C# 2.0

1. Generics
2. Nullable types
3. Iterators
4. yield Statement
5. Partial Classes/Structs/Interfaces
6. Anonymous methods
7. The :: operator (Namespace Alias Qualifier)
8. Static Classes
9. Covariance and contravariance
10. Fixed-size buffers

Note: fixed keyword [only with built-in struct except decimal, only in unsafe, cannot be static]

1. Friend assemblies
2. Extern Aliases [extern keyword has been available since C# 1.0]
3. Delegate Method Group Conversion
4. Access Modifiers with Accessors [Note: For properties and indexers]
5. #pragma directives
6. ?? Operator or Null-Coalesce Operator
7. System.Predicate
8. System.Action

//12. Extern Aliases [extern keyword has been available since C# 1.0]

//An extern alias declaration must precede all other elements defined in the namespace

extern alias ExternAssembly1; //Right-click ExternalAssemblyA in References (of CSC2010CS2) and click Properties to set the Aliases

extern alias ExternAssembly2; //Right-click ExternalAssemblyB in References (of CSC2010CS2) and click Properties to set the Aliases

using System;

using System.Collections;

using N1;

using N2;

// Give N1 an alias called N.

//7. The :: operator (Namespace Alias Qualifier)

using N = N1;

//11. Friend assemblies

//using SignedFriend; //Commented to reference unsigned assemblies //sgKeyCalculator.snk key excluded from CSC2010CS2 project

class G

{

object obj;

public G(object obj)

{

this.obj = obj;

}

public object GetObject()

{

return obj;

}

public void ShowType()

{

Console.WriteLine("Type: " + obj.GetType());

}

}

//1. Generics

class G<T>

{

T obj;

public G(T obj)

{

this.obj = obj;

}

public T GetObject()

{

return obj;

}

public void ShowType()

{

Console.WriteLine("Type: " + typeof(T));

}

}

class GClient

{

public void Print()

{

G g = new G("Hello World!");

string s = (string)g.GetObject();

Console.WriteLine("Object: " + s);

g.ShowType();

g = new G(100);

int i = (int)g.GetObject();

Console.WriteLine("Object: " + i);

g.ShowType();

G<string> gs = new G<string>("Hello World!");

s = gs.GetObject();

Console.WriteLine("Object: " + s);

gs.ShowType();

G<int> gi = new G<int>(100);

i = gi.GetObject();

Console.WriteLine("Object: " + i);

gi.ShowType();

}

}

struct StuctureType

{

}

//15. New #pragma directives

/\*

The first is warning, which is used to enable or disable specific compiler warnings:

#pragma warning disable [warnings]

#pragma warning restore [warnings]

The second #pragma option is checksum. It is used to generate checksums for ASP.NET projects.

It has this general form:

#pragma checksum “fi lename” “{GUID}” “check-sum”

Here, filename is the name of the file, GUID is the globally unique identifier associated with

filename, and check-sum is a hexadecimal number that contains the checksum. This string

must contain an even number of digits.

\*/

#pragma warning disable 0168, 0169, 3021 //3021 for CLSCompliant

[CLSCompliant(false)]

class NullableTypes

{

//2. Nullable Types

//System.Nullable<[ValueType]> or Nullable<[ValueType]> or [ValueType]?

//Can be member or local variable

//Defaulted to null as member variable

//Can be static

//warning 0169

System.Nullable<StuctureType> NullableStucture; //Can be struct

sbyte? NullableSByte;

byte? NullableByte;

char? NullableChar;

short? NullableShort;

ushort? NullableUShort;

System.Nullable<int> NullableInteger;

uint? NullableUInt;

long? NullableLong;

ulong? NullableULong;

float? NullableFloat;

double? NullableDouble;

decimal? NullableDecimal;

bool? NullableBool;

public void Print()

