Contents

[**C# PRACTICE NOTES** 2](#_Toc54092260)

[**The null coalescing operator** 2](#_Toc54092261)

[**The danger of virtual method calls from constructor.** 3](#_Toc54092262)

[**Partial types and methods.** 4](#_Toc54092263)

[**Runtime conversions with ConvertChangeType.** 4](#_Toc54092264)

[**Simplifying string emtpy and null checking code.** 5](#_Toc54092265)

[**Time-zones and using DateTimeMinValue to represent null dates.** 5](#_Toc54092266)

[**Testing char unicode validity** 5](#_Toc54092267)

[**Creating random numbers.** 6](#_Toc54092268)

[**Using tuples to reduce code.** 7](#_Toc54092269)

[**Don’t change an objects hashcode after adding to a dictionary.** 8](#_Toc54092270)

[**Creating and using combinable enums** 10](#_Toc54092271)

[**Conditional formationg for postivie, negative and zero number.** 10](#_Toc54092272)

[**Avoiding re-evaluation of LINQ queries.** 11](#_Toc54092273)

[**as or cast** 13](#_Toc54092274)

[**Replacing temporary collections with yield.** 13](#_Toc54092275)

[**Calling generic methods using implicit typing.** 17](#_Toc54092276)

[**Multicast delegates** 18](#_Toc54092277)

[**Compatibility and equality of delegate** 18](#_Toc54092278)

[**C# 7.0 FEATURES** 20](#_Toc54092279)

[**Why tuples?** 20](#_Toc54092280)

[**Pattern matching** 23](#_Toc54092281)

**[Out Variables](#_Toc54092282)** [26](#_Toc54092282)

**[Deconstruction](#_Toc54092283)** [26](#_Toc54092283)

[**Local Functions** 28](#_Toc54092284)

[**C# 8.0 FEATURES** 29](#_Toc54092285)

# **C# PRACTICE NOTES**

## **The null coalescing operator**

private static void Main(string[] args)

{

// Without using the null-coalescing operator:

string name = "Tim Nguyen";

string result = string.Empty;

if (name == null)

{

result = "No name provided";

}

else

{

result = name;

}

// with using the null-coalescing operator:

string result2 = name ?? "No name provided";

int? age = null;

// If age = null, then set age = 0

int resultAge = age ?? 0;

int? localDefaultAge = null;

int globalDefaultAge = 99;

int? age2 = null;

// If age2 is null

// Then check localDefaultAge is null or not

// If localDefaultAge is null

// Then get the value globalDefaultAge

// Else get the value of localDefaultAge

// Else get the value of age2

int resultAge2 = age2 ?? localDefaultAge ?? globalDefaultAge;

}

## **The danger of virtual method calls from constructor.**

public class BaseClass

{

private int \_length;

protected string Name;

public BaseClass()

{

InitName();

\_length = Name.Length;

}

protected virtual void InitName()

{

Name = "Sarah";

}

}

public class DerivedClass : BaseClass

{

protected override void InitName()

{

Name = null;

}

}

#region The danger of virtual method calls from constructors

// Không có lỗi

var testingClass = new BaseClass();

// Có lỗi ngay chỗ \_length = Name.Length, vì lúc này Name đang có giá trị là null.

var testingClass2 = new DerivedClass();

#endregion

## **Partial types and methods.**

namespace Partial\_Type

{

public class PartialTypeExample

{

// PartialTypeExample.cs

public void Example()

{

APartialType a = new APartialType();

a.SomeMethod();

a.SomeOtherMethod();

}

}

}

namespace Partial\_Type

{

// APartialType.cs

partial class APartialType

{

public void SomeOtherMethod() { }

// Nếu khai báo như vầy thì phải được định nghĩa sử dụng lại ở một file khác.

// Nếu không ta sẽ thấy lỗi

// No defining declaration found for implementing declaration of partial method

partial void APartialMethod() { }

}

}

namespace Partial\_Type

{

// APartialType.generated.cs

partial class APartialType

{

public void SomeMethod()

{

APartialMethod();

}

partial void APartialMethod();

}

}

## **Runtime conversions with ConvertChangeType.**

Type targetType;

Type convertedtype;

object convertedValue;

object initialValue;

initialValue = "99";

targetType = typeof(int);

convertedValue = Convert.ChangeType(initialValue, targetType);

// Result value: 99

convertedtype = convertedValue.GetType();

// Result type: int32

targetType = typeof(double);

// Change target type

convertedValue = Convert.ChangeType(initialValue, targetType);

// Result value: 99.00

convertedtype = convertedValue.GetType();

// Target type was changed: Double

## **Simplifying string emtpy and null checking code.**

var myString = " ";

bool isNullOrEmpty;

if(myString == null || myString == "")

{

isNullOrEmpty = true;

}

else

{

isNullOrEmpty = false;

}

// Thay vì vậy ta có thể viết

// Ngoài ra nếu trong string của ta có white space

// Thì ta nên dùng IsNullOrWhiteSpace

// Nếu dùng IsNullOrEmpty cho trường hợp này

// Thì ta sẽ nhận trả về là false

// (nó không nhận biết được white space)

if (string.IsNullOrEmpty(myString) )

{

isNullOrEmpty = true;

}

else

{

isNullOrEmpty = false;

}

## **Time-zones and using DateTimeMinValue to represent null dates.**

Vì Datetime là một kiểu struct nên ta không thể gán null cho nó, ta có thể sử dụng một kiểu nullable khác hoặc dùng MinValue để đại diện cho giá trị null.

