.2. Xóa thư mục

```
// Xóa thư mục
if (Directory.Exists(folderPath))
{
    Directory.Delete(folderPath, true); // Tham số `true` để xóa thư mục kèm nội
dung bên trong
    Console.WriteLine($"Thư mục '{folderPath}' đã được xóa.");
}
else
{
    Console.WriteLine($"Thư mục '{folderPath}' không tồn tại.");
}
```

1.3. Lấy danh sách file trong thư mục

```
string[] files = Directory.GetFiles("TestDirectory");
foreach (string file in files)
{
    Console.WriteLine($"File: {file}");
}
```

1.4. Lấy danh sách thư mục con

```
string[] subDirs = Directory.GetDirectories("TestDirectory");
```

```
foreach (string dir in subDirs)
{
    Console.WriteLine($"Thư mục con: {dir}");
}
```

1.5. Di chuyển thư mục

```
string sourceDir = "TestDirectory";
string destDir = "NewDirectory";

if (Directory.Exists(sourceDir))
{
    Directory.Move(sourceDir, destDir);
    Console.WriteLine($"Đã di chuyển thư mục từ '{sourceDir}' đến '{destDir}'.");
}
```

---

2. Sử dụng lớp DirectoryInfo

Lớp DirectoryInfo cung cấp các phương thức và thuộc tính làm việc với thư mục theo kiểu hướng đối tượng.

2.1. Tạo thư mục

```
DirectoryInfo dirInfo = new DirectoryInfo("TestDirectory");
if (!dirInfo.Exists)
{
    dirInfo.Create();
    Console.WriteLine($"Thư mục '{dirInfo.FullName}' đã được tạo.");
}
```

2.2. Lấy thông tin chi tiết

```
using System.Runtime;

if (dirInfo.Exists)
{
    Console.WriteLine($"Tên thư mục: {dirInfo.Name}");
    Console.WriteLine($"Đường dẫn đầy đủ: {dirInfo.FullName}");
    Console.WriteLine($"Thời gian tạo: {dirInfo.CreationTime}");
}
```

2.3. Lấy danh sách file

```
FileInfo[] files = dirInfo.GetFiles();
foreach (FileInfo file in files)
{
    Console.WriteLine($"File: {file.Name} - Kích thước: {file.Length} bytes");
}
```

2.4. Lấy danh sách thư mục con

```
DirectoryInfo[] subDirectories = dirInfo.GetDirectories();
foreach (DirectoryInfo subDir in subDirectories)
{
    Console.WriteLine($"Thư mục con: {subDir.Name}");
}
```

2.5. Xóa thư mục

```
using System.Runtime;

if (dirInfo.Exists)
```

```
{
    dirInfo.Delete(true); // Tham số `true` để xóa thư mục kèm nội dung
    Console.WriteLine($"Thư mục '{dirInfo.Name}' đã được xóa.");
}
```

---

3. So sánh Directory và DirectoryInfo

Cả hai lớp Directory và DirectoryInfo đều được sử dụng để làm việc với thư mục trong C#, nhưng mỗi lớp phù hợp với các trường hợp sử dụng khác nhau tùy thuộc vào nhu cầu và ngữ cảnh.

Tiêu chí	Directory	DirectoryInfo
Cách sử dụng	Sử dụng phương thức **tĩnh** (static).	Làm việc thông qua một **thể hiện đối tượng** (instance).
Hiệu suất	Mỗi lần gọi phương thức, dữ liệu sẽ được đọc từ hệ thống tập tin.	Giữ lại trạng thái của thư mục (có thể tái sử dụng đối tượng để tối ưu).
Đơn giản	Dễ sử dụng, thích hợp cho các thao tác đơn giản và nhanh chóng.	Yêu cầu khởi tạo đối tượng, thích hợp cho các thao tác chi tiết và phức tạp hơn.
Thông tin chi tiết	Không cung cấp thông tin chi tiết như ngày tạo, ngày sửa đổi của thư mục.	Cung cấp thông tin chi tiết như ngày tạo (CreationTime), ngày sửa đổi (LastWriteTime), v.v.
Khả năng mở rộng	Không thể mở rộng (do các phương thức là tĩnh).	Có thể mở rộng, kết hợp với các phương thức instance hoặc lớp con.
Tính năng bổ sung	Tập trung vào các thao tác cơ bản như tạo, xóa, di chuyển, lấy danh sách file/thư mục.	Cung cấp thêm các tính năng quản lý và truy vấn thông tin thư mục chi tiết hơn.
Tính linh hoạt	Ít linh hoạt hơn, phải sử dụng phương thức tĩnh nhiều lần nếu cần làm việc với nhiều thư mục.	Linh hoạt hơn khi làm việc với một thư mục hoặc nhiều thư mục liên quan nhờ tái sử dụng đối tượng.

Khi nào sử dụng Directory?

Directory phù hợp khi bạn:

- Chỉ cần thực hiện các thao tác cơ bản trên thư mục (tạo, xóa, kiểm tra sự tồn tại).
- Muốn viết mã đơn giản, nhanh chóng, không cần lưu trạng thái thư mục.
- Làm việc với nhiều thư mục khác nhau và không cần giữ thông tin chi tiết giữa các lần thao tác.

Ví dụ:

```
// Kiểm tra và tạo thư mục nếu chưa tồn tại
if (!Directory.Exists("MyFolder"))
{
    Directory.CreateDirectory("MyFolder");
}
```

Khi nào sử dụng DirectoryInfo?

DirectoryInfo phù hợp khi bạn:

- Cần truy cập các thông tin chi tiết về thư mục (thời gian tạo, thời gian sửa đổi, v.v.).
- Làm việc với một thư mục cụ thể trong nhiều thao tác khác nhau.
- Cần hiệu năng tốt hơn khi phải truy vấn hoặc thao tác lặp lại trên cùng một thư mục.
- Muốn viết mã theo hướng đối tượng để dễ mở rộng và quản lý.

Ví dụ:

```
// Truy xuất thông tin chi tiết về thư mục
DirectoryInfo dirInfo = new DirectoryInfo("MyFolder");
if (!dirInfo.Exists)
{
    dirInfo.Create();
}

Console.WriteLine($"Thư mục: {dirInfo.FullName}");
Console.WriteLine($"Ngày tạo: {dirInfo.CreationTime}");
Console.WriteLine($"Ngày sửa đổi: {dirInfo.LastWriteTime}");
```

Một số tình huống kết hợp

Trong nhiều trường hợp, bạn có thể kết hợp cả hai lớp để tận dụng ưu điểm của từng lớp:

1. Sử dụng Directory để kiểm tra sự tồn tại hoặc tạo nhanh thư mục.
2. Sử dụng DirectoryInfo để truy vấn chi tiết thông tin hoặc thao tác liên tục trên thư mục.

Ví dụ:

```
// Kết hợp Directory và DirectoryInfo
if (!Directory.Exists("MyFolder"))
{
    Directory.CreateDirectory("MyFolder");
}

DirectoryInfo dirInfo = new DirectoryInfo("MyFolder");
Console.WriteLine($"Thư mục: {dirInfo.FullName}");
Console.WriteLine($"Ngày tạo: {dirInfo.CreationTime}");
```

Tóm tắt

- **Chọn Directory**: Khi cần thao tác nhanh gọn, đơn giản.

- **Chọn DirectoryInfo**: Khi cần thông tin chi tiết hoặc thực hiện nhiều thao tác trên cùng một thư mục.

