1. Các kiểu dữ liệu cơ bản:

Int: kiểu số nguyên, 32 bit

Sbyte: kiểu số nguyên 8 bit

Byte kiểu số nguyên 8 bit từ 0 đến 255

Short; có dấu, 16 bit

Ushot, không dấu 16 bit

….

Float 32 bit biểu diễn chính xác số thực

Double, số thực chấm động 64 bot

Char, 1 kí tự, 16 bit

Bool kiểu logic

Có dấu: tức là biểu diễn được cả số nguyên và số âm

Không dấu: chỉ biểu diễn được số dương

Có dấu chấm thực: Biểu diễn độ chỉnh xác của số thực

* 1. Mảng trong C#
* Mảng là một cấu trúc dữ liệu được sử dụng để lưu trũ một tập dữ liệu cùng kiểu
* Khởi tạo mảng: new datatype[n] trong đó số phần tử là n và có kiểu dữ liệu là data type
* Các phần tử trong mảng có index trải từ 0 đến độ dài mảng -1
  1. Toán tử trong C#

1.2.1 Toán tử số học

Dùng để thực hiện các phép toán số học như cộng, trừ, nhân, chia và lấy phần dư.

* + : Cộng. Ví dụ: a + b
* - : Trừ. Ví dụ: a - b
* \* : Nhân. Ví dụ: a \* b
* / : Chia. Ví dụ: a / b
* % : Phép chia lấy dư. Ví dụ: a % b

### 1.2.2 Toán tử gán (Assignment Operators)

Dùng để gán giá trị cho biến.

* = : Gán giá trị. Ví dụ: a = 5
* += : Cộng và gán. Ví dụ: a += 3
* -= : Trừ và gán. Ví dụ: a -= 3
* \*= : Nhân và gán. Ví dụ: a \*= 3
* /= : Chia và gán. Ví dụ: a /= 3
* %= : Chia lấy dư và gán. Ví dụ: a %= 3

### 1.2.3. Toán tử quan hệ (Relational Operators)

Dùng để so sánh các giá trị.

* == : Bằng. Ví dụ: a == b
* != : Khác. Ví dụ: a != b
* > : Lớn hơn. Ví dụ: a > b
* < : Nhỏ hơn. Ví dụ: a < b
* >= : Lớn hơn hoặc bằng. Ví dụ: a >= b
* <= : Nhỏ hơn hoặc bằng. Ví dụ: a <= b

### 1.2.4. Toán tử logic (Logical Operators)

Dùng để thực hiện các phép toán logic.

* && : Phép AND logic. Ví dụ: a && b
* || : Phép OR logic. Ví dụ: a || b
* ! : Phép NOT logic. Ví dụ: !a

### 1.2.5. Toán tử đơn hạng (Unary Operators)

Áp dụng cho một toán hạng duy nhất.

* + : Giá trị dương. Ví dụ: +a
* - : Giá trị âm. Ví dụ: -a
* ++ : Tăng 1 đơn vị. Ví dụ: a++ hoặc ++a
* -- : Giảm 1 đơn vị. Ví dụ: a-- hoặc --a
* ! : Phủ định logic. Ví dụ: !a

### 1.2.6. Toán tử bit (Bitwise Operators)

Dùng để thực hiện các phép toán trên mức bit.

* & : Phép AND bit. Ví dụ: a & b
* | : Phép OR bit. Ví dụ: a | b
* ^ : Phép XOR bit. Ví dụ: a ^ b
* ~ : Phép NOT bit. Ví dụ: ~a
* << : Dịch bit sang trái. Ví dụ: a << 2
* >> : Dịch bit sang phải. Ví dụ: a >> 2

### 1.2.7. Toán tử điều kiện (Conditional Operator)

Toán tử ba ngôi (?:) là một dạng toán tử điều kiện.

* ?: : Toán tử ba ngôi (Ternary). Ví dụ: condition ? value1 : value2
  1. Cấu trúc rẽ nhánh trong C#
     1. Cấu trúc if, else, if else

Các nhánh của điều kiện xảy ra khi điều kiện của nhsnh đó đúng

* + 1. Cấu trúc switch

Cấu trúc switch dùng để kiểm tra giá trị của một biến và thực hiện các khối lệnh tương ứng với giá trị đó.

* + 1. Cấu trúc vòng lặp: For, Do While, While

2.