Ta cần chú ý và rất cẩn thận nếu ta dùng MinValue để so sánh với LocalTime thì ta phải cẩn thận vì nó sẽ có sự sai khác về múi giờ (nó sẽ đúng nếu ta ở London).

DateTime minVal = DateTime.MinValue;

DateTime localMinVal = minVal.ToLocalTime();

// isMinVal will be true if LocalTime in London.

// isMinVal will be false if LocalTime not in London.

bool isMinVal = localMinVal == DateTime.MinValue;

## **Testing char unicode validity**

var validCharacter = 'q';

// Result: LowercaseLetter

var ucCategory = char.GetUnicodeCategory(validCharacter);

// Result: true

var isValidUnicode = ucCategory != System.Globalization.UnicodeCategory.OtherNotAssigned;

// Result: '\u0378'

var inValidCharacter = (char)888;

// Result: OtherNotAssigned

ucCategory = char.GetUnicodeCategory(inValidCharacter);

// Result: OtherNotAssigned

isValidUnicode = ucCategory != System.Globalization.UnicodeCategory.OtherNotAssigned;

## **Creating random numbers.**

class Program

{

// Bằng cách implement như vầy

// Thi giá trị của r1 và r2 sẽ giống nhau.

public static void Wrong()

{

var r1 = new Random();

var r2 = new Random();

Debug.WriteLine("r1 sequence");

for (int i = 0; i < 5; i++)

{

Debug.WriteLine(r1.Next());

}

Debug.WriteLine("r2 sequence");

for (int i = 0; i < 5; i++)

{

Debug.WriteLine(r2.Next());

}

}

// Nếu ta sử dụng random number ở nhiều nơi

// Ta nên có một static random instance.

public static void Better()

{

var r1 = new Random();

Debug.WriteLine("r1 sequence");

for (int i = 0; i < 5; i++)

{

Debug.WriteLine(r1.Next());

}

Debug.WriteLine("more r1 sequence");

for (int i = 0; i < 5; i++)

{

Debug.WriteLine(r1.Next());

}

}

// For việc tạo dãy ngẫu nhiên mà cho mã hóa

// Ta nên sử dụng như sau

public void HighSecurity()

{

var r = System.Security.Cryptography.RandomNumberGenerator.Create();

// 4 bytes so we can convert to an 32 bit int.

var randomBytes = new byte[4];

r.GetBytes(randomBytes);

int rnInt = BitConverter.ToInt32(randomBytes,0);

Debug.WriteLine(rnInt);

}

static void Main(string[] args)

{

}

}

## **Using tuples to reduce code.**

public static void CreatingTuples()

{

var tupleOneElement = new Tuple<int>(1);

var tupleTwoElements =

new Tuple<int, string>(1,"Hello");

var tupleSevenElements =

new Tuple<int, int, int, int, int, int, int>(1,2,3,4,5,6,7);

var tupleEightElements

= new Tuple<int, int, int, int, int, int, Tuple<string>,

Tuple<string>>(1, 2, 3, 4, 5, 6, new Tuple<string>("Hello"), new Tuple<string>("Hello"));

var tupleThreeElements = Tuple.Create(43, "Hello",DateTime.Now );

}

// Comparing tuples

public static void ComparingTuples()

{

var t1 = Tuple.Create(43, "hello");

var t2 = Tuple.Create(43, "hello");

//Reference equality

var isEqualTuples = t1 == t2;

// "value" comparison

isEqualTuples = t1.Equals(t2);

}

// Using tuples to return multiple values

public static void UsingTuplesToReturnMultipleValues()

{

var boxingOpponents = GetOpponentNames();

var opponent1 = boxingOpponents.Item1;

var opponent2 = boxingOpponents.Item2;

}

// Using tuples as compound dictionary keys

private static void UsingTupleAsCompoundDictionaryKeys()

{

var t1 = Tuple.Create(1, "z");

var t2 = Tuple.Create(2,"a");

var t3 = Tuple.Create(3, "a");

var d = new SortedDictionary<Tuple<int, string>, string>();

d.Add(t1,"Tuple t1");

d.Add(t2, "Tuple t2");

d.Add(t3, "Tuple t3");

foreach (var item in d)

{

Debug.WriteLine(item);

}

}

private static Tuple<string, string> GetOpponentNames()

{

return new Tuple<string, string>("Stive", "Fred");

}

private static void Main(string[] args)

{

var t = Tuple.Create(24,"hello");

int aget = t.Item1;

string greeting = t.Item2;

//t.Item2 = 99; // compliler error, Tuples are immutable.

}

## **Don’t change an objects hashcode after adding to a dictionary.**

public class PersonIdBadHash

{

// Id ở đây ta đang để là public

// và nó có thể được gọi lại sau khi nó được tạo.

