C#入門

目次

- タスク
- async と await
- ポインタ
- パッケージ
- using文
- 部分クラス(partial)
- globalエイリアス
- 外部関数(extern)
- 変数名参照(nameof)
- クエリ(LINQ)
- in・ref・out修飾子
- readonly修飾子
- sizeof演算子
- stackalloc式
- volatileキーワード
- dynamic型
- whenキーワード
- ジェネリック型制約(where, unmanaged)
- リンク

C#とは

- 「シーシャープ」と呼びます。
- Microsoft が開発したプログラミング言語です。
- ECMA-334、ISO/IEC 23270 として規格化・公開されています。
- Turbo Pascal、Delphiを開発したアンダース・ヘルスバーグが設計しました。
- C言語、C++、Java、Delphi の影響を受けています。
- 基本的には、Windows の .NET Framework 上で開発します。
- Mono(.NET Framework互換環境)をインストールすれば、Linux でもコンパイルできます。
- int や double などのプリミティブな型もすべて object 型の派生として定義されます。
- 演算子をオーバーロードすることができます。

本書は、あからじめ、**JavaScript**、**C++**、**Java** などのプログラミング言語をある程度習得している方を対象に、C# についての概要をさらりと説明します。

インストール

Windows の場合は、Microsoft の 公式サイト から **Visual Studio 2019** をダウンロードしてインストールします。個人・教育機関・オープンソース開発目的であれば、無償の Community 版をダウンロードすることができます。

macOS や **Linux** の場合は、**.NET Framework** の互換ソフトウェアである **Mono** をインストールします。Mono Download に、macOS, Ubuntu, Debian, Raspbian, CentOS, Red Hat, Fedra のインストール手順が紹介されています。Docker コンテナイメージも公開されているようです。

Hello world

おなじみの Hello world は次のようになります。拡張子は ***.cs** です。

```
using System;

class MainClass
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

これを **csc** コマンドでコンパイルします。コンパイルの度に表示される Microsoft のコピーライトを表示したくない場合は **/nologo** オプションをつけます。コンパイルすると hello.exe ファイルができます。Windows の場合は直接実行なファイルになります。中間言語にコンパイルするので、Linux でコンパイルした *.exe を Windows で動かすこともできます。

```
# コンパイルする
csc /nologo hello.cs

# Windowsの場合
hello.exe

# Linuxの場合
mono hello.exe
```

コメント

```
// この部分はコメントになる

/* この部分も
   コメントになる */
```

キーワード

abstract	add	alias	as
ascending	async	await	base
bool	break	by	byte
case	catch	char	checked
class	const	continue	decimal
default	delegate	descending	do
double	dynamic	else	enum
equals	event	explicit	extern
false	finally	fixed	float
for	foreach	from	get
global	goto	group	if
implicit	in(for)	in(修飾子)	in(LINQ)
int	interface	internal	into
is	join	let	lock
long	nameof	namespace	new
null	object	on	operator
orderby	out	override	params
partial	private	protected	public
readonly	ref	return	remove
sbyte	sealed	select	set
short	sizeof	stackalloc	static
string	struct	switch	this
throw	true	try	typeof
uint	ulong	unchecked	unmanaged

unsafe	ushort	using	value
var	virtual	void	volatile
when	where(LINQ)	where	while
yield			

型

下記などの型を使用することができます。下記の型はすべて構造体の別名として定義されています。例えば、**int** は **System.Int32** 構造体の別名です。

```
bool            // 真偽値(true or false)
sbyte           // 8ビット整数
byte            // 8ビット非負整数
short           // 16ビット整数
ushort          // 16ビット非負整数
int             // 32ビット整数
uint            // 32ビット非負整数
long            // 64ビット整数
ulong           // 64ビット非負整数
char            // 16ビットUnicode文字(U+0000～U+FFFF)
float           // 32ビット単精度浮動小数点数(IEEE754)
double          // 32ビット単精度浮動小数点数(IEEE754)
decimal         // 128ビット数値(1.0e10^-128～7.9e10^28)
string          // 文字列型
object          // オブジェクト
void            // 値を返さないという型
```

変数・定数

「型名 変数名」で変数を定義します。初期値を指定することもできます。

```
int a1;                 // 型 変数名
int a2 = 123;           // 型 変数名 = 初期値
```

初期値により型が明らかな場合は型名の代わりに **var** を使用することができます。