### 1. Khái niệm về String và StringBuilder

* **String**:
  + String là một đối tượng bất biến (immutable) trong C#. Điều này có nghĩa là mỗi khi bạn thay đổi giá trị của một chuỗi, một đối tượng mới được tạo ra trong bộ nhớ và giá trị cũ bị bỏ đi.
  + String phù hợp cho các thao tác với chuỗi có ít thay đổi, vì việc thay đổi liên tục sẽ tốn tài nguyên và hiệu suất.
* **StringBuilder**:
  + StringBuilder là một đối tượng có thể thay đổi (mutable). Khi bạn thay đổi giá trị của StringBuilder, nó sẽ thay đổi trên cùng một đối tượng mà không tạo ra đối tượng mới.
  + StringBuilder phù hợp cho các trường hợp cần thực hiện nhiều thao tác nối chuỗi hoặc thay đổi chuỗi thường xuyên, vì nó tiết kiệm bộ nhớ và cải thiện hiệu suất.

### 2. Chuyển đổi giữa String và StringBuilder

* **Chuyển đổi từ String sang StringBuilder**:

string str = "Hello, World!";

StringBuilder sb = new StringBuilder(str);

* **Chuyển đổi từ StringBuilder sang String**:

Sao chép mã

StringBuilder sb = new StringBuilder("Hello, World!");

string str = sb.ToString();

3. Ép kiểu

## Ép kiểu (Casting) trong C#

Ép kiểu trong C# là quá trình chuyển đổi một biến từ kiểu dữ liệu này sang kiểu dữ liệu khác. Có hai loại ép kiểu chính: **Ép kiểu dữ liệu** và **Ép kiểu đối tượng**.

### 1. Ép kiểu dữ liệu

#### a. Ép kiểu ngầm định (Implicit Casting)

* **Miêu tả**: Xảy ra tự động khi chuyển đổi từ kiểu dữ liệu nhỏ hơn sang kiểu dữ liệu lớn hơn. Ví dụ: từ int sang long, float sang double.
* **Ví dụ:**

csharp

Sao chép mã

int a = 10;

double b = a; // Ép kiểu ngầm định từ int sang double

Console.WriteLine(b); // Kết quả: 10.0

#### b. Ép kiểu tường minh (Explicit Casting)

* **Miêu tả**: Cần được thực hiện thủ công khi chuyển đổi từ kiểu dữ liệu lớn hơn sang kiểu dữ liệu nhỏ hơn. Điều này có thể dẫn đến mất dữ liệu nếu không cẩn thận.
* **Ví dụ:**

csharp

Sao chép mã

double a = 10.5;

int b = (int)a; // Ép kiểu tường minh từ double sang int

Console.WriteLine(b); // Kết quả: 10

#### c. Ép kiểu bằng phương thức Convert

* **Miêu tả**: Sử dụng các phương thức trong lớp Convert để ép kiểu giữa các kiểu dữ liệu khác nhau.
* **Ví dụ:**

csharp

Sao chép mã

string str = "123";

int num = Convert.ToInt32(str); // Ép kiểu từ string sang int

Console.WriteLine(num); // Kết quả: 123

### 2. Ép kiểu đối tượng

#### a. Upcasting

* **Miêu tả**: Ép kiểu từ lớp con lên lớp cha. Đây là một ép kiểu ngầm định và an toàn.
* **Ví dụ:**

csharp

Sao chép mã

class Animal { }

class Dog : Animal { }

Animal a = new Dog(); // Upcasting từ Dog lên Animal

#### b. Downcasting

* **Miêu tả**: Ép kiểu từ lớp cha xuống lớp con. Đây là một ép kiểu tường minh và không an toàn nếu không kiểm tra trước.
* **Ví dụ:**

csharp

Sao chép mã

Animal a = new Dog(); // Upcasting

Dog d = (Dog)a; // Downcasting từ Animal xuống Dog

* **Sử dụng từ khóa is và as để kiểm tra và thực hiện ép kiểu an toàn**:

csharp

Sao chép mã

Animal a = new Animal();

// Sử dụng 'is' để kiểm tra kiểu đối tượng

if (a is Dog)

{

Dog d = (Dog)a;

Console.WriteLine("Đối tượng được ép kiểu xuống Dog thành công.");

}

// Sử dụng 'as' để ép kiểu an toàn

Dog d2 = a as Dog;

if (d2 != null)

{

Console.WriteLine("Ép kiểu bằng 'as' thành công.");

}

else

{

Console.WriteLine("Ép kiểu bằng 'as' thất bại.");

}

4/ Constructor trong C#

### 1. Khái niệm về Constructor

* **Constructor** (hàm khởi tạo) là một phương thức đặc biệt trong lớp được gọi tự động khi một đối tượng của lớp đó được tạo ra. Mục đích của constructor là để khởi tạo các giá trị ban đầu cho các biến thành viên (thuộc tính) của lớp.
* **Đặc điểm của Constructor**:
  + Có tên trùng với tên của lớp.
  + Không có kiểu trả về, kể cả void.
  + Có thể có các tham số hoặc không có tham số.

