



ASP.NET Core

What Is It?

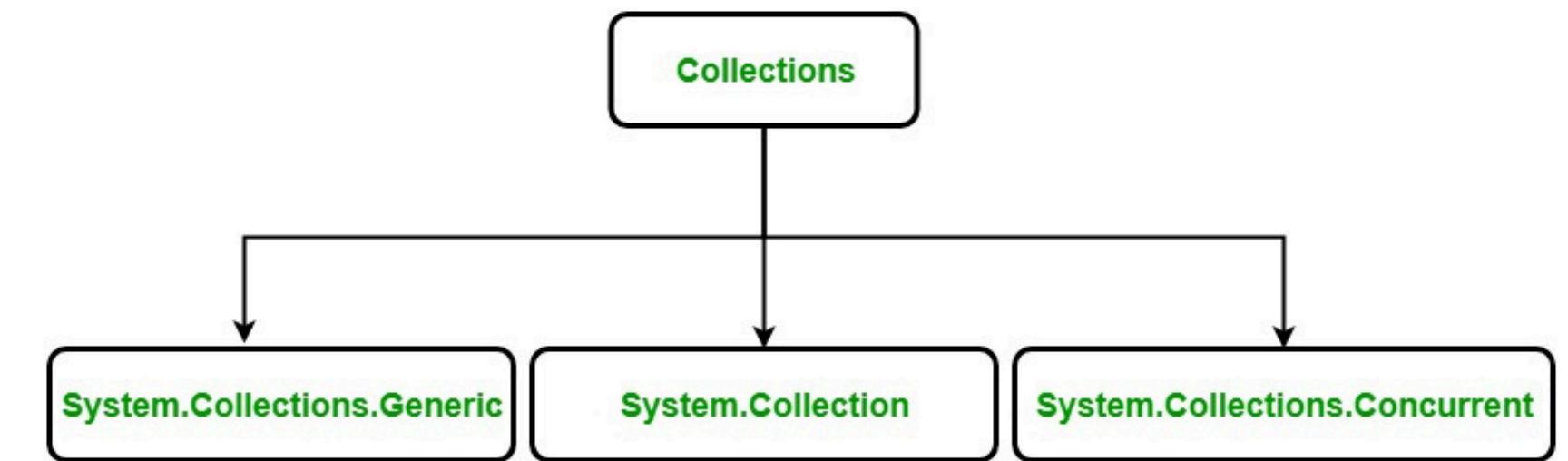
FULL STACK WEB C# ASP.NET CORE

TỪ ZERO TỚI ĐƯỢC NHÂN VIỆC VỚI KỸ NĂNG CAO CẤP MỚI NHẤT

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT CƠ BẢN

- CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG
- OBJECT & DYNAMIC & VAR

```
// List<string>: Danh sách lưu trữ các chuỗi  
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };  
  
// List<int>: Danh sách lưu trữ các số nguyên  
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
  
// Dictionary<string, int>: Từ điển lưu trữ cặp khóa-giá trị (key-value), khóa là chuỗi, giá trị là số nguyên  
Dictionary<string, int> ages = new Dictionary<string, int> { { "Alice", 25 }, { "Bob", 30 }, { "Charlie", 22 } };  
  
// HashSet<int>: Tập hợp không chứa phần tử trùng lặp  
HashSet<int> uniqueNumbers = new HashSet<int> { 1, 2, 3, 4, 5 };  
  
// LinkedList<string>: Danh sách liên kết kép, cho phép chèn và xóa phần tử ở cả hai đầu  
LinkedList<string> linkedList = new LinkedList<string>(new[] { "First", "Second", "Third" });  
  
// SortedList<string, int>: Từ điển với các phần tử được sắp xếp theo khóa (key)  
SortedList<string, int> sortedAges = new SortedList<string, int> { { "Alice", 25 }, { "Bob", 30 }, { "Charlie", 22 } };  
  
// ObservableCollection<string>: Danh sách có thể theo dõi thay đổi và phản hồi khi danh sách bị sửa đổi  
var observableNames = new ObservableCollection<string> { "Alice", "Bob", "Charlie" };
```





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

Các kiểu dữ liệu tập hợp

- **List (LinkedList, SortedList)**
- **Dictionary**
- **HashSet**
- **Array (tương tự collection) - ArrayList**

1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

LIST (LINKEDLIST, SORTEDLIST)

- Trong lập trình để lưu trữ nhiều giá trị thay vì ta khai báo nhiều biến thì C# hỗ trợ chúng ta một số kiểu dữ liệu trên để lưu trữ tập hợp biến. Một trong số đó khá phổ biến là list thuộc thư viện System.Collections.Generic;
- Ví dụ:
 - Danh sách user
 - Danh mục menu game
 - Bảng xếp hạng

KIỂU 1: CẤU TRÚC LIST

...

```
String hoTen0 = "Nam";
String hoTen1 = "Minh";
String hoTen2 = "Hằng";
String hoTen3 = "Long";
String hoTen4 = "Khải";
```

Syntax

↑



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

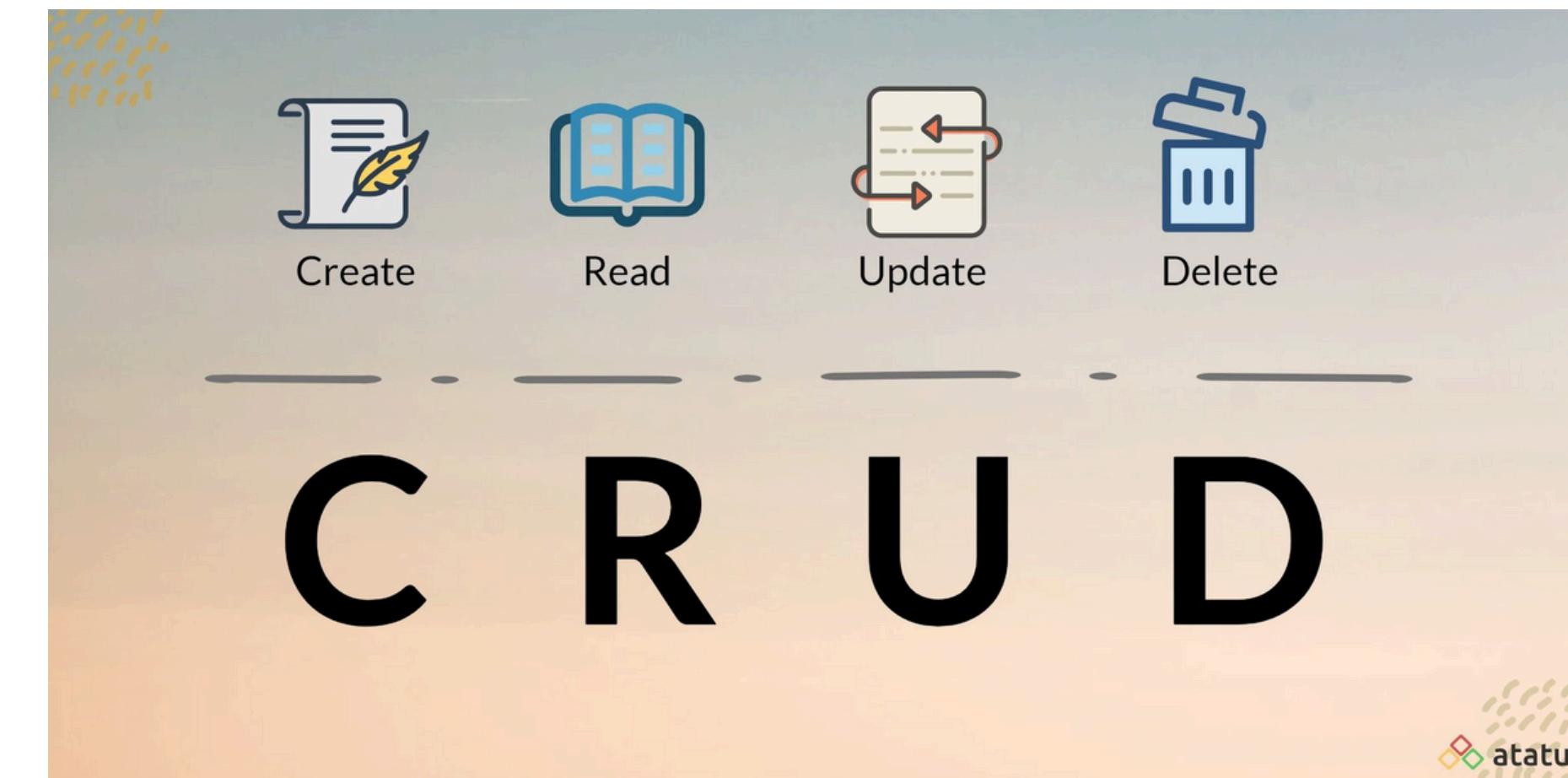
9

10

LIST (LINKEDLIST, SORTEDLIST)

Trong mọi ứng dụng, danh sách (list) dữ liệu ví dụ như danh sách sản phẩm hay khách hàng luôn cần được quản lý hiệu quả. CRUD giúp bạn thêm mới(Create), đọc(Read or Search), sửa (Update) và xóa (Delete) dữ liệu trong các danh sách đó. Hiểu rõ cách áp dụng CRUD với danh sách sẽ giúp bạn dễ dàng xử lý dữ liệu trong các dự án.