{

//2. Nullable Types

//warning 0168

StuctureType? NullableStuct;

sbyte? NullableSByteLocal;

byte? NullableByteLocal;

char? NullableCharLocal;

short? NullableShortLocal;

ushort? NullableUShortLocal;

System.Nullable<int> NullableInt = null;

uint? NullableUIntLocal;

long? NullableLongLocal;

ulong? NullableULongLocal;

float? NullableFloatLocal;

double? NullableDoubleLocal;

decimal? NullableDecimalLocal;

bool? NullableBoolLocal;

if (NullableInt == null) //Since NullableInt is assigned although with null

{

Console.WriteLine("NullableInt is null.");

}

NullableInt = NullableInteger;

if (NullableInt == null)

{

Console.WriteLine("NullableInt is still null.");

}

//16. ?? Operator or Null-Coalesce Operator

Console.WriteLine("NullableInt is defaulted to {0}", NullableInt ?? 1);

NullableInteger = NullableInt ?? default(int);

Random RandomNumber = new Random();

int Number = RandomNumber.Next(0, 100);

//Pre-C# 2.0 conditional operator (?:)

Console.WriteLine("{0} is an {1}", Number, (Number % 2 == 0) ? "Even Number" : "Odd Number");

if (NullableInteger != null)

{

Console.WriteLine("NullableInteger is not null.");

if (NullableInteger.HasValue)

{

Console.WriteLine("NullableInteger has value: {0}", NullableInteger.Value);

}

}

if (NullableStucture == null)

{

Console.WriteLine("NullableStucture is null");

}

NullableStucture = new StuctureType();

if (NullableStucture != null)

{

Console.WriteLine("NullableStucture is not null");

}

if (NullableStucture.HasValue)

{

Console.WriteLine("NullableStucture has value: {0}", NullableStucture.Value);

}

}

}

//3. Iterators

class Iterators

{

public IEnumerator GetEnumerator()

{

//Less than 100

for (int i = 0; i < 100; i++)

{

//Less than 20

if (i == 20) yield break;

{

//Even Number

if (i % 2 == 0)

{

yield return i;

}

}

}

}

public IEnumerable Power(int number, int exponent)

{

int count = 0;

int result = 1;

while (count++ < exponent)

{

//if (count == 3) yield break; //Up to exponent 2

result \*= number;

yield return result;

}

}

}

class IteratorsClient

{

public void Print()

{

Iterators Itr = new Iterators();

Console.WriteLine("Iterators:");

foreach (int i in Itr)

{

Console.Write(i + " ");

}

Console.WriteLine();

Console.WriteLine("Iterators:");

foreach (int i in Itr.Power(2, 8))

{

Console.Write(i + " ");

}

Console.WriteLine();

}

}

//Pre-C# 2.0 IEnumerable, IEnumerator

class Enumerators : IEnumerable, IEnumerator

{

//C# 4.0 dynamic

//dynamic Dyn = new dynamic[] { 0, true, "two", null, DateTime.Now };

dynamic[] Dyn = { 0, true, "two", null, DateTime.Now };

int Index = -1;

public IEnumerator GetEnumerator()

{

return this;

}

public bool MoveNext()

{

if (Index == Dyn.Length - 1)

{

Reset();

return false;

}

Index++;

return true;

}

public object Current

{

get

{

return Dyn[Index];

}

}

public void Reset()

{

Index = -1;

}

}

class EnumeratorsClient

{

public void Print()

{

Enumerators Enumtr = new Enumerators();

Console.WriteLine("Pre-C# 2.0 IEnumerable, IEnumerator:");

foreach (dynamic d in Enumtr)

{

Console.Write(d + " ");

}

Console.WriteLine();

//Note

string s = "Hello World!"; //The String class implements IEnumerable

Console.WriteLine("The String class implements IEnumerable:");

foreach (char c in s)

{

Console.Write(c + ".");

}

Console.WriteLine();

}

}

//4. yield Statement

class YieldStatement

{

char[] chrs = { 'A', 'B', 'C', 'D' };

public IEnumerator GetEnumerator()

{

foreach (char ch in chrs)

{

yield return ch;

}

}

}

class YieldClient

{

public void Print()

{

YieldStatement ys = new YieldStatement();

foreach (char ch in ys)

{

Console.Write(ch + " ");

}

Console.WriteLine();

}

}

//5. Partial Classes/Structs/Interfaces

//It is possible to split the definition of a class or a struct, or an interface over two or more source files.