Ngoài dùng constructor để khởi tạo, có thể dùng factory method để khởi tạo

public static Person CreatePerson(string name, int age) { return new Person(name, age);

## Access Modifiers trong C#

Các **Access Modifiers** (từ khóa chỉ định quyền truy cập) trong C# được sử dụng để thiết lập phạm vi và quyền truy cập của các lớp, phương thức, thuộc tính và biến trong chương trình. Dưới đây là các từ khóa truy cập phổ biến:

### 1. private

* **Khái niệm**: Thành viên hoặc lớp được khai báo là private chỉ có thể được truy cập từ bên trong cùng lớp.
* **Phạm vi**: Chỉ trong cùng một lớp.
* **Sử dụng**: Thường được sử dụng cho các biến hoặc phương thức chỉ dùng trong nội bộ lớp.

**Ví dụ:**

csharp

Sao chép mã

class Example

{

private int number; // Biến 'number' chỉ có thể truy cập từ bên trong lớp 'Example'

private void Display() // Phương thức 'Display' chỉ có thể truy cập từ bên trong lớp 'Example'

{

Console.WriteLine(number);

}

}

### 2. internal

* **Khái niệm**: Thành viên hoặc lớp được khai báo là internal chỉ có thể truy cập trong cùng một assembly (thường là cùng một dự án).
* **Phạm vi**: Trong cùng một assembly.
* **Sử dụng**: Dùng để chia sẻ mã giữa các lớp hoặc mô-đun trong cùng một dự án.

**Ví dụ:**

csharp

Sao chép mã

internal class Example

{

internal void ShowMessage()

{

Console.WriteLine("Hello, World!");

}

}

### 3. protected

* **Khái niệm**: Thành viên hoặc phương thức được khai báo là protected chỉ có thể truy cập từ bên trong lớp đó và từ các lớp dẫn xuất (kế thừa từ lớp đó).
* **Phạm vi**: Trong lớp đó và các lớp con kế thừa.
* **Sử dụng**: Dùng khi bạn muốn các lớp con có quyền truy cập vào thành viên lớp cha.

**Ví dụ:**

csharp

Sao chép mã

class BaseClass

{

protected int number; // Có thể truy cập từ lớp 'BaseClass' và các lớp kế thừa

protected void Display()

{

Console.WriteLine("This is a protected method.");

}

}

class DerivedClass : BaseClass

{

public void Show()

{

Display(); // Có thể truy cập vào phương thức 'Display' vì nó là 'protected'

}

}

### 4. protected internal

* **Khái niệm**: Thành viên hoặc phương thức được khai báo là protected internal có thể được truy cập từ bên trong cùng một assembly hoặc từ các lớp dẫn xuất trong một assembly khác.
* **Phạm vi**: Trong cùng một assembly hoặc lớp kế thừa trong các assembly khác.
* **Sử dụng**: Dùng để cung cấp khả năng truy cập rộng rãi hơn trong cùng một assembly và cho các lớp kế thừa trong các assembly khác.

**Ví dụ:**

csharp

Sao chép mã

public class Example

{

protected internal void ShowMessage()

{

Console.WriteLine("This is a protected internal method.");

}

}

### 5. public

* **Khái niệm**: Thành viên, phương thức, hoặc lớp được khai báo là public có thể được truy cập từ bất kỳ đâu.
* **Phạm vi**: Toàn bộ chương trình (các assembly khác cũng có thể truy cập được).
* **Sử dụng**: Dùng khi bạn muốn thành viên hoặc phương thức có thể truy cập từ bất kỳ đâu.