List là 1 cấu trúc dữ liệu giúp ta lưu trữ dữ liệu dưới dạng danh sách cung cấp đầy đủ các phương thức giúp ta có thể thực hiện được các thao tác CRUD.





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

LIST METHOD

LIST (READ): log ra màn hình console

- **List<string> names = new List<string> { "A", "B", "C", "D", "E" };**
- Để xem giá trị list trên console ta có thể dùng **String.Join(“,”,names)**.

The screenshot shows a C# code editor with the following code:

```
❖ Khai báo list và hiển thị
List<string> names = new List<string> { "A", "B", "C", "D", "E" };
Console.WriteLine(String.Join(", ", names));
```

To the right of the code editor is a preview window titled "Output" which displays the result of the execution:

Name

A B C D E



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

1

2

3

4

5

6

7

8

9

10

LIST (READ): lấy giá trị phần tử trong List

```
List<string> lstName = new List<string> { "A", "B", "C", "D", "E" };
```

Các thuộc tính của list dùng để truy xuất giá trị.

.Count: số lượng phần tử trong mảng (ứng với list trên sẽ là 5)

[index] hoặc elementAt(index): index từ trái sang phải 0 -> count - 1, ngoài ra index theo hướng từ phải sang thì sẽ là ^1 - đến ^count.

Ví dụ:

lstName[0] = "A" hoặc lstName.elementAt(0);

lstName[^1] = "E" hoặc lstName.elementAt(^);

.Sort(): Sắp xếp theo thứ tự tăng dần

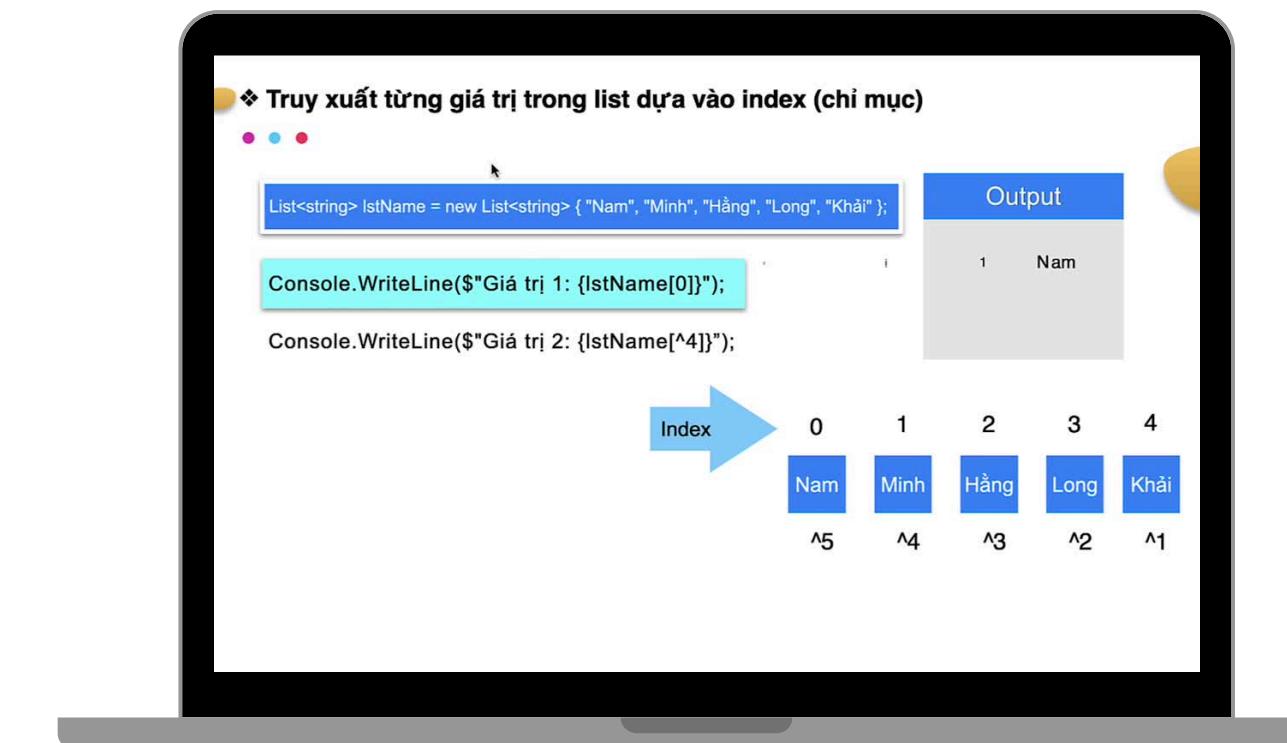
.Reverse(): Đảo ngược vị trí các phần tử trong mảng

.ToArray(): Chuyển đổi list thành Array()

.Find(lamda): Tìm 1 phần tử trong list thỏa điều kiện Predicate (lamda expression)

.FindAll(lamda): Tìm nhiều phần tử trong list thỏa điều kiện Predicate (lamda expression)

.Exists(lamda): Kiểm tra xem có phần tử nào thỏa mãn điều kiện trong Predicate (lamda expression).





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

LIST (READ): duyệt list

- Để duyệt list ta có thể dùng vòng lặp for hoặc foreach

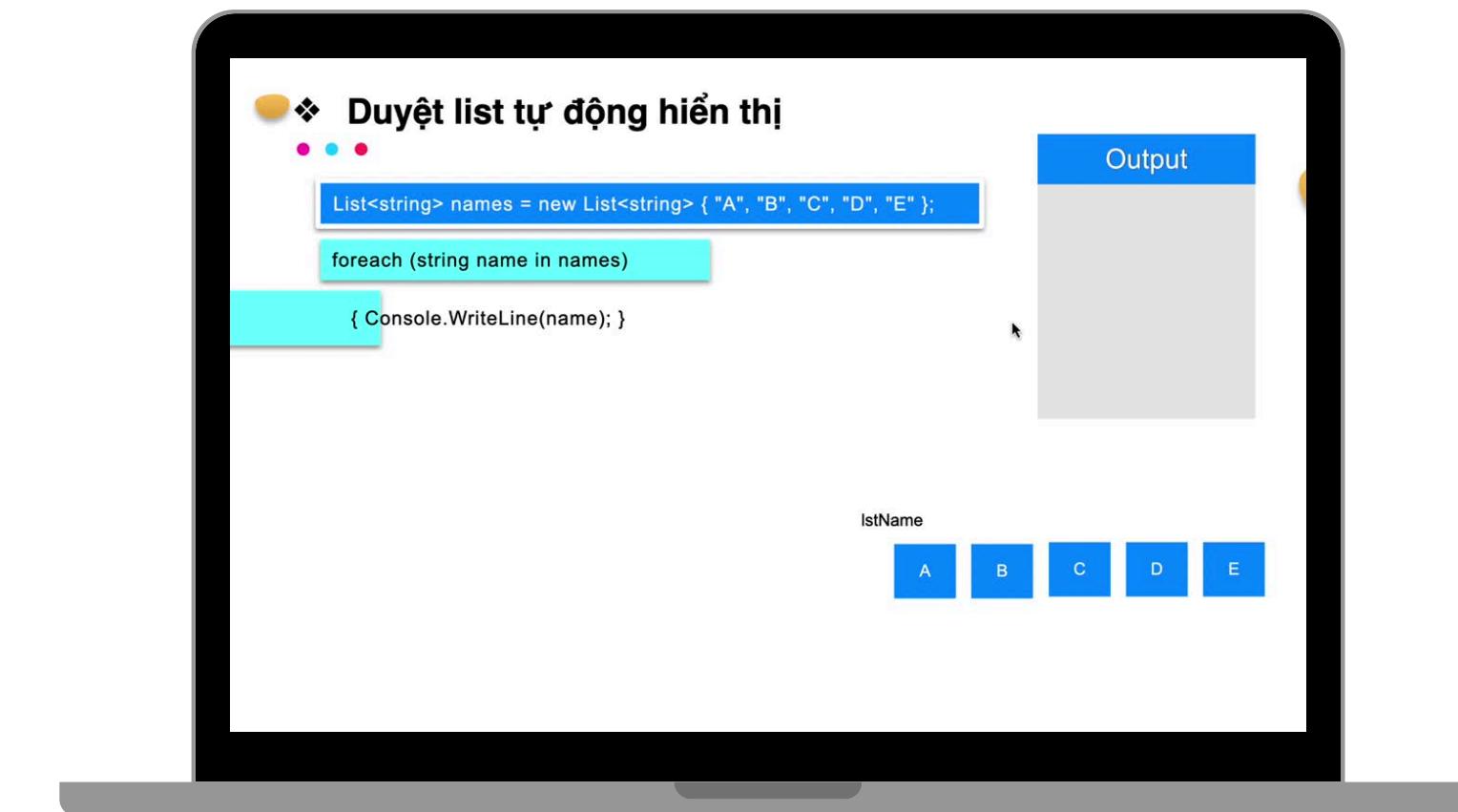
// Duyệt qua danh sách sử dụng vòng lặp foreach

```
foreach (string name in lstName) {  
    Console.WriteLine(name);  
}
```