---

4. Một số tình huống thực tế

4.1. Tạo cây thư mục

```
string rootDir = "RootDirectory";
string[] subDirs = { "SubDir1", "SubDir2", "SubDir3" };

Directory.CreateDirectory(rootDir);
foreach (string subDir in subDirs)
{
    Directory.CreateDirectory(Path.Combine(rootDir, subDir));
}
Console.WriteLine("Cây thư mục đã được tạo.");
```

4.2. Tìm file theo mẫu (pattern)

```
string[] txtFiles = Directory.GetFiles("TestDirectory", "*.txt");
foreach (string file in txtFiles)
{
    Console.WriteLine($"Tìm thấy file: {file}");
}
```

4.3. Tính tổng kích thước file trong thư mục

```
DirectoryInfo dir = new DirectoryInfo("TestDirectory");
long totalSize = dir.GetFiles().Sum(file => file.Length);

Console.WriteLine($"Tổng kích thước file trong thư mục: {totalSize} bytes");
```

---

5. Lưu ý

- Khi làm việc với thư mục/file lớn, cần chú ý đến hiệu năng (sử dụng các thao tác không đồng bộ nếu có thể).
- Sử dụng Path để xây dựng đường dẫn file một cách an toàn và tránh lỗi định dạng. Ví dụ:

```
string fullPath = Path.Combine("RootDirectory", "SubDirectory", "file.txt");
```

Bài 38: Làm việc với dữ liệu JSON và XML

1. Làm việc với JSON trong C#

JSON (JavaScript Object Notation) là một định dạng dữ liệu nhẹ, dễ đọc và dễ viết. Trong C#, bạn có thể sử dụng **System.Text.Json** (có sẵn từ .NET Core 3.0) hoặc **Newtonsoft.Json** (thư viện phổ biến) để làm việc với JSON.

a. Đọc JSON từ một chuỗi

Để đọc dữ liệu JSON từ một chuỗi và chuyển nó thành đối tượng C#:

```
using System;
using System.Text.Json;

class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        string jsonString = "{\"Name\":\"John\", \"Age\":30}";

        // Deserialize JSON thành đối tượng Person
        Person person = JsonSerializer.Deserialize<Person>(jsonString);

        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
    }
}
```

b. Viết dữ liệu ra JSON

Để chuyển một đối tượng C# thành JSON:

```
using System;
using System.Text.Json;

class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        Person person = new Person { Name = "John", Age = 30 };

        // Serialize đối tượng thành JSON
        string jsonString = JsonSerializer.Serialize(person);

        Console.WriteLine(jsonString);
    }
}
```

c. Đọc JSON từ file

Để đọc dữ liệu JSON từ một file và chuyển thành đối tượng:

```
using System;
using System.IO;
using System.Text.Json;

class Program
{
    public class Person
```

```
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        string jsonString = File.ReadAllText("person.json");

        // Deserialize từ file JSON thành đối tượng Person
        Person person = JsonSerializer.Deserialize<Person>(jsonString);

        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
    }
}
```

d. Viết JSON ra file

Để ghi dữ liệu JSON ra file:

```
using System;
using System.IO;
using System.Text.Json;

class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        Person person = new Person { Name = "John", Age = 30 };

        string jsonString = JsonSerializer.Serialize(person);

        File.WriteAllText("person.json", jsonString);
    }
}
```

2. Làm việc với XML trong C#

XML (Extensible Markup Language) là một định dạng dữ liệu cấu trúc cho phép lưu trữ và truyền tải dữ liệu. C# cung cấp các lớp trong **System.Xml** để làm việc với XML.

a. Đọc XML từ chuỗi

Để đọc XML từ một chuỗi và chuyển nó thành đối tượng C#:

```
using System;
using System.Xml.Serialization;
using System.IO;

class Program
{
    public class Person
```

```
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        string xmlString = "<Person><Name>John</Name><Age>30</Age></Person>";

        // Deserialize XML thành đối tượng Person
        XmlSerializer serializer = new XmlSerializer(typeof(Person));
        using (StringReader reader = new StringReader(xmlString))
        {
            Person person = (Person)serializer.Deserialize(reader);
            Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
        }
    }
}
```

b. Viết XML ra chuỗi

Để chuyển đối tượng C# thành XML:

```
using System;
using System.Xml.Serialization;
using System.IO;

class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        Person person = new Person { Name = "John", Age = 30 };

        XmlSerializer serializer = new XmlSerializer(typeof(Person));

        using (StringWriter writer = new StringWriter())
        {
            // Serialize đối tượng thành XML
            serializer.Serialize(writer, person);
            Console.WriteLine(writer.ToString());
        }
    }
}
```

c. Đọc XML từ file

Để đọc XML từ một file và chuyển thành đối tượng:

```
using System;
using System.Xml.Serialization;
using System.IO;
```

```
class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        XmlSerializer serializer = new XmlSerializer(typeof(Person));

        using (FileStream fs = new FileStream("person.xml", FileMode.Open))
        {
            // Deserialize từ file XML thành đối tượng Person
            Person person = (Person)serializer.Deserialize(fs);
            Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
        }
    }
}
```

d. Viết XML ra file

Để ghi XML ra file:

```
using System;
using System.Xml.Serialization;
using System.IO;

class Program
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    static void Main()
    {
        Person person = new Person { Name = "John", Age = 30 };

        XmlSerializer serializer = new XmlSerializer(typeof(Person));

        using (FileStream fs = new FileStream("person.xml", FileMode.Create))
        {
            // Serialize đối tượng thành XML và ghi ra file
            serializer.Serialize(fs, person);
        }
    }
}
```

3. Kết luận

- **JSON** là định dạng dữ liệu nhẹ và dễ đọc, phù hợp với các ứng dụng web và mobile, dễ sử dụng với **System.Text.Json** hoặc **Newtonsoft.Json**.

- **XML** là một định dạng mạnh mẽ cho dữ liệu cấu trúc, đặc biệt hữu ích trong các hệ thống phức tạp hoặc khi cần tương tác với các dịch vụ Web cũ.

Bài 39: Serialization và deserialization trong C#

1. Giới thiệu về Serialization và Deserialization:

- **Serialization** (Tuần tự hóa): Là quá trình chuyển đổi đối tượng trong bộ nhớ thành một dạng dữ liệu có thể lưu trữ hoặc truyền tải qua mạng, ví dụ như chuỗi byte hoặc JSON.
- **Deserialization** (Giải tuần tự hóa): Là quá trình ngược lại, chuyển đổi dữ liệu (thường là chuỗi byte, JSON, XML) trở lại thành đối tượng trong bộ nhớ.

2. Các hình thức Serialization phổ biến trong C#:

- **Binary Serialization**: Chuyển đổi đối tượng thành định dạng nhị phân. Dữ liệu nhị phân giúp tiết kiệm không gian và dễ dàng sử dụng khi truyền qua mạng hoặc lưu trữ vào file.
- **XML Serialization**: Chuyển đổi đối tượng thành XML (Extensible Markup Language), rất hữu ích khi dữ liệu cần chia sẻ qua các hệ thống hoặc giữa các ngôn ngữ khác nhau.
- **JSON Serialization**: Chuyển đổi đối tượng thành JSON (JavaScript Object Notation), định dạng phổ biến trong các ứng dụng web và API.

3. Cách thực hiện Serialization và Deserialization trong C#:

3.1. Binary Serialization:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable] // Đánh dấu lớp có thể tuần tự hóa
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class Program
{
    static void Main()
    {
        // Tạo đối tượng
        Person person = new Person() { Name = "Alice", Age = 30 };

        // Serialize đối tượng thành file nhị phân
        IFormatter formatter = new BinaryFormatter();
        using (Stream stream = new FileStream("person.dat", FileMode.Create,
FileAccess.Write))
        {
            formatter.Serialize(stream, person);
        }
```

```
        // Deserialize đối tượng từ file nhị phân
        using (Stream stream = new FileStream("person.dat", FileMode.Open,
FileAccess.Read))
        {
            Person deserializedPerson = (Person)formatter.Deserialize(stream);
            Console.WriteLine($"Name: {deserializedPerson.Name}, Age:
{deserializedPerson.Age}");
        }
    }
}
```

3.2. XML Serialization:

```
using System;
using System.IO;
using System.Xml.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

class Program
{
    static void Main()
    {
        // Tạo đối tượng
        Person person = new Person() { Name = "Bob", Age = 25 };

        // Serialize đối tượng thành XML
        XmlSerializer serializer = new XmlSerializer(typeof(Person));
        using (TextWriter writer = new StreamWriter("person.xml"))
        {
            serializer.Serialize(writer, person);
        }

        // Deserialize đối tượng từ XML
        using (TextReader reader = new StreamReader("person.xml"))
        {
            Person deserializedPerson = (Person)serializer.Deserialize(reader);
            Console.WriteLine($"Name: {deserializedPerson.Name}, Age:
{deserializedPerson.Age}");
        }
    }
}
```

3.3. JSON Serialization (với thư viện Newtonsoft.Json):