// và đó là issue khi ta override bằng GetHashCode.

public int Id { get; set; }

public override int GetHashCode()

{

return Id.GetHashCode();

}

}

public class PersonIdBetterHash

{

// Ở đây, ta cho set của Id là private

// và do vậy nó chỉ được set trong nội bộ class

// và ta implement việc set đó ở constructor của class.

public PersonIdBetterHash(int id)

{

Id = id;

}

public int Id { get; private set; }

public override int GetHashCode()

{

return Id.GetHashCode();

}

}

public static void BadGetHashCodeExample()

{

PersonIdBadHash p1 = new PersonIdBadHash { Id = 1 };

PersonIdBadHash p2 = new PersonIdBadHash { Id = 2 };

Dictionary<PersonIdBadHash, string> d

= new Dictionary<PersonIdBadHash, string>

{

{ p1, "Sarah" },

{ p2, "John" }

};

string john = d[p2];

p2.Id = 99;

// Ta sẽ bị lỗi ngay dòng bên dưới

// Compiler báo:

// System.Collections.Generic.KeyNotFoundException:

// 'The given key was not present in the dictionary.'

john = d[p2];

}

public static void BetterGetHashCodeExample()

{

var p1 = new PersonIdBetterHash(1);

var p2 = new PersonIdBetterHash(2);

var d = new Dictionary<PersonIdBetterHash, string>();

d.Add(p1,"Sarah");

d.Add(p2, "John");

var john = d[p2];

}

private static void Main(string[] args)

{

//BadGetHashCodeExample(); => throw an error

BetterGetHashCodeExample(); // compliler will happy.

}

## **Creating and using combinable enums**

public class FlagsEnumExample

{

[Flags]

enum Alignments

{

None = 0,

Top = 1,

Right = 2,

Bottom = 4,

Left = 8

}

public void Example()

{

// Result: Top | Right

var topRightCombination = Alignments.Top | Alignments.Right;

// Result: Bottom | Left

var bottomLeftCombination = Alignments.Bottom | Alignments.Left;

// Result: true

var isTopIncluded = (topRightCombination & Alignments.Top) != 0;

isTopIncluded = topRightCombination.HasFlag(Alignments.Top);

// Combining combinations

// Result: Top|Right|Bottom|Left

var all = topRightCombination | bottomLeftCombination;

// Toggling values

// Result: Right|Bottom|Left

all ^= Alignments.Top;

// Result: Top|Right|Bottom|Left

all ^= Alignments.Top;

}

}

## **Conditional formationg for postivie, negative and zero number.**

class Program

{

const double aPostiveNumber = 99.99;

const double aNegativeNumber = -33.33;

const double aZeroNumber = 0;

const string threePartFormat = "(+)#.##;(-)#.##;(sorry nothing at all)";

static void Main(string[] args)

{

var postiveOutput = aPostiveNumber.ToString(threePartFormat);

var negativeOutput = aNegativeNumber.ToString(threePartFormat);

var zeroOutput = aZeroNumber.ToString(threePartFormat);

// Result: (+)99.99

Debug.WriteLine(postiveOutput);

// Result: (-)33.33

Debug.WriteLine(negativeOutput);

// Result: (sorry nothing at all)

Debug.WriteLine(zeroOutput);

}

}

## **Avoiding re-evaluation of LINQ queries.**

public static void WithReEvalutation()

{

List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7 };

var q = (from n in nums

select new

{

Number = n,

ExecutionTime = DateTime.Now.ToString("mm:ss:fff")

});

//Result:

//1st enumeration

//{ Number = 1, ExecutionTime = 43:54:016 }

//{ Number = 2, ExecutionTime = 43:54:019 }

//{ Number = 3, ExecutionTime = 43:54:019 }

//{ Number = 4, ExecutionTime = 43:54:019 }

//{ Number = 5, ExecutionTime = 43:54:020 }

//{ Number = 6, ExecutionTime = 43:54:020 }

//{ Number = 7, ExecutionTime = 43:54:022 }

Debug.WriteLine("1st enumeration");

foreach (var n in q)

{

Debug.WriteLine(n);

}

//Result:

//2st enumeration

//{ Number = 1, ExecutionTime = 43:54:024 }

//{ Number = 2, ExecutionTime = 43:54:025 }

//{ Number = 3, ExecutionTime = 43:54:025 }

//{ Number = 4, ExecutionTime = 43:54:025 }

//{ Number = 5, ExecutionTime = 43:54:026 }

//{ Number = 6, ExecutionTime = 43:54:026 }

//{ Number = 7, ExecutionTime = 43:54:026 }

Debug.WriteLine("2st enumeration");

foreach (var n in q)

{

Debug.WriteLine(n);

}

}

public static void WithoutReEvalution()

{

List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7 };

var q = (from n in nums

select new

{

Number = n,

ExecutionTime = DateTime.Now.ToString("mm:ss:fff")

}).ToList();

// Ở trên, ngay khi gặp ToList (hoặc ToArray, ToDictionary) thì sẽ thực thi câu query

// và do vậy ở 2 vòng loops bên dưới

// sẽ duyệt cùng một tập dữ liệu.

Debug.WriteLine("1st enumeration");

foreach (var n in q)

{

Debug.WriteLine(n);

}

Debug.WriteLine("2st enumeration");

foreach (var n in q)

{

Debug.WriteLine(n);

}

}

## **as or cast**

Ta cùng cast khi ta biết rõ kiểu dữ liệu chuyển dỗi và kiểu dữ liệu cần được chuyển đổi, nếu mà 2 đứa này không tương thích với nhau nó sẽ dục cục error vô mặt liền.

Nếu ta không biết rõ về các kiểu dữ liệu sẽ được chuyển đổi hay ta muốn nếu việc chuyển đổi khoog thành công thì sẽ có một cái xử lý default cho việc này thì ta Dùng as, nhưng phải kèm theo việc check null exception, nếu không sẽ bị văng cục lỗi NullPointExeption vô mặt.

public class Person

{

public int Id { get; set; }

public string Name { get; set; }

}

class Program

{

public static void Cast()

{

object name = "Sarah";

Person p = (Person)name;

Debug.WriteLine(p.Name);

// Báo lỗi kiểu không hợp lệ.