// Duyệt qua danh sách sử dụng vòng lặp for

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

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

LIST (READ): kiểm tra phần tử trong list

- Kiểm tra giá trị có trong list hay không sử dụng **list.Contains("gia_trí");**

The screenshot shows a C# code editor with the following code:

```
List<string> lstName = new List<string> { "A", "B", "C", "D", "E" };
if (lstName.Contains("B"))
{
    Console.WriteLine("Yes");
}
else
{
    Console.WriteLine("No");
}
```

To the right of the code editor is a window titled "Output" which displays the result of the program's execution. Below the code editor, there is a small diagram showing five blue boxes labeled A, B, C, D, and E.

1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

1

2

3

4

5

6

7

8

9

10

LIST (Create): Thêm phần tử vào list

```
List<int> numbers = new List<int> { 10, 20, 30, 40};
```

- **.Add(100)**: Thêm phần tử vào cuối list.
- **.AddRange(new int[] { 3, 4, 5 })**: Thêm nhiều phần tử vào cuối list.
- **.Insert(1, 99)**: Thêm phần tử vào vị trí chỉ định.
- **.InsertRange(1, new int[] { 99, 100 })**: Thêm nhiều phần tử vào vị trí chỉ định.

Lưu ý: Khi dùng insert để chèn giá trị vị trí các phần tử sẽ bị thay đổi

The screenshot shows a C# code editor window. The code is:

```
❖ Các hàm xử lý list ( append - thêm phần tử vào list)
...
IstNumber = {10,20,30,40}
Istnumber.Add(100)
Write(Istnumber)
```

The output window shows the list: 10

The screenshot shows a C# code editor window. The code is:

```
❖ Các hàm xử lý list ( insert - thêm phần tử vào list vị trí bất kỳ)
...
IstNumber = [10,20,30,40]
Istnumber.Insert(0,100)
write(result)
```

The output window shows the list: 10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

UPDATE (Create): CẬP NHẬT GIÁ TRỊ

```
List<string> lstName = new List<string> { "A", "B", "C", "D", "E" };  
lstName[0] = "X";  
lstName = { "X", "B", "C", "D", "E" };
```

1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

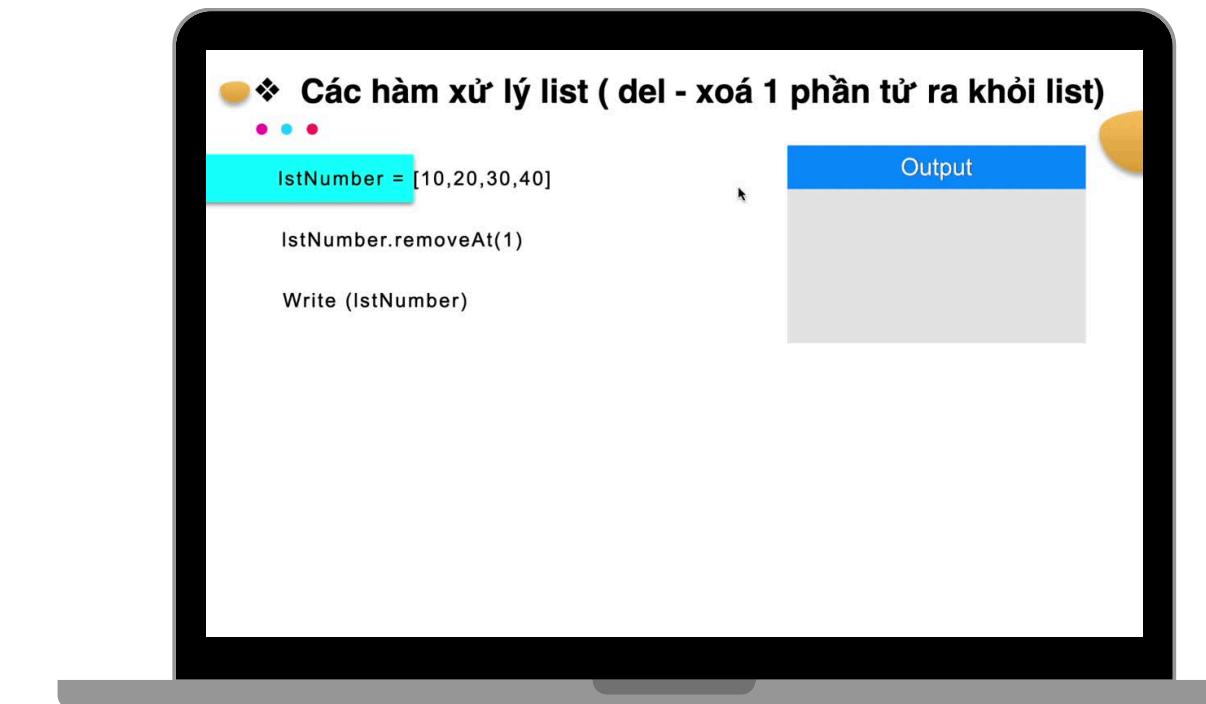
LIST METHOD

DELETE (XOÁ): XOÁ GIÁ TRỊ

```
List<int> numbers = new List<int> { 10, 20, 30, 40};
```

- **.Remove(10):** Xoá giá trị 10 ra khỏi List.
- **.RemoveAt(index):** Xoá phần tử tại vị trí index ra khỏi List.
- **.RemoveAll(x=>x>==20): Xoá phần tử thoả điều kiện Predicate (lamda expression).**
- **.Clear():** Xoá tất cả phần tử. (vẫn còn lst rỗng)
- **numbers = null:** xoá list hoàn toàn khỏi bộ nhớ.

Lưu ý: Khi xoá phần tử ra khỏi list thì vị trí các phần tử sẽ bị thay đổi



1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

1

2

3

4

5

6

7

8

9

10

IstNumber = [20, 81, 97, 63, 72, 11, 20, 15, 33, 15, 41, 20]

- **Tính tổng các số lớn hơn 50 trong danh sách**
 - Yêu cầu: Viết chương trình tính tổng các số trong IstNumber mà lớn hơn 50.
- **Đếm số phần tử lớn hơn 30**
 - Yêu cầu: Viết chương trình đếm số phần tử trong danh sách IstNumber mà lớn hơn 30.
- **Tìm số lớn nhất trong danh sách**
 - Yêu cầu: Viết chương trình để tìm số lớn nhất trong IstNumber.
- **Tính trung bình cộng của các số lẻ**
 - Yêu cầu: Viết chương trình để tính trung bình cộng của các số lẻ trong danh sách IstNumber.
- **In ra các số chẵn trong danh sách**
 - Yêu cầu: Viết chương trình để in ra tất cả các số chẵn trong IstNumber.
- **Tìm vị trí đầu tiên của số 20 trong danh sách**
 - Yêu cầu: Viết chương trình để tìm vị trí đầu tiên của số 20 trong danh sách IstNumber.
- **Tìm số lượng phần tử bằng 15 trong danh sách**
 - Yêu cầu: Viết chương trình để đếm số lượng phần tử bằng 15 trong IstNumber.
- **Tính tổng các số nhỏ hơn 40**
 - Yêu cầu: Viết chương trình tính tổng các số trong danh sách IstNumber nhỏ hơn 40.
- **Đếm số lượng các số chia hết cho 5**
 - Yêu cầu: Viết chương trình để đếm bao nhiêu số trong danh sách chia hết cho 5.
- **Tạo danh sách mới chỉ chứa các số nhỏ hơn 50**
 - Yêu cầu: Viết chương trình để tạo một danh sách mới chỉ chứa các số nhỏ hơn 50 từ danh sách IstNumber.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