```
using System;
using Newtonsoft.Json;
using System.IO;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
class Program
{
    static void Main()
    {
        // Tạo đối tượng
        Person person = new Person() { Name = "Charlie", Age = 35 };

        // Serialize đối tượng thành JSON
        string json = JsonConvert.SerializeObject(person);
        File.WriteAllText("person.json", json);

        // Deserialize đối tượng từ JSON
        string jsonFromFile = File.ReadAllText("person.json");
        Person deserializedPerson =
JsonConvert.DeserializeObject<Person>(jsonFromFile);
        Console.WriteLine($"Name: {deserializedPerson.Name}, Age:
{deserializedPerson.Age}");
    }
}
```

4. Lợi ích và ứng dụng của Serialization và Deserialization:

- **Lưu trữ và Truyền tải Dữ liệu**: Serialization rất hữu ích khi bạn cần lưu trữ trạng thái của đối tượng vào file hoặc truyền tải qua mạng, ví dụ như trong các ứng dụng phân tán.
- **Dễ dàng chuyển giao giữa các hệ thống**: Với XML hoặc JSON, bạn có thể dễ dàng chuyển giao dữ liệu giữa các hệ thống sử dụng các ngôn ngữ khác nhau.
- **Ứng dụng trong Web APIs**: JSON Serialization là kỹ thuật cơ bản trong việc xây dựng các RESTful API, giúp client và server giao tiếp với nhau.

5. Lưu ý khi sử dụng Serialization trong C#:

- **[Serializable]**: Để đối tượng có thể được tuần tự hóa, lớp phải được đánh dấu với thuộc tính [Serializable].
- **Không tuần tự hóa các thành phần không cần thiết**: Dùng [NonSerialized] để loại bỏ các thành phần không cần tuần tự hóa.
- **An toàn khi làm việc với các file**: Luôn đảm bảo kiểm tra lỗi khi làm việc với file và stream, tránh mất dữ liệu trong quá trình serialize và deserialize.

Kết luận: Serialization và Deserialization là các kỹ thuật cơ bản trong C# giúp bạn chuyển đổi đối tượng thành định dạng có thể lưu trữ hoặc truyền tải, và ngược lại. Tùy thuộc vào yêu cầu của ứng dụng, bạn có thể chọn các phương pháp như Binary, XML, hoặc JSON serialization để thực hiện.

Bài 40: Giới thiệu từ khóa async và await

Trong C#, từ khóa async và await được sử dụng để làm việc với các tác vụ bất đồng bộ (asynchronous). Việc sử dụng async và await giúp cho các ứng dụng có thể thực hiện nhiều tác vụ cùng một lúc mà không cần phải chặn (blocking) luồng chính, từ đó cải thiện hiệu suất và trải nghiệm người dùng. Dưới đây là một bài giảng chi tiết về cách sử dụng các từ khóa này.

1. Tại sao cần sử dụng bất đồng bộ?

Các tác vụ bất đồng bộ giúp ứng dụng của bạn tiếp tục thực thi các công việc khác trong khi đợi một tác vụ lâu dài (như truy vấn cơ sở dữ liệu, gọi API, đọc/ghi tệp) hoàn thành. Thay vì làm gián đoạn ứng dụng, async và await cho phép ứng dụng xử lý các tác vụ mà không bị chặn lại.

2. Cấu trúc cơ bản của async và await

2.1 Từ khóa async

- Được sử dụng để khai báo một phương thức là bất đồng bộ (asynchronous). Khi phương thức được khai báo là async, bạn có thể sử dụng từ khóa await trong phương thức đó.
- Phương thức async phải trả về một kiểu dữ liệu là Task hoặc Task<T>. Điều này cho phép phương thức thực hiện bất đồng bộ.

2.2 Từ khóa await

- Được sử dụng trong phương thức bất đồng bộ để "chờ" cho một tác vụ hoàn thành mà không chặn luồng thực thi.
- Khi gặp await, chương trình sẽ tạm dừng và tiếp tục thực thi sau khi tác vụ đó hoàn tất.

3. Ví dụ cơ bản

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Bắt đầu.");

        // Gọi phương thức bất đồng bộ
        await Task.Delay(2000);  // Mô phỏng tác vụ tốn thời gian

        Console.WriteLine("Kết thúc sau 2 giây.");
    }
}
```

- Trong ví dụ trên, Task.Delay(2000) là một tác vụ bất đồng bộ giả lập việc chờ đợi trong 2 giây.
- Mặc dù await tạm dừng phương thức Main, nhưng nó không chặn luồng thực thi của ứng dụng. Sau 2 giây, dòng "Kết thúc sau 2 giây." sẽ được in ra.

4. Sử dụng async và await với phương thức trả về giá trị

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task<int> GetDataAsync()
    {
        // Mô phỏng lấy dữ liệu từ một nguồn bất đồng bộ
        await Task.Delay(2000);
```

```
        return 42;  // Giả sử 42 là dữ liệu chúng ta nhận được
    }

    static async Task Main(string[] args)
    {
        Console.WriteLine("Bắt đầu.");

        int result = await GetDataAsync(); // Lấy dữ liệu bất đồng bộ

        Console.WriteLine($"Kết quả nhận được: {result}");
    }
}
```

- Phương thức GetDataAsync trả về một giá trị kiểu int sau khi hoàn thành tác vụ bất đồng bộ.
- Lệnh await GetDataAsync() sẽ trả về kết quả khi phương thức hoàn thành.

5. Kiểu trả về của phương thức async

Các phương thức async có thể trả về:

- Task: Nếu phương thức không trả về giá trị nào (void trong phương thức async).
- Task<T>: Nếu phương thức trả về một giá trị kiểu T bất đồng bộ.
- ValueTask<T>: Một phiên bản hiệu suất cao của Task<T> khi bạn cần trả về kết quả bất đồng bộ nhưng không muốn khởi tạo một đối tượng Task mới mỗi lần.

6. Lưu ý khi sử dụng async và await

- **Không nên sử dụng async void trừ khi bạn đang xử lý sự kiện**. async void không thể trả về Task, điều này làm cho việc kiểm tra lỗi và chờ đợi các tác vụ bất đồng bộ trở nên khó khăn hơn.
- **Xử lý lỗi**: Khi sử dụng async và await, bạn nên sử dụng try-catch để xử lý các ngoại lệ bất đồng bộ.
- **Đảm bảo không chặn luồng UI** (khi phát triển ứng dụng desktop hoặc mobile). Tránh sử dụng các phương thức đồng bộ trong khi gọi các phương thức bất đồng bộ để không gây treo ứng dụng.

7. Ví dụ xử lý lỗi trong async và await

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task<int> GetDataAsync()
    {
        await Task.Delay(1000);
        throw new Exception("Có lỗi xảy ra!");
        return 42;
    }

    static async Task Main(string[] args)
    {
        try
```

```
        {
            int result = await GetDataAsync();
            Console.WriteLine($"Kết quả nhận được: {result}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Lỗi: {ex.Message}");
        }
    }
}
```

Trong ví dụ này, chúng ta đã thêm một lỗi giả trong phương thức GetDataAsync. Khi lỗi xảy ra, nó sẽ được bắt trong khối catch và hiển thị thông báo lỗi.

8. Kết luận

- Từ khóa async và await là công cụ mạnh mẽ giúp chúng ta viết mã bất đồng bộ trong C# một cách đơn giản và dễ hiểu.
- Sử dụng chúng giúp nâng cao hiệu suất ứng dụng, đặc biệt là trong các tác vụ tốn thời gian như gọi API, truy vấn cơ sở dữ liệu, hoặc xử lý tệp lớn.
- Tuy nhiên, bạn cần phải hiểu rõ cách sử dụng chúng và các vấn đề tiềm ẩn như lỗi và hiệu suất để sử dụng chúng hiệu quả.

Khi bạn sử dụng async và await trong C#, một số điểm cần lưu ý về cách thức thực thi và sự khác biệt với lập trình song song (parallel programming) và cơ chế callback.

1. Thread nào thực thi tác vụ khi sử dụng async và await?

Khi bạn sử dụng await, tác vụ bất đồng bộ sẽ không làm gián đoạn luồng (thread) đang thực thi phương thức. Thay vào đó, tác vụ sẽ được thực thi trên **một luồng khác** nếu cần thiết, và khi tác vụ hoàn thành, nó sẽ tiếp tục thực thi trên cùng một luồng mà nó bắt đầu (trừ khi bạn chỉ định rõ ràng điều khác).