//Each source file contains a section of the type, and all parts are combined when the application is compiled.

partial class PartialType //public partial struct P // All should be structs

{

private int x;

private int y;

public PartialType(int x, int y) //Note: Constructor cannot be partial

{

this.x = x;

this.y = y;

}

}

// This part can be in a separate file.

partial class PartialType //public partial struct P // All should be structs

{

public void Print()

{

Console.WriteLine("x = {0}, y = {1}", x, y);

}

}

//5. Partial Classes/Structs/Interfaces

partial interface IPartial

{

void Medthod();

}

// This part can be in a separate file.

partial interface IPartial

{

//Must be different or overloaded

void Medthod(string s);

}

class ImplementingType : IPartial //partial class ImplementingType : IPartial //partial struct ImplementingType : IPartial

{

public void Medthod()

{

Console.Write("Partial ");

}

public void Medthod(string s)

{

Console.WriteLine(s);

}

public void Print()

{

Medthod();

Medthod("Type!");

}

}

delegate int Delegates(int i);

class AnonymousMethods

{

public void Print()

{

//6. Anonymous methods

Delegates DelegateObject = delegate(int number)

{

int Sum = 0;

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

{

Console.Write(i + " ");

Sum += i;

}

return Sum;

}; //Note

int Result = DelegateObject(3);

Console.WriteLine("Summation: " + Result);

}

}

namespace N1

{

class C

{

int i;

public C(int i)

{

this.i = i;

Console.WriteLine("Number passed to Namespace N1 is " + i);

}

}

}

namespace N2

{

//Same class as in N1

class C

{

int i;

public C(int i)

{

this.i = i;

Console.WriteLine("Number passed to Namespace N2 is " + i);

}

}

}

class NamespaceAliasQualifier

{

public void Print()

{

//7. The :: operator (Namespace Alias Qualifier)

// Here, the :: operator (Namespace Alias Qualifier) tells

// the compiler to use the class C

// that is in the N1 namespace.

N::C c1 = new N::C(10);

//Note:

N2.C c2 = new N2.C(20);

}

}

//8. static classes

static class StaticClass

{

public static int i;

//Access modifiers are not allowed on static constructors

//A static constructor must be parameterless

static StaticClass()

{

i = 100;

}

public static void Print()

{

Console.WriteLine("static class member variable value: " + i);

}

}

class BaseClass

{

}

class DerivedClass : BaseClass

{

}

//9. Covariance and contravariance

delegate BaseClass Covariance();

delegate void Contravariance(DerivedClass DC);

class Variance

{

DerivedClass CovarianceMethod()

{

DerivedClass DC = new DerivedClass();

Console.WriteLine("Covariance Method: " + DC.GetType());

return DC;

}

void ContravarianceMethod(BaseClass BC)

{

Console.WriteLine("Contravariance Method: " + BC.GetType());

}

public void Print()

{

//13. Delegate Method Group Conversion

Covariance Co = CovarianceMethod; //Covariance Co = new Covariance(CovarianceMethod);

Co += CovarianceMethod; //Note

Co -= CovarianceMethod; //Note

Contravariance Contra = ContravarianceMethod; //Contravariance Contra = new Contravariance(ContravarianceMethod);

Co();

DerivedClass DC = new DerivedClass();

Contra(DC); //Note

}

}

//10. Fixed-size buffers [Note: Available only in an unsafe context]

/\*

\* To set this compiler option in the Visual Studio development environment

\* 1. Open the project's Properties page.