LIST METHOD

```
IstStrings = ["apple", "banana", "orange", "kiwi", "mango", "pineapple", "grape", "melon"]
```

- **Tính độ dài của mảng**
 - Yêu cầu: Viết chương trình để đếm số phần tử trong mảng IstStrings.
- **In ra các chuỗi dài hơn 5 ký tự**
 - Yêu cầu: Viết chương trình để in ra các chuỗi trong IstStrings có độ dài lớn hơn 5 ký tự.
- **Tìm chuỗi dài nhất trong mảng**
 - Yêu cầu: Viết chương trình để tìm chuỗi có độ dài lớn nhất trong mảng IstStrings.
- **In ra các chuỗi có chứa chữ 'a'**
 - Yêu cầu: Viết chương trình để in ra tất cả các chuỗi trong IstStrings có chứa chữ cái 'a'.
- **Tìm chuỗi bắt đầu bằng chữ 'm'**
 - Yêu cầu: Viết chương trình để tìm tất cả các chuỗi trong IstStrings bắt đầu bằng chữ 'm'.
- **Đếm số chuỗi có độ dài nhỏ hơn 6 ký tự**
 - Yêu cầu: Viết chương trình để đếm số chuỗi có độ dài nhỏ hơn 6 ký tự trong IstStrings.
- **In ra chuỗi dài thứ hai trong mảng**
 - Yêu cầu: Viết chương trình để tìm và in ra chuỗi dài thứ hai trong mảng IstStrings.
- **Sắp xếp mảng theo thứ tự bảng chữ cái**
 - Yêu cầu: Viết chương trình để sắp xếp mảng IstStrings theo thứ tự bảng chữ cái (A-Z).
- **Chuyển tất cả các chuỗi thành chữ hoa**
 - Yêu cầu: Viết chương trình để chuyển tất cả các chuỗi trong IstStrings thành chữ in hoa.
- **Thay thế chuỗi "banana" bằng "pear"**
 - Yêu cầu: Viết chương trình để thay thế chuỗi "banana" bằng "pear" trong IstStrings.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP

❖ Bài tập 1

IstNumber = [20,81,97,63,72,11,20,15,33,15,41,20]

Bài toán: Tính tổng của các số trong một mảng.

Mô tả:

Bạn được cung cấp một mảng số nguyên IstNumber. Nhiệm vụ của bạn là tính tổng của tất cả các số trong mảng này.

Input:

IstNumber: Một danh sách (mảng) chứa các số nguyên. Đây là mảng bạn cần tính tổng.

Output:

Trả về tổng của tất cả các số trong mảng IstNumber.

1

2

3

4

5

6

7

8

9

10





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP

❖ Bài tập 2

```
lst_number = [2, 7, 11, 15]
```

Mô tả: Tìm hai số trong một danh sách số nguyên sao cho tổng của chúng bằng một giá trị target cho trước.

Ví dụ:

Input: nums = [2, 7, 11, 15], target = 9

Output: [0, 1] (vì nums[0] + nums[1] = 2 + 7 = 9) ngược lại nếu không có

1

2

3

4

5

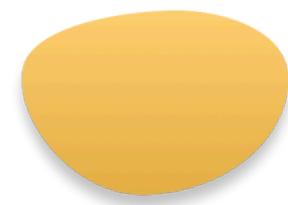
6

7

8

9

10





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP

❖ Bài tập 3

```
IstNumber = [1, 1, 2, 2, 2, 3, 4, 4, 5]
```

Remove Duplicates from Sorted Array (Easy):

Mô tả: Loại bỏ các phần tử trùng lặp từ một mảng đã sắp xếp và trả về chiều dài của mảng mới.

Ví dụ:

Input: nums = [1, 1, 2, 2, 2, 3, 4, 4, 5]

Output: 5 (mảng mới là [1, 2, 3, 4, 5])

1

2

3

4

5

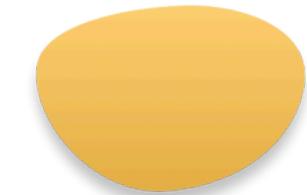
6

7

8

9

10





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP

❖ Bài tập 4

IstNumber = [1, 1, 1, 2, 2, 3]

Remove Duplicates from Sorted Array (Easy):

Mô tả: Cho một mảng số nguyên, tìm k phần tử xuất hiện nhiều lần nhất trong mảng và trả về chúng dưới dạng danh sách. Nếu có nhiều phần tử có cùng tần số xuất hiện, trả về bất kỳ trong số chúng. **Ví dụ:**

Input: nums = [1, 1, 1, 2, 2, 3], k = 2

Output: [1, 2]

Giải thích: Trong mảng nums, số 1 xuất hiện 3 lần, số 2 xuất hiện 2 lần và số 3 xuất hiện 1 lần. Ta cần trả về 2 phần tử xuất hiện nhiều lần nhất, và chúng có thể là 1 và 2 (hoặc 2 và 1).

Lưu ý:

Kết quả có thể được trả về dưới bất kỳ thứ tự nào.

Số lần xuất hiện của các phần tử không cần phải theo thứ tự tăng dần.

1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP

❖ Bài tập 5

prices = [7,1,5,3,6,4]

Best Time to Buy and Sell Stock

Mô tả: Cho một mảng prices , mỗi phần tử của nó đại diện cho giá cổ phiếu trong một ngày. Bạn chỉ được mua cổ phiếu một lần và bán cổ phiếu một lần. Hãy tìm giá trị lớn nhất bạn có thể có từ việc mua và bán cổ phiếu.

Ví dụ:

Input: prices = [7,1,5,3,6,4]

Output: 5

Giải thích: Bạn mua vào ngày thứ 2 (giá 1) và bán vào ngày thứ 5 (giá 6), lãi là $6-1 = 5$.

1

2

3

4

5

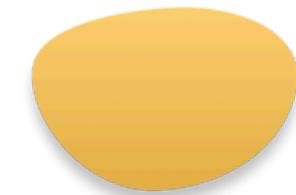
6

7

8

9

10





CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

DICTIONARY

1

2

3

4

5

6

7

8

9

10

- Ngoài list ra ta còn 1 cấu trúc khác để lưu trữ được nhiều giá trị trên 1 biến đó là dictionary. Dictionary được dùng để lưu trữ dữ liệu dưới dạng các cặp key, value. Trong đó key đóng vai trò tương tự như index (chỉ khác ở chỗ key là do ta tự định nghĩa chứ không giống như index tăng dần từ 0 -> length - 1 hoặc -1 đến - length) dùng để truy xuất đến value.

```
Dictionary<string, object> data = new Dictionary<string, object>
{
    { "id", "01" },
    { "name", "Tấn Khải" },
    { "age", 50 },
    { "job_title", "Developer" }
};
```

Tạo nhanh dữ liệu

```
data = {
    "id": "01",
    "name": "Tấn Khải",
    "age": 50,
    "job_title": "Developer"
}
```

```
C# Main.cs x
app > C# Main.cs
1 // Tạo dictionary rỗng
2 Dictionary<string, string> data = new Dictionary<string, string>();
3
4 // Thêm từng phần tử vào dictionary
5 data.Add("id", "01");
6 data.Add("name", "Tấn Khải");
7 data.Add("age", "50");
8 data.Add("job_title", "Developer");
9
10 // Truy xuất hoặc cập nhật các giá trị nếu cần
11 Console.WriteLine($"ID: {data["id"]}");
12 Console.WriteLine($"Name: {data["name"]}");
13 Console.WriteLine($"Age: {data["age"]}");
14 Console.WriteLine($"Job Title: {data["job_title"]});
```