**Ví dụ:**

csharp

Sao chép mã

public class Example

{

public void ShowMessage()

{

Console.WriteLine("This is a public method.");

}

}

### Sử dụng các Access Modifiers

* **Variable (Biến):** Bạn có thể áp dụng các từ khóa truy cập cho biến để kiểm soát phạm vi truy cập của chúng. Ví dụ: private int age;
* **Method (Phương thức):** Tương tự, bạn có thể áp dụng các từ khóa truy cập cho phương thức. Ví dụ: internal void Calculate();
* **Class (Lớp):** Các lớp cũng có thể có các từ khóa truy cập, nhưng chỉ public và internal được sử dụng cho lớp (lớp không thể là private hoặc protected nếu chúng nằm bên ngoài lớp khác).

6

**Khái niệm**: enum là kiểu dữ liệu giá trị trong C# dùng để định nghĩa tập hợp các hằng số có tên. Mỗi hằng số trong enum được đại diện bởi một tên rõ ràng, giúp mã nguồn dễ đọc và dễ hiểu hơn.

**Khi nào nên sử dụng enum**: Sử dụng enum khi bạn cần một nhóm hằng số liên quan với nhau, ví dụ như các ngày trong tuần, các tháng trong năm, hoặc trạng thái của một quy trình.

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

6.2 **Khái niệm**: struct là kiểu dữ liệu giá trị trong C# dùng để nhóm các biến có liên quan lại với nhau. Không giống như class, struct không hỗ trợ kế thừa và thường được sử dụng cho các loại dữ liệu nhỏ gọn.

**Sự khác biệt giữa struct và class**:

* struct là kiểu dữ liệu giá trị (value type) và được lưu trữ trong stack, còn class là kiểu tham chiếu (reference type) và được lưu trữ trong heap.
* struct thích hợp cho các loại dữ liệu nhỏ và không yêu cầu xử lý kế thừa.
* struct có hiệu năng cao hơn vì nó không cần phải dùng bộ thu gom rác (garbage collector).

### 6.3. ****Var và Dynamic****

* **Sự khác biệt giữa var và dynamic**:
  + var là từ khóa khai báo kiểu dữ liệu tường minh (ngầm hiểu) được xác định trong thời gian biên dịch. Sau khi kiểu của var được gán, nó không thể thay đổi.
  + dynamic là kiểu dữ liệu quyết định trong thời gian chạy. Nó có thể thay đổi kiểu của giá trị gán trong suốt thời gian thực thi.
  + var đảm bảo tính an toàn kiểu tại thời gian biên dịch, trong khi dynamic không đảm bảo điều này.

Dưới đây là phần lý thuyết và các ví dụ cụ thể cho các khái niệm trong C# để bạn dễ hiểu hơn:

### 1. ****Enum (Enumeration)****

* **Khái niệm**: enum là kiểu dữ liệu giá trị trong C# dùng để định nghĩa tập hợp các hằng số có tên. Mỗi hằng số trong enum được đại diện bởi một tên rõ ràng, giúp mã nguồn dễ đọc và dễ hiểu hơn.
* **Cách sử dụng**: Dùng enum để biểu diễn tập hợp các giá trị mà một biến có thể có. Ví dụ, bạn có thể dùng enum để biểu diễn các ngày trong tuần:

csharp

Sao chép mã

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

* **Khi nào nên sử dụng enum**: Sử dụng enum khi bạn cần một nhóm hằng số liên quan với nhau, ví dụ như các ngày trong tuần, các tháng trong năm, hoặc trạng thái của một quy trình.
* **Ví dụ**:

csharp

Sao chép mã

enum OrderStatus { Pending, Approved, Rejected }

class Program

{

static void Main()

{

OrderStatus status = OrderStatus.Approved; // Gán giá trị Approved cho biến status

Console.WriteLine($"Order status is: {status}"); // Kết quả: Order status is: Approved

if (status == OrderStatus.Approved)

{

Console.WriteLine("The order has been approved.");

}

}

}

### 2. ****Struct****

* **Khái niệm**: struct là kiểu dữ liệu giá trị trong C# dùng để nhóm các biến có liên quan lại với nhau. Không giống như class, struct không hỗ trợ kế thừa và thường được sử dụng cho các loại dữ liệu nhỏ gọn.
* **Sự khác biệt giữa struct và class**:
  + struct là kiểu dữ liệu giá trị (value type) và được lưu trữ trong stack, còn class là kiểu tham chiếu (reference type) và được lưu trữ trong heap.
  + struct thích hợp cho các loại dữ liệu nhỏ và không yêu cầu xử lý kế thừa.
  + struct có hiệu năng cao hơn vì nó không cần phải dùng bộ thu gom rác (garbage collector).
* **Ví dụ**:

csharp

Sao chép mã

struct Point

{

public int X;

public int Y;

public Point(int x, int y)