\* 2.Click the Build property page.

\* 3. Select the Allow Unsafe Code check box.

\*/

// Create a fixed-size buffer.

unsafe struct FixedBankRecord

{

public fixed byte name[80]; // create a fixed-size buffer

public double balance;

public long ID;

}

class Properties

{

string PropertyName;

//14. Access Modifiers with Accessors [Note: For properties and indexers]

//Accessibility modifiers on accessors may only be used if the property or indexer has both a get and a set accessor

//Cannot specify accessibility modifiers for both accessors of the property or indexer

//The accessibility modifier of the get or set accessor must be more restrictive than the property or indexer

//Event accessor declarations are available in pre-C# 2.0. Modifiers cannot be placed on event accessor declarations in C# at all

internal protected string Property

{

internal get

{

return PropertyName;

}

set

{

PropertyName = value;

}

}

public void Print()

{

Property = "Access Modifiers only with either Accessor of the property";

Console.WriteLine(Property);

}

}

class FailSoftArray

{

int[] IntegerArray; // reference to underlying array

public int Length; // Length is public

public bool ErrorFlag; // indicates outcome of last operation

// Construct array given its size.

public FailSoftArray(int size)

{

IntegerArray = new int[size];

Length = size;

}

// This is the indexer for FailSoftArray.

public int this[int Index]

{

// This is the get accessor.

get

{

if (Okay(Index))

{

ErrorFlag = false;

return IntegerArray[Index];

}

else

{

ErrorFlag = true;

return 0;

}

}

//14. Access Modifiers with Accessors [Note: For properties and indexers]

// This is the set accessor

internal set

{

if (Okay(Index))

{

IntegerArray[Index] = value;

ErrorFlag = false;

}

else

{

ErrorFlag = true;

}

}

}

// Return true if index is within bounds.

private bool Okay(int Index)

{

if (Index >= 0 & Index < Length) return true;

return false;

}

}

class FailSoftClient

{

public void Print()

{

FailSoftArray FSA = new FailSoftArray(5);

int Number;

for (int i = 0; i < (FSA.Length \* 2); i++)

{

FSA[i] = i \* 10;

}

Console.WriteLine("Fail quietly:");

for (int i = 0; i < (FSA.Length \* 2); i++)

{

Number = FSA[i];

if (Number != -1)

{

Console.Write(Number + " ");

}

}

Console.WriteLine();

Console.WriteLine("Fail with error reports:");

for (int i = 0; i < (FSA.Length \* 2); i++)

{

Number = FSA[i];

if (!FSA.ErrorFlag)

{

Console.Write(Number + " ");

}

else

{

Console.WriteLine("FSA[" + i + "] out-of-bounds");

}

}

}

}

//17. System.Predicate

/\*

C# 2.0 introduced a new feature: the predicate. A predicate is a delegate of type

System.Predicate that returns either true or false, based upon some condition.

The object to be tested against the condition is passed in obj. If obj satisfies that condition,

the predicate must return true. Otherwise, it must return false. Predicates are used by

several new methods added to Array by C# 2.0, including Exists( ), Find( ), FindIndex( ), and FindAll().

The following program demonstrates using a predicate to determine if an array of integers

contains a negative value. If a negative value is found, the program then obtains the first

negative value in the array. To accomplish this, the program uses Exists( ) and Find( ).

\*/

class Predicates

{

// A predicate method.