Tạo list rỗng sau đó add dữ liệu vào



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

DICTIONARY

1

2

3

4

5

6

7

8

9

10

Các ưu điểm của dictionary

Tìm kiếm và truy xuất nhanh hơn, so với list
thay vì phải truy xuất dựa trên tìm index thì

```
C# Main.cs  x
app > C# Main.cs
1  using System.Collections.Generic;
2  using System.Linq;
3  // Tạo dictionary lưu trữ tên sinh viên và điểm số
4  Dictionary<string, int> studentScores = new Dictionary<string, int>();
5
6  // Thêm phần tử
7  studentScores.Add("Trinh", 85);
8  studentScores["Nga"] = 90; // Truy cập hoặc thêm giá trị
9
10 // Truy xuất giá trị
11 int trinhScore = studentScores["Trinh"];
12
13 // Cập nhật giá trị
14 studentScores["Nga"] = 95;
15
16 // Xóa phần tử
17 studentScores.Remove("Nga");
18
19
20 //Lấy tất cả các Values của Dic chuyển về List
21 studentScores.Values.ToList();
22 //Lấy tất cả các key của Dic chuyển về List
23 studentScores.Keys.ToList();
```

KIỂU 2: CẤU TRÚC DICTIONARY

❖ Các ưu điểm của dictionary

- Tìm kiếm và truy xuất nhanh hơn, so với list thay vì phải truy xuất
dựa trên tìm index thì

```
data = {
    "id": "01",
    "name": "Tấn Khải",
    "age": 50,
    "job_title": "Developer"
}
```

```
Write(data["id"])
Write(data["name"])
Write(data["age"])
Write(data["job_title"])
```

Output

01



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

HASHSET

1

2

3

4

5

6

7

8

9

10

- Set là một cấu trúc dữ liệu trong Python, là một tập hợp các phần tử không có thứ tự và không trùng lặp. Set được sử dụng để lưu trữ các giá trị duy nhất.

1. Khởi tạo HashSet với các phần tử:

csharp

[Sao chép mã](#)

```
HashSet<string> setId = new HashSet<string> { "01", "02", "03", "04", "05" };
```

2. Sử dụng mảng để khởi tạo HashSet :

csharp

[Sao chép mã](#)

```
HashSet<int> mySet = new HashSet<int>(new int[] { 1, 2, 3 });
```



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

HASHSET

1

2

3

4

5

6

7

8

9

10

- Các thao tác CRUD với set
 - Set không thể truy xuất phần tử = index hoặc key như list hoặc dict vì vậy để update và read ta sẽ dùng 1 số cách như ví dụ bên dưới

```
C# Main.cs  X
app > C# Main.cs
1 // Tạo một HashSet ban đầu với các phần tử duy nhất
2 HashSet<string> names = new HashSet<string> { "Nga", "Khải", "Hằng", "Trinh", "Sang" };
3
4 // Create: Thêm phần tử vào HashSet
5 names.Add("Hải");
6 names.Add("Khải"); // "Khải" đã tồn tại nên sẽ không được thêm
7 foreach (var name in names)
8 {
9     Console.WriteLine(name);
10 }
```

```
14 // Read: Kiểm tra xem một phần tử có tồn tại không
15 bool exists = names.Contains("Hằng");
16 Console.WriteLine($"\\nHằng có tồn tại trong danh sách không? {exists}");
17 // Update: Xóa phần tử cũ và thêm phần tử mới
18 if (names.Contains("Hằng"))
19 {
20     names.Remove("Hằng");
21     names.Add("Tấn Khải");
22 }
23 Console.WriteLine("\\nDanh sách sau khi cập nhật:");
24 // Delete: Xóa phần tử khỏi HashSet
25 names.Remove("Khải");
26 Console.WriteLine("\\nDanh sách sau khi xóa Khải:");
27 foreach (var name in names)
28 {
29     Console.WriteLine(name);
30 }
31 // Clear: Xóa tất cả các phần tử trong HashSet
32 names.Clear();
33 Console.WriteLine("\\nDanh sách sau khi xóa toàn bộ:");
34 Console.WriteLine(names.Count == 0 ? "HashSet trống." : "HashSet vẫn còn phần tử.");
```



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

CHUYỂN ĐỔI GIỮA CÁC COLLECTION TYPE VỚI NHAU

- **1. Chuyển từ List sang Dictionary**

- Khi chuyển từ List sang Dictionary, bạn cần xác định khóa (key) và giá trị (value). Ví dụ, sử dụng chỉ mục của List làm khóa.

```
C# Main.cs  X  
app > C# Main.cs  
1  List<string> names = new List<string> { "Ngọc", "Khải", "An" };  
2  
3  // Chuyển từ List sang Dictionary  
4  Dictionary<int, string> dict = names.Select((name, index) => new { name, index })  
5  .ToDictionary(x => x.index, x => x.name);
```

- **3. Chuyển từ Dictionary sang List**

- Bạn có thể chuyển đổi cặp khóa-giá trị của Dictionary thành List dạng KeyValuePair.

```
C# Main.cs  X  
app > C# Main.cs  
1  Dictionary<int, string> dict = new Dictionary<int, string>  
2  {  
3      { 1, "Ngọc" },  
4      { 2, "Khải" },  
5      { 3, "An" }  
6  };  
7  
8  // Chuyển từ Dictionary sang List  
9  List<KeyValuePair<int, string>> list = dict.ToList();
```

- **2. Chuyển từ List sang HashSet**

- Đơn giản chuyển từ List sang HashSet, các phần tử trùng lặp sẽ tự động bị loại bỏ.

```
# Main.cs  X  
app > C# Main.cs  
1  List<string> names = new List<string> { "Ngọc", "Khải", "An", "Ngọc" };  
2  
3  // Chuyển từ List sang HashSet  
4  HashSet<string> nameSet = new HashSet<string>(names);
```

- **4. Chuyển từ HashSet sang List**

- Đơn giản chuyển từ HashSet sang List:

```
C# Main.cs  X  
app > C# Main.cs  
1  HashSet<string> nameSet = new HashSet<string> { "Ngọc", "Khải", "An" };  
2  
3  // Chuyển từ HashSet sang List  
4  List<string> nameList = nameSet.ToList();
```



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

CHUYỂN ĐỔI GIỮA CÁC COLLECTION TYPE VỚI NHAU

5. Chuyển từ Dictionary sang HashSet (Chỉ lấy giá trị)

Nếu bạn chỉ muốn lấy giá trị từ Dictionary và chuyển sang HashSet :

```
csharp
Sao chép mã

Dictionary<int, string> dict = new Dictionary<int, string>
{
    { 1, "Ngọc" },
    { 2, "Khải" },
    { 3, "An" }
};

// Chuyển từ Dictionary sang HashSet (lấy giá trị)
HashSet<string> nameSet = new HashSet<string>(dict.Values);
```

6. Chuyển từ HashSet sang Dictionary (Sử dụng giá trị làm khóa)

Nếu bạn muốn chuyển từ HashSet sang Dictionary, bạn có thể sử dụng chính các giá trị của HashSet làm khóa trong Dictionary .

```
csharp
Sao chép mã

HashSet<string> nameSet = new HashSet<string> { "Ngọc", "Khải", "An" };

// Chuyển từ HashSet sang Dictionary (giá trị làm khóa)
Dictionary<string, int> dict = nameSet.Select((name, index) => new { name, index })
    .ToDictionary(x => x.name, x => x.index);
```



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP SỬ DỤNG DICT LIST

LÀM LẠI BÀI NÀY VỚI GIẢI PHÁP SỬ DỤNG DICT

lst_number = [2, 7, 11, 15]

Mô tả: Tìm hai số trong một danh sách số nguyên sao cho tổng của chúng bằng một giá trị target cho trước.

- **Ví dụ:**

- **Input:**

- nums = [2, 7, 11, 15], target = 9

- **Output:**

- [0, 1] (vì nums[0] + nums[1] = 2 + 7 = 9) ngược lại nếu không có

1

2

3

4

5

6

7

8

9

10



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP SỬ DỤNG DICT LIST

1

2

3

4

5

6

7

8

9

10

LÀM LẠI BÀI NÀY VỚI GIẢI PHÁP SỬ DỤNG DICT

prices = [1,2,5,3,6,4]

Best Time to Buy and Sell Stock

Mô tả: Cho một mảng prices , mỗi phần tử của nó đại diện cho giá cổ phiếu trong một ngày. Bạn chỉ được mua cổ phiếu một lần và bán cổ phiếu một lần. Hãy tìm giá trị lớn nhất bạn có thể có từ việc mua và bán cổ phiếu.