{

X = x;

Y = y;

}

}

class Program

{

static void Main()

{

Point p = new Point(10, 20); // Tạo một đối tượng Point

Console.WriteLine($"Point coordinates: X = {p.X}, Y = {p.Y}"); // Kết quả: X = 10, Y = 20

}

}

### 3. ****Var và Dynamic****

* **Sự khác biệt giữa var và dynamic**:
  + var là từ khóa khai báo kiểu dữ liệu tường minh (ngầm hiểu) được xác định trong thời gian biên dịch. Sau khi kiểu của var được gán, nó không thể thay đổi.
  + dynamic là kiểu dữ liệu quyết định trong thời gian chạy. Nó có thể thay đổi kiểu của giá trị gán trong suốt thời gian thực thi.
  + var đảm bảo tính an toàn kiểu tại thời gian biên dịch, trong khi dynamic không đảm bảo điều này.
* **Ví dụ**:

csharp

Sao chép mã

class Program

{

static void Main()

{

// Sử dụng var

var number = 10; // Kiểu dữ liệu là int

Console.WriteLine(number); // Kết quả: 10

// number = "Hello"; // Lỗi biên dịch vì kiểu int không thể gán string

// Sử dụng dynamic

dynamic anything = 10; // Ban đầu là kiểu int

Console.WriteLine(anything); // Kết quả: 10

anything = "Hello"; // Thay đổi kiểu thành string

Console.WriteLine(anything); // Kết quả: Hello

}

}

### 4. ****Ref và Out****

* **Sự khác biệt giữa ref và out**:
  + Cả ref và out đều được sử dụng để truyền tham chiếu cho các tham số trong hàm.
  + Với ref, biến truyền vào phải được khởi tạo trước khi được truyền cho phương thức.
  + Với out, biến truyền vào không cần được khởi tạo trước, nhưng phải được gán giá trị bên trong phương thức trước khi phương thức kết thúc.

### 1. Generic Type Parameters (Tham số Kiểu Generic)

**Tham số kiểu Generic** là các ký tự đại diện cho một kiểu dữ liệu. Khi bạn tạo một lớp, phương thức, hoặc giao diện generic, bạn định nghĩa một tham số kiểu generic mà có thể được thay thế bằng một kiểu cụ thể khi sử dụng.

Ví dụ, trong lớp List<T>, T là tham số kiểu generic. Khi bạn tạo một đối tượng List<int>, T sẽ được thay thế bằng int.

### 2. Generic Classes (Lớp Generic)

**Lớp Generic** là lớp có thể làm việc với bất kỳ kiểu dữ liệu nào. Nó cho phép bạn tạo một lớp mà không cần chỉ định trước kiểu dữ liệu mà nó sẽ làm việc. Kiểu dữ liệu được xác định khi bạn khởi tạo đối tượng của lớp.

### 3. Generic Interfaces (Giao diện Generic)

**Giao diện Generic** định nghĩa một hợp đồng cho các hoạt động generic có thể làm việc với bất kỳ kiểu dữ liệu nào. Các lớp triển khai giao diện này sẽ chỉ định kiểu dữ liệu khi sử dụng.

### 4. Generic Methods (Phương thức Generic)

**Phương thức Generic** là phương thức có thể hoạt động trên các tham số của bất kỳ kiểu nào. Bạn khai báo phương thức generic bằng cách đặt một tham số kiểu trong dấu ngoặc nhọn (<T>) trước kiểu trả về của phương thức.

**Exception** trong C# là một cơ chế dùng để xử lý các lỗi hoặc tình huống không mong muốn xảy ra trong quá trình thực thi chương trình. Khi một lỗi xảy ra, một "exception" (ngoại lệ) sẽ được ném ra (throw) và nếu không được bắt (catch), chương trình sẽ bị dừng lại.

**Exception Handling** (Xử lý ngoại lệ) cho phép bạn xử lý các lỗi này để chương trình không bị dừng đột ngột và có thể tiếp tục chạy hoặc kết thúc một cách hợp lý.