}

public static void AsBad()

{

object name = "Sarah";

Person p = name as Person;

Debug.WriteLine(p.Name);

// Error

// Báo lỗi về Null exception

// Trong khi đó nó nên là lỗi về kiểu dữ liệu

}

public static void AsBetter()

{

object name = "Sarah";

Person p = name as Person;

if (p == null)

{

p = new Person { Name = "Default" };

}

Debug.WriteLine(p.Name);

}

static void Main(string[] args)

{

Cast();

}

}

## **Replacing temporary collections with yield.**

internal class Program

{

public static IList<int> GetEvenNumbers(IEnumerable<int> numbers)

{

List<int> evens = new List<int>();

foreach (int number in numbers)

{

Debug.WriteLine("GetEvenNumbers: Processing input number: " + number);

if (number % 2 == 0)

{

evens.Add(number);

}

}

return evens;

}

public static IEnumerable<int>GetEvenNumbersUsingYield(IEnumerable<int> numbers)

{

foreach (var number in numbers)

{

Debug.WriteLine("GetEvenNumberUsingYield: Processing input number "+number);

if (number % 2 == 0)

{

Debug.WriteLine("GetEvenNumbersUsingYeld: yealding: "+number);

yield return number;

}

}

}

public static void WithYield()

{

var someNumbers = Enumerable.Range(1,10);

var evens = GetEvenNumbersUsingYield(someNumbers);

foreach (var even in evens)

{

Debug.WriteLine("Output: "+even);

//if (even == 6)

//{

// break;

//}

}

}

public static void WithoutYield()

{

IEnumerable<int> someNumbers = Enumerable.Range(1, 10);

IList<int> evens = GetEvenNumbers(someNumbers);

foreach (int even in evens)

{

Debug.WriteLine("Output: " + even);

//if (even == 6)

//{

// break;

//}

}

}

public static void ComposingWithYield()

{

var someNumbers = Enumerable.Range(1,10);

var twoAndSix = GetTwoAnd6(GetEvenNumbersUsingYield(someNumbers));

foreach (var number in twoAndSix)

{

Debug.WriteLine("Output: "+number);

}

}

public static IEnumerable<int>GetTwoAnd6(IEnumerable<int> numbers)

{

foreach (var number in numbers)

{

Debug.WriteLine("GetTwoAnd6: Processing input number: " +number);

if(number ==2 || number == 6)

{

Debug.WriteLine("GetTwoAnd6: yealding: "+number);

yield return number;

}

}

}

private static void Main(string[] args)

{

#region WithoutYield

WithoutYield();

// Result: Khi đã comment line code if (even == 6)

//GetEvenNumbers: Processing input number: 1

//GetEvenNumbers: Processing input number: 2

//GetEvenNumbers: Processing input number: 3

//GetEvenNumbers: Processing input number: 4

//GetEvenNumbers: Processing input number: 5

//GetEvenNumbers: Processing input number: 6

//GetEvenNumbers: Processing input number: 7

//GetEvenNumbers: Processing input number: 8

//GetEvenNumbers: Processing input number: 9

//GetEvenNumbers: Processing input number: 10

//Output: 2

//Output: 4

//Output: 6

//Output: 8

//Output: 10

// Result: Khi đã uncomment line code if (even == 6)

//GetEvenNumbers: Processing input number: 1

//GetEvenNumbers: Processing input number: 2

//GetEvenNumbers: Processing input number: 3

//GetEvenNumbers: Processing input number: 4

//GetEvenNumbers: Processing input number: 5

//GetEvenNumbers: Processing input number: 6

//GetEvenNumbers: Processing input number: 7

//GetEvenNumbers: Processing input number: 8

//GetEvenNumbers: Processing input number: 9

//GetEvenNumbers: Processing input number: 10

//Output: 2

//Output: 4

//Output: 6

#endregion

#region WithYield

WithYield();

// Result: Khi đã comment line code if (even == 6)

//GetEvenNumberUsingYield: Processing input number 1

//GetEvenNumberUsingYield: Processing input number 2

//GetEvenNumbersUsingYeld: yealding: 2

//Output: 2

//GetEvenNumberUsingYield: Processing input number 3

//GetEvenNumberUsingYield: Processing input number 4

//GetEvenNumbersUsingYeld: yealding: 4

//Output: 4

//GetEvenNumberUsingYield: Processing input number 5

//GetEvenNumberUsingYield: Processing input number 6

//GetEvenNumbersUsingYeld: yealding: 6

//Output: 6

//GetEvenNumberUsingYield: Processing input number 7

//GetEvenNumberUsingYield: Processing input number 8

//GetEvenNumbersUsingYeld: yealding: 8

//Output: 8

//GetEvenNumberUsingYield: Processing input number 9

//GetEvenNumberUsingYield: Processing input number 10

//GetEvenNumbersUsingYeld: yealding: 10

//Output: 10

// Result: khi đã uncomment line code if (even == 6)

//GetEvenNumberUsingYield: Processing input number 1

//GetEvenNumberUsingYield: Processing input number 2

//GetEvenNumbersUsingYeld: yealding: 2

//Output: 2

//GetEvenNumberUsingYield: Processing input number 3

//GetEvenNumberUsingYield: Processing input number 4

//GetEvenNumbersUsingYeld: yealding: 4

//Output: 4

//GetEvenNumberUsingYield: Processing input number 5

//GetEvenNumberUsingYield: Processing input number 6

//GetEvenNumbersUsingYeld: yealding: 6

//Output: 6

#endregion

//=> Từ 2 ví dụ có sử dụng yield => PERFORMANCE PROBLEMS.