- **Ví dụ:**
 - Input: prices = [7,1,5,3,6,4]
- **Output: 5**
 - Giải thích: Bạn mua vào ngày thứ 2 (giá 1) và bán vào ngày thứ 5 (giá 6), lãi là $6 - 1 = 5$.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

BÀI TẬP: SỬ DỤNG DICT + LIST + SET

1

2

3

4

5

6

7

8

9

10

ĐỀ BÀI: TÌM CHUỖI CON LIÊN TỤC DÀI NHẤT

nums = [100, 4, 200, 1, 3, 2 , 9 , 10 , 15, 14, 13, 12, 11]

Longest Consecutive

Mô tả: Chuỗi con liên tục dài nhất trong mảng là [1,2,3,4]. Lưu ý rằng chuỗi này có thể không xuất hiện theo thứ tự trong mảng.

- **Ví dụ:**

- **Input:**

- nums = [100, 4, 200, 1, 3, 2]

- **Output:**

- Độ dài chuỗi liên tục dài nhất là: 4

- **Giải thích:**

- Chuỗi con liên tục dài nhất trong mảng là [1, 2, 3, 4]. Lưu ý rằng chuỗi này có thể không xuất hiện theo thứ tự trong mảng.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

ARRAY & ARRAY LIST

- Trong C#, array là một collection type. Tuy nhiên, có một số sự khác biệt giữa array và các collection khác như List, Dictionary, hay HashSet, đặc biệt là trong cách chúng hoạt động và quản lý bộ nhớ.
- Array lưu trữ** một tập hợp các phần tử có **cùng kiểu dữ liệu**. Mỗi phần tử có thể được truy cập thông qua **chỉ mục** của nó (giống list).

```
string[] names = new string[5] { "A", "B", "C", "D", "E" };
```

- Các cách truy xuất phần tử tương tự List tuy nhiên có các điều cần lưu ý: Array không thể thêm hoặc xoá phần tử vì bộ nhớ được cấp phát là cố định (nếu xoá phần tử thì phần tử đó sẽ được gán về giá trị mặc định ví dụ int gán về 0, float gán về 0.0, string gán về "", ...)

Tóm tắt:

Array: Thích hợp cho dữ liệu cố định, hiệu suất cao, ít sử dụng bộ nhớ hơn và tương thích với các hệ thống cũ.

List<T>: Thích hợp cho dữ liệu thay đổi, linh hoạt hơn, cung cấp nhiều phương thức tiện lợi để thêm, xóa và tìm kiếm.

Tiêu chí	Array	List<T>
Kích thước	Cố định, phải biết trước khi khởi tạo.	Động, tự động mở rộng khi thêm phần tử.
Hiệu suất truy cập	O(1), truy cập phần tử thông qua chỉ mục nhanh và ổn định.	O(1), truy cập phần tử thông qua chỉ mục tương tự array.
Thêm/Xóa phần tử	Không thể thêm/xóa, phải tạo lại array mới khi thay đổi kích thước.	Thêm (Add) và xóa (RemoveAt) dễ dàng, tự động quản lý bộ nhớ.
Bộ nhớ	Quản lý bộ nhớ trực tiếp, hiệu quả hơn khi sử dụng cho dữ liệu cố định.	Quản lý bộ nhớ động, tiêu tốn bộ nhớ hơn khi mở rộng.
Mảng đa chiều	Hỗ trợ tốt mảng đa chiều (2D, 3D, v.v.).	Không hỗ trợ trực tiếp, phải sử dụng List<List<T>> để mô phỏng.
Khả năng tìm kiếm	Không có phương thức tích hợp sẵn, cần tự viết logic tìm kiếm.	Hỗ trợ sẵn các phương thức như Contains, Find, IndexOf.
Sắp xếp	Hỗ trợ Array.Sort() để sắp xếp các phần tử.	Hỗ trợ List<T>.Sort() để sắp xếp các phần tử.
Tính năng tiện ích	Rất ít tính năng tiện ích tích hợp, phải quản lý thủ công.	Nhiều phương thức tích hợp như Add, Remove, Insert, Sort.
An toàn kiểu dữ liệu	Chưa chắc chắn an toàn về kiểu nếu dùng array kiểu object, cần ép kiểu.	An toàn về kiểu dữ liệu nhờ generic, không cần ép kiểu.
Sử dụng khi	Khi biết trước kích thước của dữ liệu và cần hiệu suất tối ưu.	Khi kích thước dữ liệu có thể thay đổi hoặc cần thao tác linh hoạt.
Hiệu suất bộ nhớ	Tốt hơn cho dữ liệu cố định, vì không cần quản lý bộ nhớ động.	Sử dụng nhiều bộ nhớ hơn do quản lý động khi thêm/xóa phần tử.
Tương thích với API cũ	Được sử dụng rộng rãi trong các API và thư viện cũ.	List<T> thường không tương thích với các API hoặc thư viện cũ.
Lập trình hệ thống/thời gian thực	Phù hợp cho các ứng dụng nhúng và thời gian thực với yêu cầu bộ nhớ chặt chẽ.	Không phù hợp cho ứng dụng nhúng hoặc hệ thống thời gian thực do quản lý bộ nhớ động.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

1

2

3

4

5

6

7

8

9

10

ARRAY & ARRAY LIST

- ArrayList là một cấu trúc dữ liệu cũ trong C#, đã có từ trước khi List<T> xuất hiện. Dưới đây là bảng so sánh chi tiết giữa ArrayList, List<T>, và Array trong C#.
- So sánh chi tiết:**
 - ArrayList so với List<T>:**
 - ArrayList** là cấu trúc dữ liệu cũ và không an toàn kiểu dữ liệu. Mỗi phần tử được lưu trữ dưới dạng object, điều này yêu cầu bạn phải ép kiểu khi truy xuất các phần tử, dễ gây ra lỗi nếu không ép kiểu chính xác.
 - List<T>** là cấu trúc dữ liệu generic, giúp an toàn kiểu tại thời điểm biên dịch (compile-time). Với List<T>, bạn không cần ép kiểu khi truy cập các phần tử, điều này giúp tránh các lỗi về kiểu dữ liệu và tối ưu hiệu suất hơn.
 - Hiệu suất:**
 - ArrayList** có hiệu suất thấp hơn List<T> vì khi truy cập phần tử, bạn phải ép kiểu từ object về kiểu dữ liệu cụ thể. Điều này làm chậm quá trình thực thi và có thể gây ra ngoại lệ runtime nếu ép kiểu sai.
 - Tóm lại:**
 - Về nguyên lý hoạt động thì arrayList tương tự list nhưng hiệu quả hơn (Vì từ C# 2.0) List là 1 bản nâng cấp của arrayList khi bổ sung tính năng Generic type.

Tiêu chí	Array	ArrayList	List<T>
Kích thước	Cố định, phải biết trước khi khởi tạo.	Động, tự động mở rộng khi thêm phần tử.	Động, tự động mở rộng khi thêm phần tử.
Kiểu dữ liệu	Chỉ lưu trữ các phần tử có cùng kiểu dữ liệu.	Không an toàn kiểu, lưu trữ các phần tử dưới dạng object.	An toàn kiểu nhờ generic, lưu trữ các phần tử theo kiểu dữ liệu cụ thể (T).
Hiệu suất truy cập	O(1), truy cập nhanh và ổn định thông qua chỉ mục.	O(1), truy cập nhanh qua chỉ mục nhưng cần ép kiểu khi sử dụng.	O(1), truy cập nhanh qua chỉ mục, không cần ép kiểu.
Thêm/Xóa phần tử	Không thể, phải tạo array mới nếu muốn thêm/xóa phần tử.	Thêm (Add) và xóa (Remove) linh hoạt, nhưng cần ép kiểu khi truy cập phần tử.	Thêm (Add) và xóa (Remove) linh hoạt, an toàn kiểu, không cần ép kiểu.
Bộ nhớ	Sử dụng bộ nhớ tĩnh, không cấp phát lại khi thêm phần tử.	Quản lý bộ nhớ động, có thể tăng kích thước khi cần.	Quản lý bộ nhớ động, nhưng hiệu quả hơn ArrayList do sử dụng generic.
Tính năng tiện ích	Ít tính năng, chủ yếu dùng cho lưu trữ và truy cập phần tử.	Nhiều phương thức tiện lợi như Add , Remove , nhưng thiếu an toàn kiểu dữ liệu.	Nhiều phương thức tiện lợi như Add , Remove , Contains , an toàn kiểu dữ liệu.
An toàn kiểu dữ liệu	Chỉ lưu trữ các phần tử có cùng kiểu, an toàn kiểu.	Không an toàn kiểu, cần ép kiểu về đúng loại dữ liệu khi truy xuất phần tử.	An toàn kiểu dữ liệu nhờ generic, không cần ép kiểu.
Hiệu suất bộ nhớ	Hiệu suất cao cho dữ liệu cố định, không cần quản lý bộ nhớ động.	Hiệu suất kém hơn List<T> do sử dụng kiểu object và cần ép kiểu.	Hiệu suất tốt hơn ArrayList vì không cần ép kiểu và sử dụng bộ nhớ động hiệu quả hơn.
Tương thích với API cũ	Được sử dụng rộng rãi trong các API và thư viện cũ.	Tương thích với các API cũ vì tồn tại từ trước khi có List<T> .	Không tương thích với các API cũ, nhưng tốt hơn cho các dự án mới.
Tính năng tìm kiếm	Không hỗ trợ các phương thức tìm kiếm tiên lợi.	Hỗ trợ Contains , nhưng cần ép kiểu khi so sánh.	Hỗ trợ các phương thức tìm kiếm như Contains , Find , không cần ép kiểu.
Sử dụng khi	Khi biết trước kích thước dữ liệu và cần hiệu suất tối ưu.	Khi cần làm việc với nhiều kiểu dữ liệu khác nhau nhưng cần tương thích với API cũ.	Khi cần một danh sách động và muốn an toàn kiểu, sử dụng cho dữ liệu cụ thể.