// Returns true if Value is negative.

bool IsNegative(int Value)

{

if (Value < 0)

{

return true;

}

return false;

}

public void Print()

{

int[] Numbers = { 1, 4, -1, 5, -9 };

Console.Write("Contents of numbers: ");

foreach (int i in Numbers)

{

Console.Write(i + " ");

}

Console.WriteLine();

// First see if nums contains a negative value.

if (Array.Exists(Numbers, IsNegative))

{

Console.WriteLine("Numbers contains a negative value.");

// Now, find first negative value.

int x = Array.Find(Numbers, IsNegative);

Console.WriteLine("First negative value is: " + x);

}

else

{

Console.WriteLine("Numbers contains no negative values.");

}

}

}

class Number

{

public int Integer;

public Number(int Integer)

{

this.Integer = Integer;

}

}

//18. System.Action

/\*

Another new feature introduced by C# 2.0 is the System.Action delegate. An Action is used

by another new C# feature, Array.ForEach( ), to perform an action on each element of an array.

The object to be acted upon is passed in obj. When used with ForEach( ), each element of the

array is passed to obj in turn. Thus, through the use of ForEach( ) and Action, you can, in a

single statement, perform an operation over an entire array.

The following program demonstrates both ForEach( ) and Action. It first creates an

array of MyClass objects, then uses the method show( ) to display the values. Next, it uses

neg( ) to negate the values. Finally, it uses show( ) again to display the negated values.

These operations all occur through calls to ForEach( ).

\*/

class Actions

{

// An Action method.

// Displays the value it is passed.

void Show(Number N)

{

Console.Write(N.Integer + " ");

}

// Another Action method.

// Negates the value it is passed.

void Negate(Number N)

{

N.Integer = -N.Integer;

}

public void Print()

{

Number[] Numbers = new Number[5];

Numbers[0] = new Number(5);

Numbers[1] = new Number(4);

Numbers[2] = new Number(3);

Numbers[3] = new Number(2);

Numbers[4] = new Number(1);

Console.Write("Contents of numbers: ");

// Use action to show the values.

Array.ForEach(Numbers, Show);

Console.WriteLine();

// Use action to negate the values.

Array.ForEach(Numbers, Negate);

Console.Write("Contents of numbers negated: ");

// Use action to negate the values again.

Array.ForEach(Numbers, Show);

Console.WriteLine();

}

}

class MainApp

{

//10. Fixed-size buffers [Note: Available only in an unsafe context]

// mark Main as unsafe

unsafe static void Main()

{

GClient GClnt = new GClient();

GClnt.Print();

NullableTypes NT = new NullableTypes();

NT.Print();

IteratorsClient IC = new IteratorsClient();

IC.Print();

EnumeratorsClient EC = new EnumeratorsClient();

EC.Print();

PartialType PT = new PartialType(1, 2);

PT.Print();

ImplementingType IT = new ImplementingType();

IT.Print();

AnonymousMethods AM = new AnonymousMethods();

AM.Print();

NamespaceAliasQualifier NAQ = new NamespaceAliasQualifier();

NAQ.Print();

StaticClass.Print();

Variance Varnce = new Variance();

Varnce.Print();

FailSoftClient FSC = new FailSoftClient();

FSC.Print();

//10. Fixed-size buffers [Note: Available only in an unsafe context]

Console.WriteLine("Size of FixedBankRecord is " + sizeof(FixedBankRecord));

Console.WriteLine("C# 2.0 sizeof(double): " + sizeof(double));

Console.WriteLine("C# 2.0 sizeof(long): " + sizeof(long));

//11. Friend assemblies

//Calculator Calc = new Calculator();

//Console.WriteLine("Using Friend Assembly, {0} power {1} = {2}", 2, 8, Calc.Power(2, 8));

//12. Extern Aliases [extern keyword has been available since C# 1.0]

ExternAssembly1::ExternAssembly.ExternClass EC1 = new ExternAssembly1::ExternAssembly.ExternClass();

ExternAssembly2::ExternAssembly.ExternClass EC2 = new ExternAssembly2::ExternAssembly.ExternClass();

Predicates Predct = new Predicates();

Predct.Print();

Actions Actn = new Actions();

Actn.Print();

Console.ReadKey();

}

}