Cách thức hoạt động của await:

- Khi bạn gặp await, phương thức async tạm dừng và trả về một Task (hoặc Task<T> nếu trả về giá trị). Quá trình này không chặn luồng (thread) chính. Ví dụ, trong ứng dụng WinForms hoặc WPF, luồng UI có thể tiếp tục xử lý các sự kiện trong khi chờ tác vụ bất đồng bộ hoàn thành.
- **Trên một ứng dụng console hoặc một ứng dụng không có UI**, luồng chính (main thread) sẽ tiếp tục xử lý các công việc khác và chỉ quay lại khi tác vụ bất đồng bộ hoàn thành.
- **Trên các ứng dụng với UI (như WinForms hoặc WPF)**, mặc định await sẽ tiếp tục thực thi mã tiếp theo trên cùng một luồng UI sau khi tác vụ hoàn thành, trừ khi bạn chỉ định rõ ràng (bằng cách sử dụng ConfigureAwait(false)) để tránh việc quay lại luồng UI.

2. So sánh với lập trình song song (parallel programming):

Lập trình song song liên quan đến việc chia nhỏ công việc thành các phần có thể thực thi đồng thời trên nhiều luồng (threads). Tuy nhiên, khi sử dụng async và await, bạn không cần phải quản lý luồng một cách thủ công như trong lập trình song song. Đây là sự khác biệt chính giữa hai phương pháp:

- **Lập trình song song**: Các công việc được chia thành các phần nhỏ và thực thi đồng thời trên nhiều luồng khác nhau. Ví dụ, bạn có thể sử dụng Parallel.For hoặc các API như Task.WhenAll để thực thi nhiều tác vụ song song, mỗi tác vụ có thể được thực thi trên một luồng riêng biệt.
- **Lập trình bất đồng bộ (async/await)**: Dù là bất đồng bộ, nhưng không phải lúc nào các tác vụ đều được thực thi song song. Thay vào đó, các tác vụ có thể chạy trên cùng một luồng nếu không có tác vụ I/O chờ (như đọc/ghi tệp, gọi API) hoặc có thể chạy trên các luồng khác nếu cần.

Sự khác biệt chính: Lập trình song song chia công việc thành các tác vụ đồng thời chạy trên các luồng độc lập, trong khi lập trình bất đồng bộ chỉ tạm dừng luồng khi gặp các tác vụ chờ (I/O), và sau đó quay lại khi các tác vụ hoàn thành. Điều này giúp tiết kiệm tài nguyên, vì bạn không cần phải duy trì nhiều luồng đồng thời như trong lập trình song song.

3. Cơ chế callback trong async và await:

- Trước khi async/await được giới thiệu, lập trình viên thường phải sử dụng các hàm callback để thực hiện các tác vụ bất đồng bộ. Khi một tác vụ bất đồng bộ hoàn thành, callback sẽ được gọi.
- Với async và await, callback không còn cần thiết trong hầu hết các trường hợp. Thay vào đó, bạn viết mã như thể bạn đang xử lý đồng bộ, với từ khóa await giúp tạm dừng và tiếp tục sau khi tác vụ hoàn thành. Điều này giúp mã trở nên dễ đọc và dễ duy trì hơn.

Ví dụ, thay vì sử dụng một hàm callback:

```
public void DoSomethingAsync()
{
    SomeAsyncMethod(result =>
    {
        Console.WriteLine(result);
    });
}
```

Với async và await, bạn có thể viết mã như sau:

```
public async Task DoSomethingAsync()
{
    var result = await SomeAsyncMethod();
    Console.WriteLine(result);
}
```

Ở đây, SomeAsyncMethod là một phương thức bất đồng bộ trả về Task hoặc Task<T>. Phương thức sẽ tạm dừng tại await và tiếp tục sau khi tác vụ hoàn thành, và không cần sử dụng callback.

4. Tóm tắt các điểm chính:

- **Thread trong async và await**: Tác vụ bất đồng bộ có thể chạy trên luồng khác hoặc tiếp tục trên luồng chính, tùy thuộc vào ngữ cảnh và môi trường (UI hoặc console). Mặc định trong ứng dụng UI, tác vụ sẽ quay lại luồng UI sau khi hoàn thành.
- **So với lập trình song song**: Lập trình song song chạy nhiều tác vụ đồng thời trên nhiều luồng, trong khi lập trình bất đồng bộ không thực sự chạy đồng thời mà chỉ "chờ" tác vụ mà không chặn luồng. Các tác vụ bất đồng bộ chủ yếu liên quan đến I/O.
- **Callback**: async và await giúp giảm bớt sự phức tạp của callback, làm cho mã dễ hiểu hơn, thay vì phải gọi lại một phương thức khi tác vụ hoàn thành.

Việc quyết định xem tác vụ bất đồng bộ (async/await) sẽ thực thi trên luồng chính hay một luồng khác phụ thuộc vào **ngữ cảnh đồng bộ hóa (synchronization context)** của ứng dụng và cách bạn sử dụng các từ khóa await trong phương thức bất đồng bộ.

1. Ngữ cảnh đồng bộ hóa (Synchronization Context) và luồng chính:

Khi bạn sử dụng await trong phương thức async, hệ thống sẽ kiểm tra **ngữ cảnh đồng bộ hóa** để quyết định liệu tác vụ đó có nên tiếp tục trên cùng một luồng (ví dụ, luồng chính) hay không.

- **Trong ứng dụng UI (như WinForms, WPF, hoặc ASP.NET):** Khi bạn sử dụng await trong một phương thức bất đồng bộ, hệ thống sẽ giữ lại ngữ cảnh đồng bộ hóa và tiếp tục thực thi mã sau await trên cùng một luồng (thường là luồng UI hoặc luồng chính). Điều này là vì hệ thống muốn tránh việc thay đổi luồng khi tương tác với UI, vì hầu hết các thao tác UI cần phải thực hiện trên luồng UI.
- **Trong ứng dụng console hoặc nền (non-UI):** Nếu bạn gọi await trong phương thức bất đồng bộ và không có ngữ cảnh đồng bộ hóa đặc biệt (như trong ứng dụng console hoặc dịch vụ web), thì tác vụ có thể chạy trên bất kỳ luồng nào, tùy thuộc vào hệ thống. Khi await hoàn thành, chương trình có thể tiếp tục trên **một luồng khác**.

2. Tại sao luồng chính vẫn thực thi mà không bị gián đoạn khi chờ tác vụ bất đồng bộ?

Khi bạn gọi await trong một phương thức bất đồng bộ, bạn không chặn luồng chính hoặc luồng gọi (ví dụ, luồng UI hoặc luồng main). Thay vào đó:

- **Trong ứng dụng không phải UI (ví dụ, console hoặc dịch vụ web):** Phương thức async sẽ trả về một Task hoặc Task<T> ngay lập tức, cho phép luồng gọi tiếp tục thực thi. Khi tác vụ bất đồng bộ hoàn thành, mã sau await sẽ được tiếp tục. Tuy nhiên, luồng chính không bị chặn lại, vì tác vụ không yêu cầu chờ đợi đồng bộ (blocking) trong suốt quá trình thực thi.
- **Trong ứng dụng UI (WinForms, WPF, hoặc ASP.NET):** Khi bạn sử dụng await, mặc định, hệ thống sẽ đảm bảo rằng mã sau await tiếp tục chạy trên cùng một luồng (thường là luồng UI) sau khi tác vụ bất đồng bộ hoàn thành. Điều này là rất quan trọng đối với các thao tác với UI, vì bạn không thể cập nhật UI từ một luồng khác ngoài luồng chính. Tuy nhiên, nếu bạn muốn tránh việc quay lại luồng chính sau khi tác vụ hoàn thành, bạn có thể sử dụng ConfigureAwait(false) để yêu cầu tác vụ tiếp tục trên một luồng khác.