* 1 số Exception thường gặp

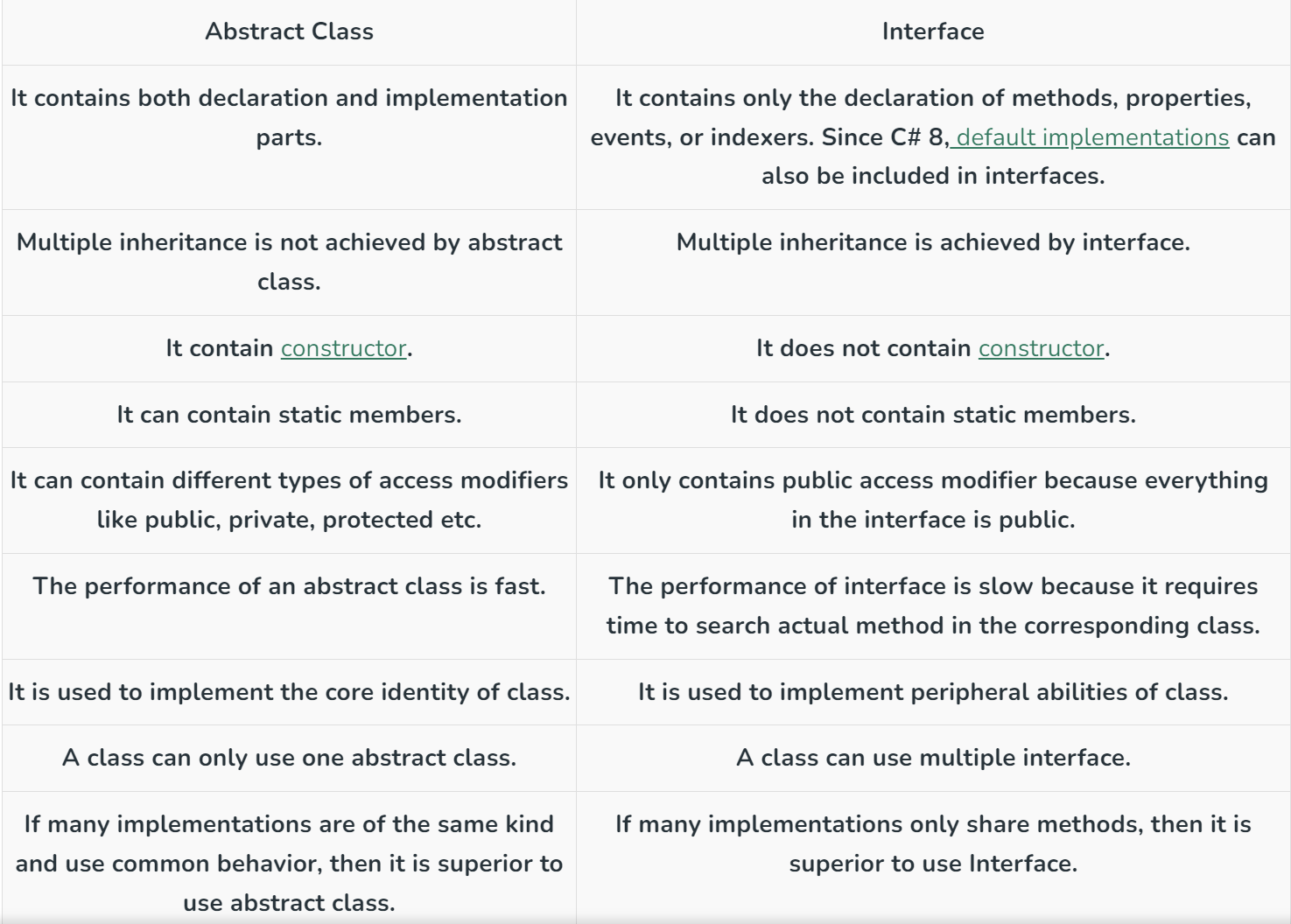
+ **System.NullReferenceException**: Xảy ra khi bạn cố gắng truy cập một thành viên trên một đối tượng không được khởi tạo (null).

**System.IndexOutOfRangeException**: Xảy ra khi bạn cố gắng truy cập một phần tử bên ngoài giới hạn của mảng.

**System.DivideByZeroException**: Xảy ra khi có phép chia cho số 0.  
**Cách xử lý:** Kiểm tra mẫu số trước khi thực hiện phép chia.

 **System.InvalidOperationException**: Xảy ra khi một thao tác không hợp lệ được thực hiện trong trạng thái hiện tại của đối tượng.  
**Cách xử lý:** Đảm bảo đối tượng hoặc tài nguyên đang ở trạng thái thích hợp trước khi thực hiện thao tác.