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

COLLECTION EXTEND

1

2

3

4

5

6

7

8

9

10

Trước khi nói về các thành phần mở rộng của kiểu dữ liệu tập hợp ta sẽ nói về kiểu dữ liệu cơ sở (primitive type). Máy tính xử lý các kiểu dữ liệu nguyên thủy một cách nhanh chóng và trực tiếp vì chúng: có kích thước cố định, được tối ưu hoá ở cấp độ hệ điều hành và phần cứng, đơn giản và cơ bản

Primitive type bao gồm: **Integer**, **Float**, **Bolearn**, **char** hay **string**, **byte short** v...v... (ít sử dụng)

22fe53 = 'a'
22fe54 = 'b'
22fe55 = 'c'



a
22fe53
b
22fe54
c
22fe55



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

COLLECTION EXTEND

1

2

3

4

5

6

7

8

9

10



❖ Khai báo và lưu trữ biến



a = 10

system table

a



objects



CÁC KIỂU DỮ LIỆU TẬP HỢP (COLLECTION TYPES) CƠ BẢN THƯỜNG HAY SỬ DỤNG

COLLECTION EXTEND

COLLECTION EXTEND

Quay về 1 số dữ liệu collection type(List, dict, set, tuple) dùng để lưu các giá trị tham chiếu:

```
prod = new Dictionary<string, string>()
```

variable	Address	value

1

2

3

4

5

6

7

8

9

10



- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

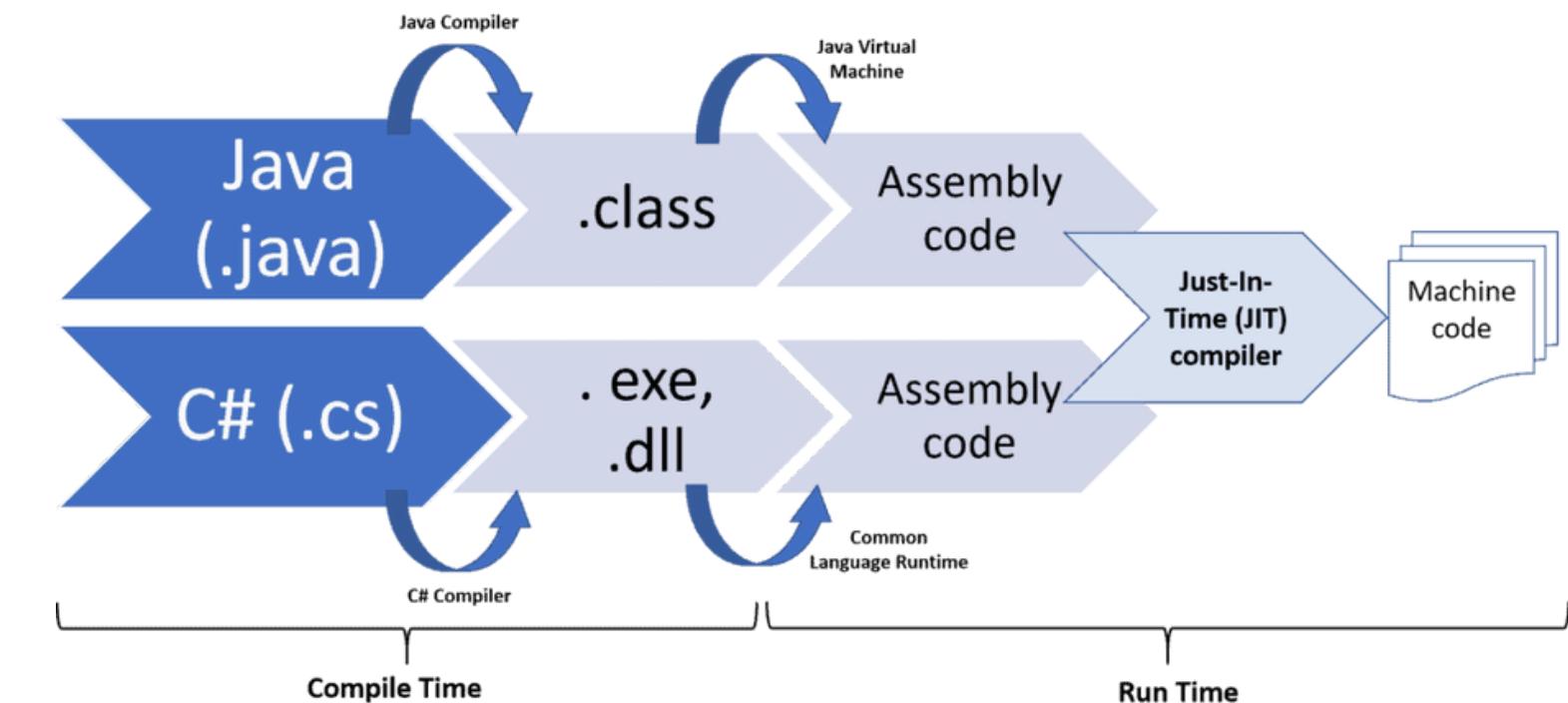
THỜI ĐIỂM XÁC ĐỊNH KIỂU (TYPE DETERMINATION)

Trong lập trình, thời điểm xác định kiểu (**type determination**) đề cập đến việc khi nào một ngôn ngữ lập trình xác định kiểu dữ liệu của một biến hoặc một biểu thức. Thời điểm này có thể xảy ra tại **thời điểm biên dịch (compile-time)** hoặc **thời điểm chạy (runtime)**. Cả hai thời điểm này có ảnh hưởng rất lớn đến cách chương trình được thực thi, hiệu suất và khả năng kiểm tra lỗi.

- **Có mấy thời điểm tổng cộng trong lập trình?**

- **Thời điểm biên dịch (Compile-time):** Khi mã nguồn được kiểm tra và biên dịch thành mã máy hoặc mã trung gian.
- **Thời điểm chạy (Runtime):** Khi chương trình đã được biên dịch và đang thực thi các lệnh của mã chương trình.

Đặc điểm	Thời điểm biên dịch (Compile-time)	Thời điểm chạy (Runtime)
Mục tiêu	Kiểm tra cú pháp, kiểu dữ liệu và biên dịch mã	Thực thi lệnh, quản lý bộ nhớ và xử lý lỗi
Lỗi có thể xảy ra	Lỗi cú pháp, lỗi kiểu dữ liệu, lỗi tham chiếu không xác định	Lỗi logic, lỗi runtime, lỗi ngoài tầm kiểm soát (như chia cho 0)
Phát hiện lỗi	Được phát hiện bởi trình biên dịch trước khi chương trình chạy	Chỉ được phát hiện khi chương trình đang chạy
An toàn kiểu dữ liệu	Đảm bảo an toàn vì kiểu dữ liệu được kiểm tra chặt chẽ	Không đảm bảo an toàn nếu có lỗi tại thời điểm chạy
Hiệu suất	Chương trình được tối ưu hóa trước khi chạy	Hiệu suất có thể bị ảnh hưởng nếu lỗi runtime xuất hiện
Ví dụ về ngôn ngữ/khai niệm	C#, Java, C++ (kiểm tra cú pháp trước khi chạy)	Python, JavaScript (kiểm tra kiểu dữ liệu tại runtime)





C# OBJECT & DYNAMIC TYPE & VAR

1

2

3

4

5

6

7

8

9

10

1. OBJECT TYPE

object là kiểu cơ bản nhất trong C#. Mọi kiểu dữ liệu trong C# đều kế thừa từ object. Điều này có nghĩa là bạn có thể gán bất kỳ giá trị nào cho một biến có kiểu object, nhưng khi bạn truy xuất giá trị đó, cần phải ép kiểu (type casting) về kiểu dữ liệu ban đầu.