3. Ví dụ về sự khác biệt giữa UI và non-UI:

Ví dụ trong ứng dụng UI (WinForms/WPF):

```
using System;
using System.Threading.Tasks;
using System.Windows.Forms;

public class MainForm : Form
{
    private async void button_Click(object sender, EventArgs e)
    {
        // Gọi một tác vụ bất đồng bộ
        await Task.Delay(2000);
```

```
        // Sau khi Task.Delay hoàn thành, mã tiếp theo sẽ được thực thi trên
cùng một luồng (luồng UI)
        MessageBox.Show("Tác vụ đã hoàn thành!");
    }
}
```

Ở đây, phương thức button_Click là một phương thức bất đồng bộ. Khi gọi await Task.Delay(2000), hệ thống sẽ đảm bảo rằng sau khi tác vụ Task.Delay hoàn thành, mã tiếp theo (MessageBox.Show) sẽ được thực thi trên cùng một luồng UI (luồng chính).

Ví dụ trong ứng dụng Console:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        // Gọi một tác vụ bất đồng bộ
        await Task.Delay(2000);
        // Sau khi Task.Delay hoàn thành, mã tiếp theo có thể được thực thi trên
một luồng khác
        Console.WriteLine("Tác vụ đã hoàn thành!");
    }
}
```

Trong ứng dụng console, khi bạn gọi await Task.Delay(2000), luồng chính sẽ không bị chặn lại và có thể thực thi các công việc khác (nếu có). Sau khi tác vụ Task.Delay hoàn thành, chương trình sẽ tiếp tục thực thi mã sau await, có thể trên cùng một luồng hoặc một luồng khác, tùy thuộc vào ngữ cảnh đồng bộ hóa.

4. Cơ chế ConfigureAwait(false)

Khi bạn gọi await mà không muốn tiếp tục trên cùng một luồng (như luồng UI), bạn có thể sử dụng ConfigureAwait(false) để tránh việc quay lại ngữ cảnh đồng bộ hóa:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        // Đảm bảo rằng mã sau await không quay lại luồng chính
        await Task.Delay(2000).ConfigureAwait(false);
        Console.WriteLine("Tác vụ đã hoàn thành!");
    }
}
```

ConfigureAwait(false) đảm bảo rằng sau khi tác vụ hoàn thành, mã tiếp theo không phải thực thi trên cùng một luồng mà có thể chạy trên một luồng khác, giúp cải thiện hiệu suất trong một số trường hợp (như ứng dụng không có UI).

Tóm tắt:

- **Ai quyết định luồng thực thi**: Hệ thống quyết định luồng thực thi dựa trên ngữ cảnh đồng bộ hóa. Trong ứng dụng UI, mã sau await thường tiếp tục trên cùng một luồng (luồng chính). Trong ứng dụng console hoặc không có UI, mã có thể tiếp tục trên một luồng khác.

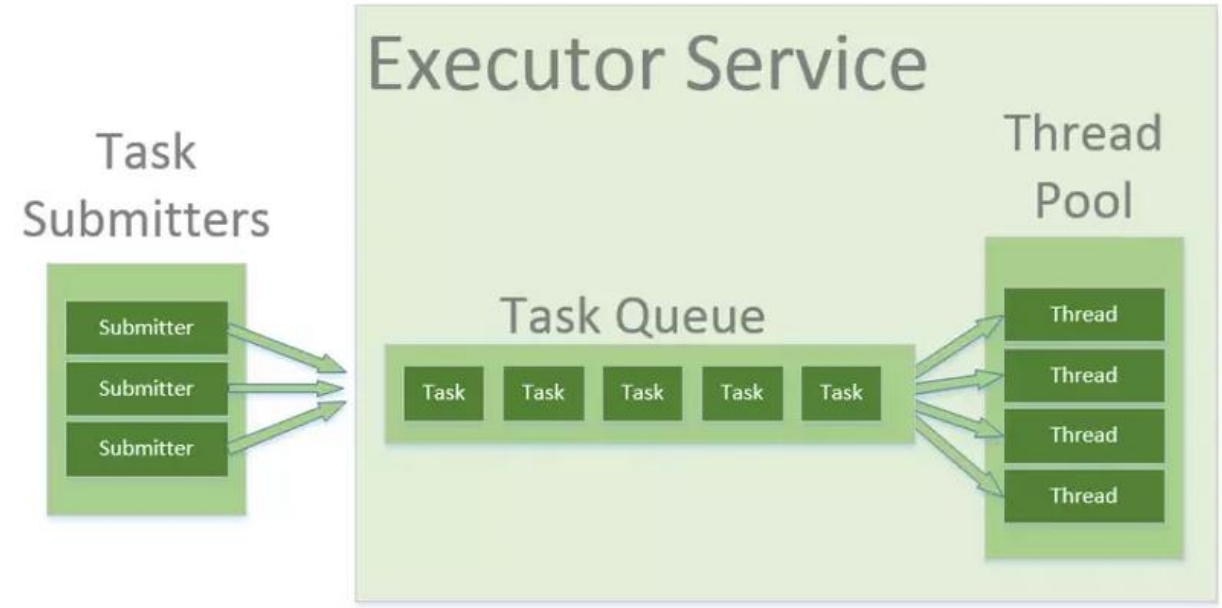
- **Tại sao luồng chính không bị gián đoạn**: Khi gọi await, chương trình không chặn luồng chính mà trả về một Task. Điều này cho phép luồng chính tiếp tục xử lý các công việc khác cho đến khi tác vụ bất đồng bộ hoàn thành.

Khi luồng chính không thực thi một Task (tác vụ bất đồng bộ) trong C#, có một cơ chế đặc biệt đảm nhiệm việc thực thi tác vụ đó, gọi là **Thread Pool** hoặc một luồng khác tùy thuộc vào ngữ cảnh. Dưới đây là cách thức hoạt động chi tiết:

1. Thread Pool:

Trong C#, khi bạn gọi một tác vụ bất đồng bộ (ví dụ: Task.Run(), Task.Delay(), hay các phương thức bất đồng bộ như HttpClient.GetAsync()), tác vụ đó thường không được thực thi ngay trên luồng chính mà sẽ được gán cho một **luồng khác**, thường là một luồng trong **Thread Pool**.

Thread Pool là một cơ chế của .NET để quản lý và tái sử dụng các luồng, nhằm tránh việc tạo và hủy luồng liên tục, giúp tiết kiệm tài nguyên và giảm chi phí khi thực thi các tác vụ. Khi có một tác vụ bất đồng bộ cần thực thi, hệ thống sẽ mượn một luồng từ Thread Pool, thực thi tác vụ, và sau khi hoàn thành, luồng đó sẽ được trả lại cho Thread Pool để tái sử dụng.



2. Cơ chế thực thi của Task:

Khi bạn gọi một tác vụ bất đồng bộ (ví dụ, await Task.Delay(2000)), sau đây là những bước chính:

1. **Tạo Task**: Tác vụ bất đồng bộ (Task) sẽ được tạo và đưa vào hàng đợi.
2. **Thread Pool thực thi**: Task này sẽ được thực thi bởi một luồng từ **Thread Pool** (hoặc một luồng khác nếu có yêu cầu đặc biệt).
3. **Không chặn luồng chính**: Trong suốt thời gian tác vụ bất đồng bộ đang thực thi, **luồng chính không bị chặn**. Nó có thể tiếp tục thực thi các công việc khác.

4. **Task hoàn thành**: Sau khi tác vụ hoàn thành (chẳng hạn như khi hết thời gian trong Task.Delay), mã tiếp theo trong phương thức async (sau await) sẽ được tiếp tục thực thi. Tùy thuộc vào ngữ cảnh đồng bộ hóa:
    - Trong ứng dụng UI (như WinForms, WPF), mã tiếp theo có thể tiếp tục trên **luồng UI**.
    - Trong ứng dụng console, mã có thể tiếp tục trên **luồng khác** nếu không sử dụng ConfigureAwait(false).

3. Trường hợp đặc biệt: Cơ chế await và luồng chính:

Khi sử dụng await, hệ thống sẽ lưu trữ một "tiến trình tiếp tục" (continuation) cho tác vụ bất đồng bộ. Điều này có nghĩa là, khi tác vụ đó hoàn thành, hệ thống sẽ tiếp tục thực thi đoạn mã tiếp theo trong phương thức bất đồng bộ. Nếu đang trong một ứng dụng UI (như WPF hay WinForms), mặc định hệ thống sẽ quay lại luồng chính (UI thread) sau khi tác vụ bất đồng bộ hoàn thành.