ComposingWithYield();

// Result:

//GetEvenNumberUsingYield: Processing input number 1

//GetEvenNumberUsingYield: Processing input number 2

//GetEvenNumbersUsingYeld: yealding: 2

//GetTwoAnd6: Processing input number: 2

//GetTwoAnd6: yealding: 2

//Output: 2

//GetEvenNumberUsingYield: Processing input number 3

//GetEvenNumberUsingYield: Processing input number 4

//GetEvenNumbersUsingYeld: yealding: 4

//GetTwoAnd6: Processing input number: 4

//GetEvenNumberUsingYield: Processing input number 5

//GetEvenNumberUsingYield: Processing input number 6

//GetEvenNumbersUsingYeld: yealding: 6

//GetTwoAnd6: Processing input number: 6

//GetTwoAnd6: yealding: 6

//Output: 6

//GetEvenNumberUsingYield: Processing input number 7

//GetEvenNumberUsingYield: Processing input number 8

//GetEvenNumbersUsingYeld: yealding: 8

//GetTwoAnd6: Processing input number: 8

//GetEvenNumberUsingYield: Processing input number 9

//GetEvenNumberUsingYield: Processing input number 10

//GetEvenNumbersUsingYeld: yealding: 10

//GetTwoAnd6: Processing input number: 10

}

}

## **Calling generic methods using implicit typing.**

Để tránh redundant code khi sử dụng generic method ta xem ví dụ dưới đây

internal class Program

{

private static void Main(string[] args)

{

Example();

}

public static void Example()

{

const string name = "Sarah";

const int age = 42;

WriteToDebug<string>(name);

WriteToDebug<int>(age);

// Ta cũng có thể làm như sau

WriteToDebug(name);

WriteToDebug(age);

// Điều cho kết quả giống nhau

}

private static void WriteToDebug<T>(T obj)

{

Debug.WriteLine(obj);

}

}

Về việc overload generic method ta xem xét ví dụ dưới đây..

internal class Program

{

private static void Main(string[] args)

{

Example();

}

public static void Example()

{

const string name = "Sarah";

const int age = 42;

WriteToDebug(name);

// Sẽ gọi tới hàm WriteToDebug(string s)

// Nếu ta muốn nó phải gọi tới hàm WriteToDebug<T>(T obj)

// Thì ta phải tường minh ra là muốn gọi đứa nào

// WriteToDebug<string>(name);

WriteToDebug(age);

// Sẽ gọi tới hàm WriteToDebug<T>(T obj)

}

private static void WriteToDebug<T>(T obj)

{

Debug.WriteLine(obj);

}

private static void WriteToDebug(string s)

{

Debug.WriteLine(s);

}

}

## **Multicast delegates**

class Program

{

private delegate void ProcessChangeNotifier(int percent);

void WriteToDebug(int percent)

{

Debug.WriteLine(percent);

}

void WriteToDebugWithMessage(int percent)

{

Debug.WriteLine("Process now at: "+percent);

}

public void MulticastExample()

{

var processDelegate = new ProcessChangeNotifier(WriteToDebug);

Debug.WriteLine("Invoking delegate with a single target method assigned");

processDelegate(50);

// Result: 50

// To add another delegate method

processDelegate += WriteToDebugWithMessage;

Debug.WriteLine("Involking delegate with two target methods assigned");

processDelegate(50);

// Result lần lượt là

// 50

// Progress now at: 50

// To remove a delegate method named WriteToDebug.

processDelegate -= WriteToDebug;

Debug.WriteLine("Invoking delegate with first target metho removed");

processDelegate(50);

// Result: Progress not at: 50

}

// Demo for Multi cast return values.

private delegate int MathOperation(int a, int b);

private int Add(int a, int b)

{

Debug.WriteLine("Add called");

return a + b;

}

private int Multiply(int a, int b)

{

Debug.WriteLine("Multiply called");

return a \* b;

}

public void MulticastReturnValues()

{

var mathDelegate = new MathOperation(Add);

mathDelegate += Multiply;

var result = mathDelegate(10, 10);

// Result: 100

// Mặc dù delegate tới method Add vẫn được gọi

// Nhưng kết quả trả về thì nó chỉ trả về kết của của

// delegate cuối cùng mà thôi.

}

}

## **Compatibility and equality of delegate**

class Program

{

private delegate void ADelegateWithAnInt(int i);

private delegate void AnotherDelegateWithAnInt(int i);

void WriteToDebug(int percent)

{

}

void WriteToDebugWithMessage(int percent)

{

}

public void CampatibilityDelegateTypes()

{

//ADelegateWithAnInt a = new ADelegateWithAnInt(WriteToDebug);

ADelegateWithAnInt a = WriteToDebug;

AnotherDelegateWithAnInt b;

b = a;

// Result: Compiler error

// Cannot implicity convert type.

b = (AnotherDelegateWithAnInt)a;

// Result: Compiler error

// Cannot implicity convert type.

b = new AnotherDelegateWithAnInt(a);

// Result: compiler is happy

// There is no error.