- **Đặc điểm:**

- object là kiểu tham chiếu (reference type).
- Tất cả các kiểu dữ liệu trong C# đều kế thừa từ object.
- Khi lưu trữ kiểu giá trị (như int, bool, double, v.v.) trong một biến object, boxing xảy ra. Khi truy xuất, cần thực hiện unboxing (ép kiểu).

- **Khi nào dùng object?**

- Khi bạn cần lưu trữ nhiều kiểu dữ liệu khác nhau trong cùng một biến, và bạn sẵn sàng chấp nhận việc phải ép kiểu lại khi truy xuất dữ liệu.
- Ví dụ điển hình là các bộ sưu tập kiểu cũ như ArrayList trước khi có Generics.

The screenshot shows a code editor window with a tooltip for the 'object' keyword. The tooltip text is: "Tạo object chứa các kiểu dữ liệu khác nhau (local variable) object obj2". The code in the editor is:

```
C# Main.cs  ×  
app > C# Main.cs  
1 // Tạo object chứa các kiểu dữ liệu khác nhau  
2 object obj1;  
3 object obj2 = "Hello"; // string  
4 object obj3 = true; // bool  
5  
6 // Ép kiểu ngược lại và in ra kết quả  
7 int intValue = (int)obj1;  
8 string stringValue = (string)obj2;  
9 bool boolValue = (bool)obj3;  
10  
11 // In ra các giá trị  
12 Console.WriteLine($"Integer value: {intValue}");  
13 Console.WriteLine($"String value: {stringValue}");  
14 Console.WriteLine($"Boolean value: {boolValue}");
```



1

2

3

4

5

6

7

8

9

10

2. DYNAMIC TYPE

dynamic là một kiểu dữ liệu đặc biệt trong C#. Khi sử dụng dynamic, kiểu của biến sẽ được xác định tại thời điểm chạy (runtime), thay vì tại thời điểm biên dịch (compile-time). Trình biên dịch sẽ không kiểm tra tính đúng đắn của kiểu cho biến dynamic, do đó bạn có thể gán và truy cập giá trị của nó mà không cần ép kiểu.

- **Đặc điểm:**

- Kiểu dữ liệu của biến dynamic chỉ được xác định khi chương trình chạy.
- Không cần ép kiểu khi truy xuất giá trị từ dynamic.
- Tính năng kiểm tra kiểu không xảy ra khi biên dịch, dẫn đến nguy cơ lỗi chỉ xuất hiện khi chương trình chạy.

- **Khi nào dùng dynamic?**

- Khi bạn cần sự linh hoạt và không muốn trình biên dịch kiểm tra kiểu dữ liệu trước (ví dụ như khi làm việc với các API không xác định rõ kiểu trả về hoặc dữ liệu đến từ các nguồn không chắc chắn).
- Tuy nhiên, dynamic không được khuyến khích sử dụng nhiều, vì nó làm mất đi tính an toàn của kiểu dữ liệu (type safety) trong C#.

```
C# Main.cs 2 ×  
app > C# Main.cs  
1  dynamic dynamicVariable = 123;  
2  dynamicVariable = "Hello, World!";  
3  dynamicVariable = true;  
4  dynamicVariable = 10.5;  
5  dynamic result = dynamicVariable * 2;  
6  
7  Console.WriteLine($"Result: {result}, Type: {result.GetType()}");
```

Dùng dynamic để gán giá trị và xử lý

```
10 object objVariable = 123;  
11 objVariable = "Hello, World!";  
12 objVariable = true;  
13 objVariable = 10.5;  
14 object result = objVariable * 2; // Lỗi biên dịch: không thể thực hiện phép toán trên object object objVariable = 123;  
15  
16
```

Dùng object sẽ bị lỗi nếu không ép kiểu



C# OBJECT & DYNAMIC TYPE & VAR

1

2

3

4

5

6

7

8

9

10

3. VAR

var là một từ khóa cho phép khai báo biến mà không cần chỉ rõ kiểu. Trình biên dịch sẽ tự động xác định kiểu của biến dựa trên giá trị được gán cho nó tại thời điểm biên dịch (compile-time). Không giống như dynamic, với var, kiểu dữ liệu vẫn là tĩnh và phải được xác định tại thời điểm biên dịch.

- **Đặc điểm:**

- var phải được khởi tạo giá trị ngay tại lúc khai báo, vì trình biên dịch cần dựa vào giá trị để suy luận kiểu.
- Sau khi xác định kiểu của biến, biến đó có kiểu tĩnh và không thể thay đổi.
- var không phải là dynamic, trình biên dịch sẽ kiểm tra kiểu dữ liệu và báo lỗi nếu có sự không phù hợp.

- **Khi nào dùng var?**

- Khi bạn muốn trình biên dịch tự suy ra kiểu dữ liệu, đặc biệt hữu ích khi kiểu dữ liệu quá dài dòng hoặc phức tạp (như với Linq hoặc các kiểu dữ liệu nặc danh).
- Tuy nhiên, bạn vẫn cần biết chính xác kiểu dữ liệu mà var sẽ được suy luận, vì nó không hoạt động giống dynamic.

```
C# Main.cs 2 ×  
app > C# Main.cs  
1  dynamic dynamicVariable = 123;  
2  dynamicVariable = "Hello, World!";  
3  dynamicVariable = true;  
4  dynamicVariable = 10.5;  
5  dynamic result = dynamicVariable * 2;  
6  
7  Console.WriteLine($"Result: {result}, Type: {result.GetType()}");
```

Dùng dynamic để gán giá trị và xử lý

```
10 object objVariable = 123;  
11 objVariable = "Hello, World!";  
12 objVariable = true;  
13 objVariable = 10.5;  
14 object result = objVariable * 2; // Lỗi biên dịch: không thể thực hiện phép toán trên object object objVariable = 123;  
15  
16
```

Dùng object sẽ bị lỗi nếu không ép kiểu



1

2

3

4

5

6

7

8

9

10

4. SO SÁNH NHANH

- **object:**

- Là kiểu dữ liệu cơ bản trong C#, có thể chứa bất kỳ kiểu dữ liệu nào.
- Khi thực hiện các phép toán, bạn phải ép kiểu rõ ràng về kiểu dữ liệu ban đầu.
- Không an toàn kiểu tại thời điểm biên dịch, nhưng bạn sẽ nhận lỗi nếu thực hiện sai phép toán sau khi ép kiểu tại runtime.

- **dynamic:**

- Là kiểu xác định động tại thời điểm chạy (runtime). Kiểu của biến có thể thay đổi trong suốt quá trình thực thi chương trình.
- Không cần ép kiểu khi thực hiện phép toán, nhưng có thể gây lỗi khi chạy nếu kiểu không phù hợp.
- Không an toàn kiểu: Các lỗi liên quan đến kiểu chỉ được phát hiện tại thời điểm chạy chương trình.

- **var:**

- Là kiểu xác định tĩnh tại thời điểm biên dịch (compile-time). C# sẽ tự động suy luận kiểu của biến dựa trên giá trị gán ban đầu.
- Bạn không thể thay đổi kiểu của biến sau khi đã gán giá trị.
- Không cần ép kiểu khi thực hiện phép toán, vì kiểu đã được xác định rõ từ trước.
- An toàn kiểu (type-safe): Các lỗi kiểu dữ liệu sẽ được phát hiện tại thời điểm biên dịch.

So sánh nhanh:

Đặc điểm	object	dynamic	var
Thời điểm xác định kiểu	Compile-time	Runtime	Compile-time
Phải ép kiểu	Có (với kiểu giá trị)	Không	Không
Kiểm tra kiểu tại biên dịch	Có	Không	Có
Tính an toàn kiểu dữ liệu	An toàn, nhưng cần ép kiểu	Không an toàn	An toàn
Khi nào sử dụng	Khi cần lưu trữ nhiều loại dữ liệu khác nhau	Khi cần tính linh hoạt, làm việc với kiểu không xác định trước	Khi muốn trình biên dịch suy luận kiểu để đơn giản hóa mã

TRẮC NGHIỆM HỆ THỐNG

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10