4. Ví dụ minh họa:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Trước khi await");

        // Task.Delay sẽ chạy trên một luồng khác từ Thread Pool
        await Task.Delay(2000); // Giả sử đây là tác vụ bất đồng bộ.

        Console.WriteLine("Sau khi await");
    }
}
```

- **Bước 1:** "Trước khi await" được in ra trên luồng chính.
- **Bước 2:** Khi gọi await Task.Delay(2000), tác vụ sẽ được đưa vào hàng đợi của **Thread Pool**, và một luồng từ **Thread Pool** sẽ thực hiện tác vụ này.
- **Bước 3:** Trong 2 giây chờ đợi, **luồng chính không bị chặn** mà có thể thực hiện các công việc khác.
- **Bước 4:** Sau khi tác vụ Task.Delay hoàn thành, hệ thống sẽ tiếp tục thực thi mã sau await, và nếu không có ngữ cảnh đồng bộ hóa, mã này có thể được thực thi trên **luồng chính** (hoặc một luồng khác trong môi trường không phải UI).

5. Tóm tắt cơ chế:

- **Task** được thực thi trên một luồng trong **Thread Pool** hoặc một luồng khác được hệ thống chỉ định.
- **Luồng chính không bị chặn** khi thực thi tác vụ bất đồng bộ. Khi await được gọi, luồng chính có thể tiếp tục làm việc hoặc xử lý các sự kiện khác (trong môi trường UI, ví dụ).

- Khi tác vụ hoàn thành, mã tiếp theo trong phương thức async sẽ tiếp tục chạy, và luồng thực thi sẽ phụ thuộc vào ngữ cảnh đồng bộ hóa (UI thread trong ứng dụng UI, hoặc bất kỳ luồng nào trong ứng dụng console).

Bài 41: Sử dụng tasks cho các hoạt động bất đồng bộ (asynchronous operations)

Mục tiêu bài học

- Hiểu được khái niệm **bất đồng bộ** (asynchronous) trong C#.
- Làm quen với lớp Task trong C#.
- Ứng dụng Task để viết mã tối ưu hơn, đặc biệt khi xử lý các tác vụ nặng hoặc chờ đợi I/O.

---

1. Tại sao cần bất đồng bộ?

- Trong lập trình đồng bộ, một tác vụ chậm (ví dụ: truy cập cơ sở dữ liệu hoặc đọc tệp) có thể khiến toàn bộ ứng dụng bị treo.
- Bất đồng bộ giúp cải thiện hiệu suất và trải nghiệm người dùng, đặc biệt là trong ứng dụng web, ứng dụng desktop hoặc xử lý dữ liệu lớn.

---

2. Giới thiệu về Task

- Task là một trong những cách chính để thực hiện các tác vụ bất đồng bộ trong C#.
- Một Task đại diện cho một công việc (work) sẽ được thực hiện hoặc đã hoàn thành.

Đặc điểm của Task:

- Hỗ trợ theo dõi trạng thái (Pending, Running, Completed, Faulted...).
- Có thể trả về kết quả (Task<T>).
- Cho phép lập trình dễ dàng hơn với các từ khóa async và await.

---

3. Cách sử dụng Task trong C#

3.1. Tạo và chạy một Task

```
using System;
using System.Threading.Tasks;

class Program
```

```
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Bắt đầu thực hiện...");

        // Gọi hàm bất đồng bộ
        await PerformAsyncTask();

        Console.WriteLine("Hoàn tất!");
    }

    static async Task PerformAsyncTask()
    {
        await Task.Run(() =>
        {
            // Mô phỏng tác vụ nặng
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine($"Đang xử lý {i + 1}/5...");
                Task.Delay(1000).Wait(); // Chờ 1 giây
            }
        });
    }
}
```

Kết quả:

- Mỗi giây, chương trình in ra một dòng thông báo.
- Trong khi đó, ứng dụng không bị treo và có thể thực hiện các công việc khác.

3.2. Sử dụng Task<T> để trả về kết quả

```
static async Task<int> CalculateSumAsync(int a, int b)
{
    return await Task.Run(() =>
    {
        Task.Delay(2000).Wait(); // Mô phỏng độ trễ
        return a + b;
    });
}

static async Task Main(string[] args)
{
    Console.WriteLine("Tính toán đang thực hiện...");
    int result = await CalculateSumAsync(10, 20);
    Console.WriteLine($"Kết quả: {result}");
}
```

Kết quả:

- Sau 2 giây, chương trình trả về tổng của hai số.

---

4. Các từ khóa async và await

4.1. async

- Được sử dụng khi định nghĩa một phương thức bất đồng bộ.
- Một phương thức async sẽ trả về:
    - Task hoặc Task<T> cho các tác vụ bất đồng bộ.
    - void cho các tác vụ không cần chờ kết quả (không khuyến nghị).

4.2. await

- Tạm dừng việc thực thi phương thức hiện tại cho đến khi Task hoàn thành.
- Giúp mã dễ đọc hơn, thay thế callback phức tạp.

Ví dụ:

```
static async Task DownloadFileAsync(string url)
{
    Console.WriteLine($"Bắt đầu tải {url}...");
    await Task.Delay(3000); // Giả lập tải file
    Console.WriteLine($"Hoàn tất tải {url}");
}
```

---

5. Lưu ý khi sử dụng Task

1. **Tránh sử dụng .Wait() hoặc .Result trên Task**:
    - Có thể gây **deadlock** trong một số trường hợp.
    - Ưu tiên sử dụng await để giữ ứng dụng bất đồng bộ.
2. **Xử lý lỗi trong Task**:
    - Sử dụng try-catch để bắt lỗi:

```
try
{
    await Task.Run(() => throw new Exception("Lỗi xảy ra!"));
}
catch (Exception ex)
{
    Console.WriteLine($"Đã bắt lỗi: {ex.Message}");
}
```

3. **Quản lý tài nguyên trong các tác vụ song song**:
    - Sử dụng Task.WhenAll hoặc Task.WhenAny để đồng bộ nhiều Task.

---

6. Kết hợp Task trong thực tế

6.1. Gọi API bất đồng bộ

```
using System.Net.Http;

static async Task GetApiDataAsync()
```

```
{
    using HttpClient client = new HttpClient();
    string url = "https://jsonplaceholder.typicode.com/posts/1";
    string result = await client.GetStringAsync(url);

    Console.WriteLine($"Dữ liệu nhận được: {result}");
}
```

---

7. Tổng kết

- Task và các từ khóa async/await là công cụ mạnh mẽ giúp tối ưu hóa các ứng dụng C#.
- Việc áp dụng bất đồng bộ sẽ cải thiện hiệu suất và khả năng mở rộng của ứng dụng.
- Khi sử dụng Task, hãy lưu ý xử lý lỗi và tránh lạm dụng các phương pháp đồng bộ hóa (blocking).

Bài tập thực hành

1. Viết chương trình sử dụng Task để tải đồng thời 3 tệp từ các URL khác nhau.
2. Viết một hàm Task<T> tính toán số Fibonacci thứ n bất đồng bộ.
3. Kết hợp Task và HttpClient để gửi yêu cầu POST đến một API và xử lý phản hồi.

Bài 42: Best practices for handling asynchronous code

1. Giới thiệu về bất đồng bộ (asynchronous programming)

- **Asynchronous programming là gì?**
    - Giúp cải thiện hiệu năng và khả năng phản hồi (responsiveness) của ứng dụng.
    - Giảm thiểu thời gian chờ trong các tác vụ I/O (đọc/ghi file, gọi API, truy vấn database).
- **Các khái niệm cơ bản:**
    - async và await: Cặp từ khóa chính trong lập trình bất đồng bộ.
    - Task và Task<T>: Đại diện cho một thao tác bất đồng bộ.
    - ValueTask: Tối ưu hiệu suất trong một số trường hợp.

---

2. Best Practices cho lập trình bất đồng bộ

2.1. Sử dụng async/await đúng cách

- **Tránh sử dụng async void:**
    - Chỉ dùng cho event handlers.

- Các hàm bất đồng bộ nên trả về Task hoặc Task<T> để quản lý trạng thái.
- Ví dụ sai:

```
public async void ProcessData() // Sai
{
    await Task.Delay(1000);
}
```

- Ví dụ đúng:

```
public async Task ProcessData() // Đúng
{
    await Task.Delay(1000);
}
```

2.2. Không "block" thread với .Result hoặc .Wait()

- Điều này có thể dẫn đến **deadlock** hoặc giảm hiệu năng.
- **Không nên:**

```
var result = SomeAsyncMethod().Result; // Sai
```

- **Thay vào đó:**

```
var result = await SomeAsyncMethod(); // Đúng
```

2.3. Sử dụng ConfigureAwait(false) trong thư viện

- Giúp tránh capture SynchronizationContext, cải thiện hiệu năng.
- Dùng trong code không phụ thuộc UI.
- Ví dụ:

```
await SomeTask.ConfigureAwait(false);
```

2.4. Tránh tạo quá nhiều Task không cần thiết

- Đừng bọc code đồng bộ bằng Task.Run() trừ khi thực sự cần thiết.
- **Sai:**

```
Task.Run(() => HeavyComputation()); // Không cần thiết
```

- **Đúng:**

```
HeavyComputation(); // Gọi trực tiếp nếu không liên quan I/O
```

2.5. Quản lý ngoại lệ bất đồng bộ với try-catch

- Đảm bảo bắt ngoại lệ xảy ra trong các hàm bất đồng bộ.
- Ví dụ:

```
try
{
    await SomeAsyncMethod();
}
catch (Exception ex)
```

```
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

---

3. Xử lý song song với Task.WhenAll và Task.WhenAny

- **Task.WhenAll:** Chờ tất cả các tác vụ hoàn thành.

```
var tasks = new[] { Task1(), Task2(), Task3() };
await Task.WhenAll(tasks);
```

- **Task.WhenAny:** Chờ bất kỳ một tác vụ hoàn thành.

```
var firstCompleted = await Task.WhenAny(Task1(), Task2(), Task3());
```

---

4. Hủy bỏ với CancellationToken

- Dùng CancellationToken để cho phép dừng tác vụ bất đồng bộ.
- Ví dụ:

```
public async Task ProcessData(CancellationToken token)
{
    for (int i = 0; i < 10; i++)
    {
        token.ThrowIfCancellationRequested();
        await Task.Delay(1000, token); // Hỗ trợ hủy
    }
}
```

---

5. Sử dụng công cụ hỗ trợ kiểm tra hiệu năng và lỗi

- **Công cụ:**
    - Visual Studio Profiler.
    - SonarQube (phát hiện lỗi tiềm ẩn).
    - AsyncAnalyzer: Phân tích bất đồng bộ trong mã nguồn.

---

6. Lời khuyên thực tế

- Hãy dùng bất đồng bộ một cách có chủ đích: chỉ áp dụng khi thực sự cần thiết.
- Code bất đồng bộ dễ dẫn đến lỗi hơn code đồng bộ, hãy viết unit tests kỹ càng.

---

7. Bài tập thực hành

1. Viết một ứng dụng console sử dụng async/await để tải dữ liệu từ một API.
2. Thêm tính năng hủy bỏ (CancellationToken) khi gọi API.

3. Tối ưu hóa ứng dụng với ConfigureAwait(false).

Ví dụ cụ thể về CancellationToken và ConfigureAwait

1. Ví dụ về CancellationToken

CancellationToken được sử dụng để cho phép hủy bỏ một tác vụ bất đồng bộ. Trong ví dụ này, chúng ta sẽ có một ứng dụng console chạy một tác vụ đếm số, và bạn có thể dừng tác vụ bằng cách nhấn một phím.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        var cts = new CancellationTokenSource();

        Console.WriteLine("Nhấn bất kỳ phím nào để dừng...");

        // Bắt đầu một Task đếm số
        var countingTask = CountAsync(cts.Token);

        // Chờ người dùng nhấn phím để hủy bỏ
        Console.ReadKey();
        cts.Cancel();

        try
        {
            await countingTask;
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("Đếm bị hủy.");
        }
    }

    static async Task CountAsync(CancellationToken token)
    {
        for (int i = 0; i < 100; i++)
        {
            token.ThrowIfCancellationRequested(); // Kiểm tra yêu cầu hủy bỏ

            Console.WriteLine($"Đang đếm: {i}");
            await Task.Delay(500, token); // Đợi 500ms và hỗ trợ hủy
        }

        Console.WriteLine("Hoàn thành đếm.");
    }
}
```

Giải thích:

- **CancellationTokenSource:** Được sử dụng để phát tín hiệu hủy bỏ.

- **token.ThrowIfCancellationRequested():** Kiểm tra xem có yêu cầu hủy nào chưa, nếu có thì ném ngoại lệ OperationCanceledException.
- **Task.Delay(500, token):** Chờ 500ms và hỗ trợ hủy bỏ trong khi chờ.

---

2. Ví dụ về ConfigureAwait(false)

ConfigureAwait(false) được dùng trong các thư viện hoặc ứng dụng không cần sử dụng SynchronizationContext (thường trong các ứng dụng không liên quan UI, như console hoặc web API).

Trường hợp không sử dụng ConfigureAwait:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Bắt đầu Main");
        await DoWorkAsync();
        Console.WriteLine("Kết thúc Main");
    }

    static async Task DoWorkAsync()
    {
        Console.WriteLine("Bắt đầu DoWork");
        await Task.Delay(1000); // Chờ 1 giây
        Console.WriteLine("Kết thúc DoWork");
    }
}
```

Khi chạy trên môi trường có SynchronizationContext, như ứng dụng WPF hoặc WinForms, nó sẽ "capture" ngữ cảnh đồng bộ để tiếp tục công việc sau await.

Sử dụng ConfigureAwait(false) trong thư viện:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine("Bắt đầu Main");
        await DoWorkAsync();
        Console.WriteLine("Kết thúc Main");
    }