}

public void EqualityOfDelegateInstances()

{

ADelegateWithAnInt a = WriteToDebug;

ADelegateWithAnInt b = WriteToDebug;

ADelegateWithAnInt c = WriteToDebugWithMessage;

var isInstanceAEqualToB = a == b;

// Result: true

var isInstanceAEqualToC = a == c;

// Result: false

}

static void Main(string[] args)

{

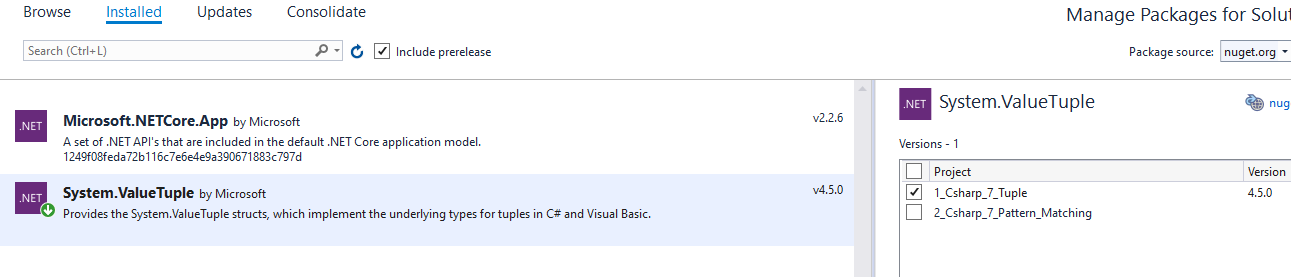
}

}

# **C# 7.0 FEATURES**

## **Why tuples?**

* Want to return more than one value from method.
* Existing options available to developers are not optimal
* Introduced tuple types and tuple literals.
* Structs – value types
* Created locally and passed by copying contents
* Are mutable.



public class Chapter1

{

public (int average, int studentCount) GetAverageAndCount(int[] scores)

{

// Khai báo thông thường

// var returnTuple = (ave:0,sCount:0);

// (int ave, int sCount) returnTuple;

//Khai báo theo kiểu rút gọn luôn.

(int ave, int sCount) returnTuple = (scores.Sum() / scores.Count(), scores.Count());

return returnTuple;

}

public (double average, int studentCount, bool belowAverage) GetAverageAndCount(int[] scores, int threshold)

{

var returnTuple = (ave: 0D, sCount: 0, subAve: true);

returnTuple = ((double)scores.Sum() / scores.Count(),

scores.Count(),

returnTuple.ave.checkIfBelowEverage(threshold));

return returnTuple;

}

}

public static class ExtensionMethods

{

public static bool checkIfBelowEverage(this double classAverage, int threshold)

{

if (classAverage < threshold)

return true;

else

return false;

}

}

class Program

{

static void Main(string[] args)

{

// Demo 1

int[] scores = { 2, 2 };

Chapter1 ch1 = new Chapter1();

var (average, studentCount) = ch1.GetAverageAndCount(scores);

WriteLine($"Average was {average} across {studentCount}");

ReadLine();

// Demo 2

int[] scores = { 2, 2, 2, 2, 3 };

Chapter1 ch1 = new Chapter1();

int threshold = 51;

var (average, studentCount, belowAverage) = ch1.GetAverageAndCount(scores, threshold);

WriteLine(

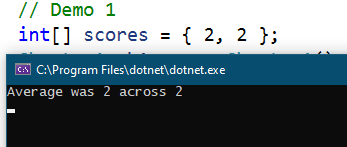
$"Average was {Round(average, 2)} across {studentCount} students. {(average < threshold ? " Class score below average." : " Class score above average.")}");

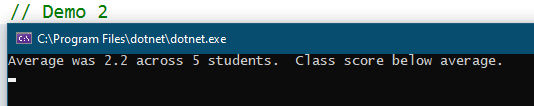
ReadLine();

}

}

Kết quả sẽ lần lượt là





## **Pattern matching**

* Can test values in different ways
* Two language constructs have bene enhanced
* They are
  + Is expression
  + Case clause in switch statements.
* Have pattern on right instead of type.
* Cause clause can now match on patterns
* No longer limited to primitive types.

namespace \_2\_Csharp\_7\_Pattern\_Matching

{

public class Chapter2

{

public void OutputInformation(object person)

{

// Nếu ta chỉ khai báo theo kiểu cũ như sau:

// person is Student

// thì ta cần phải tạo ra một đối tượng sinh viên từ việc ép kiểu một person trước đó.

// Điều này cũng tương tự cho Professor.

if (person is Student student)

{

//Student student = (Student) person;

WriteLine($"Student {student.Name} {student.LastName} is enrolled for courses {String.Join<int>(", ", student.CourseCodes)}");

}

if (person is Professor prof)

{

//Professor prof = (Professor) person;

WriteLine($"Professor {prof.Name} {prof.LastName} teaches {String.Join<string>(",", prof.TeachesSubjects)}");

}

}

public void OutputInformation2(object person)

{

switch (person)

{

case Student student when (student.CourseCodes.Contains(2)):

WriteLine($"Student {student.Name} {student.LastName} is enrolled for course 2.");

break;

case Student student:

WriteLine($"Student {student.Name} {student.LastName} is enrolled for course {string.Join<int>(",", student.CourseCodes)}");

break;

case Professor prof:

WriteLine($"Professor {prof.Name} {prof.LastName} teaches {string.Join<string>(",",prof.TeachesSubjects)}");

break;

case null:

WriteLine($"Object {nameof(person) is null}");

break;

default:

WriteLine("Unknown object detected");

break;

}

}

}

public class Student

{

public string Name { get; set; }

public string LastName { get; set; }

public List<int> CourseCodes { get; set; }

}

public class Professor

{

public string Name { get; set; }

public string LastName { get; set; }

public List<string> TeachesSubjects { get; set; }

}

}

namespace \_2\_Csharp\_7\_Pattern\_Matching

{

// SCENARIO

// Two object types:

// Student

// Professor

// Minimize code

// Create single method to output data from object passed to it

// Method needs to figure out which object it is work.

class Program

{

static void Main(string[] args)

{

Chapter2 ch2 = new Chapter2();

Student student = new Student();

student.Name = "Dirk";

student.LastName = "Strauss";

student.CourseCodes = new List<int>{3,2,5};

ch2.OutputInformation(student);

Professor prof = new Professor();

prof.Name = "Reinhardt";

prof.LastName = "Botha";

prof.TeachesSubjects = new List<string>{"A","B"};

ch2.OutputInformation(prof);

ReadLine();

}

}

}

## **Out Variables**

TryParse

* Method that tests to see if a value parse to specific type.
* If yes – returns a boolean value of true.