 **System.FormatException**: Xảy ra khi một chuỗi không thể được chuyển đổi thành một định dạng mong muốn.  
**Cách xử lý:** Sử dụng phương thức TryParse để kiểm tra định dạng trước khi chuyển đổi.





#### 1. OOP trong Lập Trình Nói Chung:

* **Khái niệm OOP:** Lập trình Hướng đối tượng là một mô hình lập trình sử dụng "đối tượng" để đại diện cho dữ liệu và các phương thức. Các nguyên lý chính của OOP bao gồm đóng gói, kế thừa, đa hình, và trừu tượng.
* **Ưu điểm:**
  + **Đóng gói:** Kết hợp dữ liệu và các phương thức thao tác dữ liệu trong một đơn vị duy nhất, bảo vệ dữ liệu khỏi sự can thiệp từ bên ngoài.
  + **Kế thừa:** Cho phép tạo ra các lớp mới dựa trên các lớp đã có, khuyến khích tái sử dụng mã nguồn.
  + **Đa hình:** Cho phép sử dụng một giao diện duy nhất để đại diện cho các dạng thức (kiểu dữ liệu) khác nhau.
  + **Trừu tượng hóa:** Đơn giản hóa thực tế phức tạp bằng cách mô hình hóa các lớp phù hợp với vấn đề.

**Nhược điểm:**

* Có thể phức tạp và khó học đối với người mới bắt đầu.
* Đôi khi dẫn đến quá tải hiệu suất.
* Tốn tài nguyên trong việc quản lý đối tượng và lượng mã nguồn cần viết.

#### OOP trong C#:

* **Khái niệm OOP trong C#:** C# là một ngôn ngữ lập trình hiện đại, hướng đối tượng và an toàn kiểu (type-safe) do Microsoft phát triển. Nó được thiết kế để hoạt động trong hệ sinh thái .NET, cung cấp hỗ trợ mạnh mẽ cho OOP.
* **Ưu điểm trong C#:**
  + **Tích hợp với .NET Framework:** Cho phép truy cập vào một thư viện lớn các giải pháp mã nguồn đã được lập trình sẵn cho các vấn đề lập trình phổ biến.
  + **Mã Quản lý (Managed Code):** Cung cấp quản lý bộ nhớ và xử lý ngoại lệ.
  + **Tính Năng Nâng Cao:** Hỗ trợ các thuộc tính, sự kiện, ủy nhiệm (delegates), và biểu thức lambda giúp nâng cao thực hành OOP.
  + **Dễ Sử Dụng:** Cú pháp trực quan và môi trường phát triển phong phú (như Visual Studio) giúp đơn giản hóa việc phát triển.
* **Nhược điểm trong C#:**
  + **Phụ Thuộc Nền Tảng:** Mặc dù .NET Core có thể chạy trên nhiều nền tảng, .NET truyền thống chủ yếu chạy trên Windows.
  + **Đường Cong Học Tập:** Sự kết hợp giữa ngôn ngữ C# và khung làm việc .NET có thể gây khó khăn cho người mới học.
  + **Quá Tải Hiệu Suất:** Mã quản lý có thể gây ra một số chi phí hiệu suất so với các ngôn ngữ không quản lý như C++.

### Collections trong C# là gì?

Các lớp hỗ trợ lưu trữ, quản lý và thao tác với các đối tượng một cách có thứ tự.

Các lớp này nằm trong namespace System.Collections.