    static async Task DoWorkAsync()
    {
        Console.WriteLine("Bắt đầu DoWork");
        await Task.Delay(1000).ConfigureAwait(false); // Không capture
SynchronizationContext
        Console.WriteLine("Kết thúc DoWork");
    }
```

}

Khác biệt:

- Khi dùng ConfigureAwait(false), mã sau await sẽ chạy trên thread pool, không "capture" lại ngữ cảnh ban đầu. Điều này giúp cải thiện hiệu năng và tránh deadlock trong các môi trường không liên quan đến giao diện người dùng (UI).

So sánh tình huống sử dụng

- **Dùng CancellationToken:** Khi bạn cần khả năng hủy bỏ một tác vụ bất đồng bộ.
- **Dùng ConfigureAwait(false):** Trong thư viện hoặc các ứng dụng không cần quản lý ngữ cảnh đồng bộ, ví dụ như backend services, APIs, hoặc các công việc nặng không phụ thuộc vào UI.

Bài 43: Bài tập tổng hợp C# căn bản

1. Làm việc với biến và kiểu dữ liệu

Viết chương trình nhập vào thông tin của một sinh viên bao gồm:

- Tên (chuỗi)
- Tuổi (số nguyên)
- Điểm trung bình (số thực)

Sau đó, in ra thông tin vừa nhập theo định dạng:
Tên: [Tên], Tuổi: [Tuổi], Điểm trung bình: [Điểm trung bình]

---

2. Kiểm tra số chẵn/lẻ

Nhập vào một số nguyên từ bàn phím và kiểm tra xem số đó là số chẵn hay lẻ. In ra màn hình kết quả.

---

3. Tính giai thừa

Viết chương trình tính giai thừa của một số nguyên dương n nhập vào từ bàn phím.
Công thức giai thừa:
n!=1×2×3×...×nn! = 1 \times 2 \times 3 \times ... \times nn!=1×2×3×...×n

---

4. Tìm số nguyên tố

Viết chương trình nhập vào một số nguyên n và kiểm tra xem nó có phải là số nguyên tố hay không.
Số nguyên tố là số lớn hơn 1 và chỉ chia hết cho 1 và chính nó.

---

5. Làm việc với mảng

Nhập vào một mảng gồm n số nguyên từ bàn phím (với n do người dùng nhập). Thực hiện các thao tác:

- In ra mảng vừa nhập.
- Tính tổng các phần tử trong mảng.
- Tìm phần tử lớn nhất và nhỏ nhất trong mảng.

---

6. Tính toán với chuỗi

Nhập vào một chuỗi bất kỳ và thực hiện:

- Đếm số ký tự trong chuỗi.
- Đếm số ký tự là chữ cái.
- Đảo ngược chuỗi và in ra kết quả.

---

7. Hàm kiểm tra năm nhuận

Viết một hàm bool KiemTraNamNhuan(int nam) để kiểm tra một năm có phải năm nhuận không.
Gợi ý: Năm nhuận là năm chia hết cho 4 nhưng không chia hết cho 100, hoặc chia hết cho 400.

---

8. Xây dựng lớp SinhVien

Tạo một lớp SinhVien với các thuộc tính:

- Mã sinh viên
- Tên
- Tuổi
- Điểm trung bình

Thực hiện các thao tác:

- Khởi tạo danh sách 3 sinh viên.
- In thông tin của tất cả sinh viên.
- Tìm sinh viên có điểm trung bình cao nhất.

---

9. Vẽ tam giác Pascal

Viết chương trình hiển thị tam giác Pascal với số dòng n nhập từ bàn phím.

Ví dụ với n = 5:

markdown

Sao chép mã

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

---

10. Quản lý danh sách sản phẩm

Tạo một lớp SanPham với các thuộc tính:

- Mã sản phẩm
- Tên sản phẩm
- Giá sản phẩm

Viết chương trình:

- Thêm sản phẩm vào danh sách.
- Hiển thị danh sách sản phẩm.
- Tìm kiếm sản phẩm theo tên.
- Sắp xếp sản phẩm theo giá tăng dần.

Bài 44: Giải thích độ phức tạp của các thuật toán

Độ phức tạp của giải thuật là một cách để đánh giá hiệu quả của một giải thuật dựa trên hai yếu tố chính:

1. **Độ phức tạp thời gian**: Là lượng thời gian cần thiết để giải thuật thực thi, thường được biểu diễn dưới dạng số lần thực hiện các phép toán cơ bản.
2. **Độ phức tạp không gian**: Là lượng bộ nhớ cần thiết để giải thuật hoạt động, bao gồm cả bộ nhớ dành cho biến, cấu trúc dữ liệu và ngăn xếp lời gọi hàm.

Độ phức tạp thường được biểu diễn bằng ký hiệu Big-O (O-notation), miêu tả **tốc độ tăng trưởng** của thời gian hoặc không gian khi kích thước đầu vào tăng lên.

Các bước tính độ phức tạp của giải thuật

1. **Xác định kích thước đầu vào**:
    - Xác định biến hoặc tham số đầu vào ảnh hưởng đến thời gian chạy (thường được ký hiệu là n).
2. **Xác định số lần thực hiện các phép toán cơ bản**:
    - Đếm số lần lặp, so sánh, tính toán, hoặc truy cập dữ liệu cần thiết trong giải thuật.
3. **Loại bỏ các hằng số và hệ số nhỏ**:
    - Trong Big-O, chúng ta chỉ quan tâm đến tốc độ tăng trưởng, vì vậy loại bỏ các hằng số hoặc các bậc thấp hơn.
    - Ví dụ: nếu số phép tính là 3n2+5n+23, ta chỉ quan tâm đến bậc cao nhất 2n^2, vậy độ phức tạp là O(n2).
4. **Tìm trường hợp xấu nhất (Worst Case)**:
    - Đánh giá giải thuật trong trường hợp mất nhiều thời gian nhất (tình huống bất lợi nhất).

Một số ví dụ

1. **Giải thuật duyệt mảng**:

```
for (int i = 0; i < n; i++)
{
    Console.WriteLine(arr[i]);
}
```

    - Có một vòng lặp chạy từ 0 đến n−1, vậy độ phức tạp thời gian là O(n).
2. **Giải thuật sắp xếp Bubble Sort**:

```
for (int i = 0; i < n - 1; i++)
{
    for (int j = 0; j < n - i - 1; j++)
    {
        if (arr[j] > arr[j + 1])
        {
            // Hoán đổi
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

    - Có hai vòng lặp lồng nhau, mỗi vòng chạy tối đa n lần, nên tổng thời gian là O(n^2).
3. **Tìm kiếm nhị phân**:

```
while (low <= high)
```

```
{
    int mid = (low + high) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
}
```

- Ở mỗi bước, dải tìm kiếm giảm đi một nửa, vậy độ phức tạp là O(log n).

Một số mức độ phức tạp thường gặp

1. **O(1)**: Thời gian không phụ thuộc vào kích thước đầu vào (ví dụ: truy cập phần tử trong mảng).
2. **O(log n)**: Tăng trưởng logarithmic, thường gặp trong các giải thuật chia để trị (như tìm kiếm nhị phân).
3. **O(n)**: Tăng trưởng tuyến tính, phổ biến khi duyệt qua tất cả phần tử của đầu vào.
4. **O(n log n)**: Thường gặp trong các giải thuật sắp xếp hiệu quả như Merge Sort hoặc Quick Sort.
5. **O(n^2)**: Tăng trưởng bậc hai, phổ biến trong các giải thuật sử dụng hai vòng lặp lồng nhau (như Bubble Sort).
6. **O(2^n)** hoặc **O(n!)**: Tăng trưởng rất nhanh, thường gặp trong các bài toán đệ quy hoặc tổ hợp phức tạp.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Bài 45: Kết thúc khóa học và giới thiệu khóa học Lập trình C# nâng cao

Chào mừng các bạn đến với khóa học Lập trình C# Nâng cao!

Link khóa học: https://tedu.com.vn/khoa-hoc/c-mastery-from-professional-to-advanced-developer-58.html

Sau khi hoàn thành khóa học này, bạn sẽ nắm vững những kiến thức cơ bản và nâng cao về C#, từ đó có thể phát triển các ứng dụng mạnh mẽ và tối ưu hơn. Tuy nhiên, đây chỉ là sự khởi đầu! Khóa học Lập trình C# Nâng cao sẽ giúp bạn đi sâu vào các chủ đề quan trọng mà lập trình viên cần hiểu rõ để phát triển phần mềm chất lượng cao.

Nội dung khóa học Lập trình C# Nâng cao:

1. **Lập trình hướng đối tượng (OOP)**:
    - Tìm hiểu sâu hơn về các nguyên lý OOP như kế thừa, đa hình, và đóng gói.
    - Áp dụng các kỹ thuật thiết kế phần mềm như SOLID principles.
2. **Xử lý bất đồng bộ và đa luồng (Asynchronous and Multithreading)**:
    - Sử dụng async và await để tối ưu hóa hiệu suất.
    - Quản lý đa luồng trong các ứng dụng thực tế.
3. **Lập trình với các API và Frameworks phổ biến**:
    - Làm việc với ASP.NET Core, Entity Framework Core, và các công nghệ liên quan.
    - Tạo API RESTful và ứng dụng Web mạnh mẽ.
4. **Kiến thức về mô hình thiết kế (Design Patterns)**:
    - Tìm hiểu về các mẫu thiết kế phổ biến như Singleton, Factory, Observer, v.v.
5. **Xử lý dữ liệu và cơ sở dữ liệu**:
    - Làm việc với các cơ sở dữ liệu SQL và NoSQL.
    - Tối ưu hóa truy vấn và thao tác với dữ liệu lớn.
6. **Kiểm thử (Testing)**:
    - Tìm hiểu về các phương pháp kiểm thử tự động với xUnit, NUnit và Moq.

Lý do bạn nên tham gia khóa học Lập trình C# Nâng cao:

- **Nâng cao kỹ năng lập trình**: Bạn sẽ học được các kỹ thuật và công cụ giúp nâng cao hiệu quả và tối ưu hóa mã nguồn.
- **Ứng dụng thực tế**: Các bài học được thiết kế với mục tiêu giúp bạn áp dụng lý thuyết vào thực tế công việc.

Khóa học phù hợp cho ai?

- Các lập trình viên C# đã có kinh nghiệm cơ bản và muốn nâng cao kỹ năng.

- Các bạn muốn tìm hiểu cách tối ưu mã nguồn, xây dựng ứng dụng phức tạp và làm việc với các công nghệ mới nhất.