// Đoạn code sau sẽ giúp ta kiểm tra

// xem liệu rằng sValue có phải là một

// string có thể chuyển thành kiểu int

// được hay không, khi ta sử dụng TryParse

// sẽ không văng exception khi không convert được

// mà thay vào đó nó chỉ trả về là false thôi.

int intVal;

if (int.TryParse(sValue, out intVal))

{

    WriteLine($"{intVal} is a valid integer");

    // Do something with intVal

}

// Ngoài ra ta còn có thể xem thêm

// Các cách sử dụng của TryParse như bên dưới

#region New out variable implementation

if (int.TryParse(sValue, out int intVal))

{

    WriteLine($"{intVal} is a valid integer");

    // Do something with intVal

}

if (int.TryParse(sValue, out var intVal))

{

    WriteLine($"{intVal} is a valid integer");

    // Do something with intVal

}

var (original, intVal, isInteger) = sValue.ToInt();

if (isInteger)

{

    WriteLine($"{original} is a valid integer");

    // Do something with intVal

}

#endregion

## **Deconstruction**

* Tuples can be consumed
* Simply splits tuple into individual parts
* Assigns parts to new variables
* Not only reserved for tuples
* Ensure that type has deconstructor method

Để hiểu được cách sử dụng ta xét đoạn code sau:

public class Student

{

    public string Name { get; set; }

    public string LastName { get; set; }

    public List<int> CourseCodes { get; set; }

    // Định nghĩa Deconstruct

    public void Deconstruct(out string name, out string lastName)

    {

        name = Name;

        lastName = LastName;

    }

}

class Program

{

    static void Main(string[] args)

    {

        Student student = new Student();

        student.Name = "Tim";

        student.LastName = "Nguyen";

        // Gọi ra sử dụng deconstuct

        var (FirstName, Surname) = student;

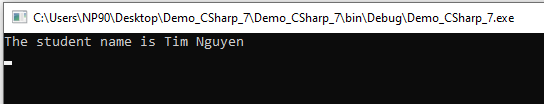
        WriteLine($"The student name is {FirstName} {Surname}");

        ReadLine();

    }

}

Khi ta chạy chương trình ta sẽ được kết quả như sau:



Deconstuction còn dễ dàng được sử dụng thông ma một tín năng của C# gọi là Extension method như sau:

public class Student

{

    public string Name { get; set; }

    public string LastName { get; set; }

    public List<int> CourseCodes { get; set; }

}

public static class ExtensionMethodClass

{

    public static void Deconstruct(

        this Student student,

        out string firstItem,

        out string secondItem)

    {

        firstItem = student.Name;

        secondItem = student.LastName;

    }

}

class Program

{

    static void Main(string[] args)

    {

        Student student = new Student();

        student.Name = "Tim";

        student.LastName = "Nguyen";

        // Gọi ra sử dụng deconstuct

        var (FirstName, Surname) = student;

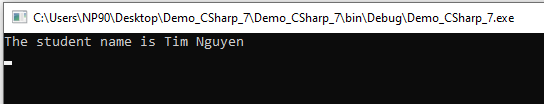
        WriteLine($"The student name is {FirstName} {Surname}");

        ReadLine();

    }

}

Khi ta chạy đoạn chương trình mà implement theo cách extension method ta cũng nhận được kết quả tương tự



## **Local Functions**

* Often used in most functional languages
* Helper method for specific method
* Makes sense when used from specific method
* Not be useful for ther methods in application
* Use it inside existing method.

# **C# 7.0 through C# 7.3 FEATURES**

## **Tuples and discards**

Tuples đã có được từ trước khi C# 7.0 ra đời, nhưng lúc đó tuple không mang lại nhiều lợi ích, và bạn chỉ có thể gọi đến nó bằng cách truy cập đến Item1, Item2 và nó cho phép bạn đối đa có 8 Items. Ở C# 7.0 ra đời thì tuple được làm cho mạnh mẽ hơn nhiều ta sét ví dụ dưới đây.

**Chương trình 1:**

public static void Demo1()

{

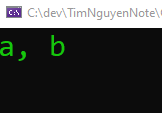
(string Alpha, string Beta) namedLetters = ("a", "b");

WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");

ReadLine();

}

Thì khi thực thi chương trình kết quả sẽ là



**Chương trình 2:**

private static int Plus(int a, int b)

{

return (a + b);

}

private static int Subtract(int a, int b)

{

return (a - b);

}

public static void Demo2()

{

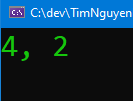
(int plusResult, int substractResult) resultObject = (Plus(2,2),Subtract(5,3));

WriteLine($"{resultObject.plusResult}, {resultObject.substractResult}");

ReadLine();

}

Khi thực thi chương trình ta sẽ được kết quả như sau:



**Chương trình 3:**

Ngoài ra bạn có thể thực hiện khai báo nhanh như sau cũng cho kết quả tương tự

public static void Demo3()

{

var alphabetStart = (Alpha: "a", Beta: "b");

Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");

}

**Chương trình 4:**

Khi bạn thực hiện lấy một kết quả là một kiểu là tuple của một method trả về lồi dùng kết quả làm gì gì đó thì nó gọi là deconstructing tuple.