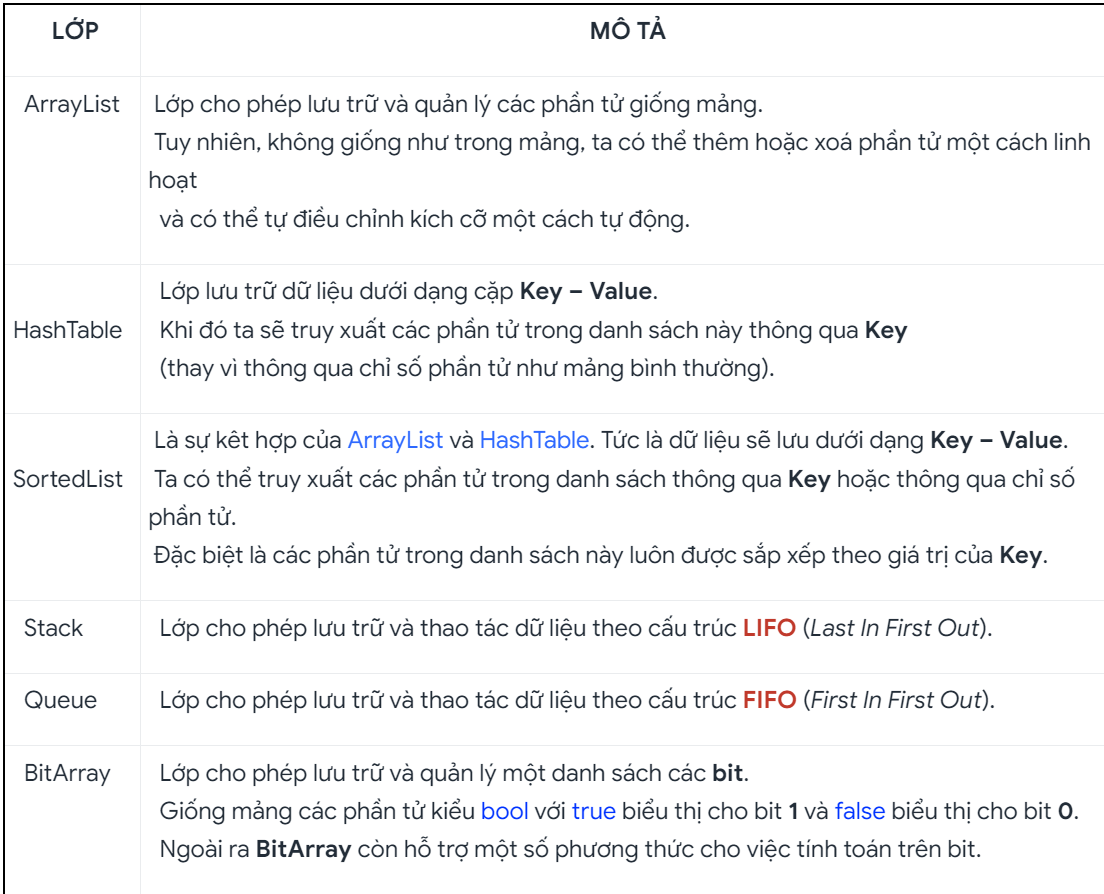
Là một mảng có kích thước động:

* Không cần khai báo kích thước khi khởi tạo.
* Có thể tăng giảm số lượng phần tử trong mảng một cách linh hoạt.

Có thể lưu trữ một tập hợp đối tượng thuộc nhiều kiểu khác nhau.

Hỗ trợ rất nhiều phương thức để thao tác với tập hợp như: tìm kiếm, sắp xếp, đảo ngược, . . .

Mỗi collections được tổ chức thành một lớp nên cần khởi tạo đối tượng trước khi sử dụng.



Các Generic Collections trong C# cung cấp các tập hợp tổng quát an toàn kiểu, giúp tránh lỗi runtime liên quan đến kiểu dữ liệu và cải thiện hiệu suất:

* **List<T>:** Một danh sách tổng quát, có thể chứa các phần tử của một kiểu cụ thể. Thay thế cho ArrayList.
* **LinkedList<T>:** Một danh sách liên kết hai chiều tổng quát.
* **Dictionary<TKey, TValue>:** Một tập hợp tổng quát lưu trữ cặp khóa-giá trị với tính an toàn kiểu.
* **SortedDictionary<TKey, TValue>:** Tương tự như Dictionary, nhưng các phần tử được sắp xếp theo khóa.
* **Stack<T>:** Phiên bản tổng quát của Stack, chỉ chứa các phần tử của một kiểu cụ thể.
* **Queue<T>:** Phiên bản tổng quát của Queue, chỉ chứa các phần tử của một kiểu cụ thể.

Linq là một query language được dùng để truy vấn dữ liệu từ các nguồn dữ liệu khác nhau (data sources).

Trong câu query sẽ có 3 mệnh đề (clause) from, where, select tương tự như khi viết query trong database.

* from: nguồn lấy dữ liệu (data source)
* where: điều kiện để lọc dữ liệu. (filter data)
* select: lấy kết quả
* lambda expressions được dùng để tạo ra một function ẩn danh (anonymous function).
* productId => productId % 2 == 0 là một biểu thức lambda (Lambda expression) với productId là tham số đầu vào (input parameter) của function ẩn danh.