Ta xét đoạn chương trình sau:

private static (int max, int min) Range(int[]numbers)

{

(int max, int min) resultTuple = (numbers.Max(),numbers.Min());

return resultTuple;

}

/// <summary>

/// Deconstructing the tuple

/// </summary>

public static void Demo4()

{

var numbers = new[] {2, 3, 4, 5, 1};

(int max, int min) = Range(numbers);

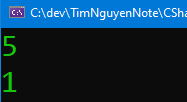
WriteLine(max);

WriteLine(min);

ReadLine();

}

Ta sẽ nhận được kết quả trả về như sau:



**Chương trình 5:**

Bạn có thể thực hiện việc deconstruction đối với bất kỳ loại dữ liệu nào trong .NET. Việc phải làm là bạn viết một method Deconstruct như là một member của class đó. Cái method Deconstruct đó sẽ cũng cấp tham chiếu (out) tới những properties nào mà bạn muốn extract.

public class Point

{

public Point(double x, double y)

=> (X, Y) = (x, y);

// Thật ra bạn có thể hoàn toàn viết lại cho tường minh như sau:

//public Point(double x, double y)

//{

// (X, Y) = (x, y);

//}

public double X { get; }

public double Y { get; }

public void Deconstruct(out double x, out double y) =>

(x, y) = (X, Y);

// Bạn có thể viết lại tường minh như sau:

//public void Deconstruct(out double x, out double y)

//{

// (x, y) = (X, Y);

//}

}

public static void Demo5()

{

var p = new Point(3.14, 2.71);

(double X, double Y) = p;

WriteLine("Before change: ");

WriteLine(X);

WriteLine(Y);

WriteLine("After change: ");

X = 2;

Y = 1;

WriteLine(X);

WriteLine(Y);

WriteLine("Object Point after deconstruction: ");

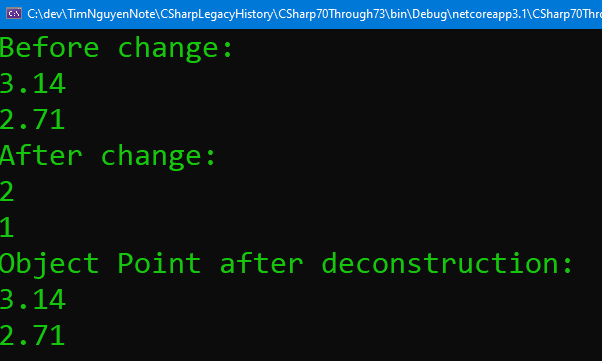
WriteLine(p.X);

WriteLine(p.Y);

ReadLine();

}

Khi thực thi chương trình ta sẽ được kết quả như sau:



**Chương trình 6**

Theo mặc định nếu bạn không chỉ ra các tên của các phần tử trong tuple thì nó sẽ tự lấy tên biến của bạn gán vào làm tên phần tử đó, ta xem sét ví dụ sau:

public static void Demo6()

{

int count = 5;

string label = "Colors used in the map";

var pair = (count, label);

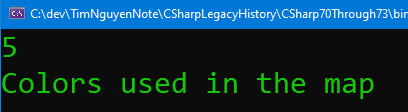
WriteLine(pair.count);

WriteLine(pair.label);

ReadLine();

}

Kết quả khi thực thi chương trình



Chương trình 7

Đôi khi bạn nhận về một kết quả là một tuple từ một method nào đó và trong tuple có rất nhiều những key trong đó, nhưng bạn chỉ quan tâm tới một vài key mà thôi. Thấu hiểu được điều đó ta có sự ra đời của discard (kí tự “ \_”).

Ta xét đoạn chương trình sau:

Trong đoạn chương trình method QueryCityDataForYears sẽ trả về một cái cục tuple có hằm bà lằng các thứ ta chỉ qua tâm item thứ 4 và item cuối cùng của tuple này do vậy ta specific nó ra còn mấy thằng ông nội kia ta méo quan tâm nên tả bỏ “\_” vào cho nó phải phép.

private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int year2)

{

int population1 = 0, population2 = 0;

double area = 0;

if (name == "New York City")

{

area = 468.48;

if (year1 == 1960)

{

population1 = 7781984;

}

if (year2 == 2010)

{

population2 = 8175133;

}

return (name, area, year1, population1, year2, population2);

}

return ("", 0, 0, 0, 0, 0);

}

public static void Demo7()

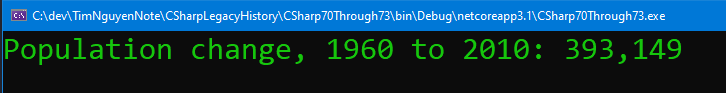
{

var (\_, \_, \_, pop1, \_, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

}

Kết quả trả về như sau:



## **Pattern matching**

## **Async main**

## **Local functions**

## **More expression-bodied members**

## **Throw expressions**

## **Default literal expressions**

## **Numeric literal syntax improvements**

## **out variables**

## **Non-trailing named arguments**

## **private protected access modifier**

## **Improved overload candidates**

## **Enabling more efficient safe code**

## **New compiler options**