```
var a3 = 123;           // var 変数名 = 初期値
```

const は固定値を宣言します。

```
const int C4 = 123;     // const 型 定数名 = 初期値
```

列挙型(enum)

enum は列挙型を定義します。

```
enum Color
{
    Red,
    Green,
    Blue
}
```

列挙型のメンバーは int 型で 0 から順に数値が割り振られますが、型や値を指定することもできます。

```
enum Color : ushort
{
    Red = 10,
    Green = 20,
    Blue = 30
}
```

配列

型名[] を用いて配列を表します。配列個数はあらかじめ決めておく必要があります。

```
int[] a = new int[3];
a[0] = 100;
a[1] = 200;
a[2] = 300;
```

配列の個数を動的に変更するには **List**<型名> を用います。

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

class MyClass
{
    static void Main()
    {
        var list = new List<string>();
        list.Add("Red");
        list.Add("Green");
        list.Add("Blue");
        Console.WriteLine(list[2]);
    }
}
```

ヌル値(null)

null は値が存在しないことを示す特別な値です。**string** や **object** は null値を持つことができます。

```
string str = null;
object obj = null;
```

int などその他のプリミティブな型は通常null値を持つことができませんが、**int?** の様に **?** をつけることで、null値を許容することができます。

```
int? num = null;
```

演算子

```
x + y           // 加算
x - y           // 減算
x * y           // 乗算
x / y           // 除算
x % y           // 剰余
x++             // インクリメント
x--             // デクリメント
+x              // 単項プラス演算子
```

```
-x              // 単項マイナス演算子
!x              // 論理否定
x & y           // 論理積(AND)
x | y           // 論理和(OR)
x ^ y           // 排他的論理和(XOR)
x && y          // xもyも真であれば真
x || y          // xまたはyが真であれば真
~x              // 補数
x << y          // 左シフト
x >> y          // 右シフト
x & y           // 論理積(AND)
x | y           // 論理和(OR)
x ^ y           // 排他的論理和(XOR)
x == y          // xとyが等しければ真
x != y          // xとyが等しくなければ真
x < y           // xがyより小さければ真
x > y           // xがyより大きければ真
x <= y          // xがy以下であれば真
x >= y          // xがy以上であれば真
x += y          // x = x + yと等価
x -= y          // x = x - yと等価
x *= y          // x = x * yと等価
x /= y          // x = x / yと等価
x %= y          // x = x % yと等価
x &= y          // x = x & yと等価
x |= y          // x = x | yと等価
x ^= y          // x = x ^ yと等価
x <<= y         // x = x << yと等価
x >>= y         // x = x >> yと等価
x.y             // xのメンバーyにアクセス
x[y]            // 配列xのy番目(0ベース)の要素にアクセス
x[^y]           // 配列xの最後からy番目(1ベース)の要素にアクセス
x[y..z]         // 配列xのy番目からz番目(0ベース)の要素にアクセス
x?.y            // xがnullでない時のみyにアクセス
x?[y]           // xがnullでない時のみyにアクセス
x ??= y         // xがnullであればy、さもなくばx
x()             // xメソッドを呼び出す
x ? y : z       // xが真であればy、さもなくばz
x() => ...      // ラムダ式
x::y            // using x = zの時、x::y でz.yにアクセス
```

制御構文

if文

```
// nが100より大きければBig!、さもなくばSmall!を表示する
if (n > 100)
{
    Console.WriteLine("Big!");
}
else
{
    Console.WriteLine("Small");
}
```

for文

```
// i=0, i=1, ..., i=9 の間ループする
for (int i = 0; i < 10; i++)
```

```
{
    Console.WriteLine(i);
}
```

foreach文

```
// 文字列配列をひとつずつループ処理する
foreach (string s in strings_list)
{
    Console.WriteLine(s);
}
```

while文

```
// nが10より小さい間ループする
while (n < 10)
{
    Console.WriteLine(n);
    n++;
}
```

do文

```
// nが10より小さい間ループする
do
{
    Console.WriteLine(n);
    n++;
} while (n < 10);
```

break文

```
ループ
{
    // nが10より大きければループを抜ける
    if (n > 10)
    {
        break;
    }
}
```

continue文

```
ループ
{
    // nが5より小さければ次のループを繰り返す
    if (n < 5)
    {
        continue;
    }
}
```

switch文

```
// nが1であればOne、2であればTwo、さもなくばManyを表示する
switch (n)
```

```
{
    case 1:
        Console.WriteLine("One");
        break;
    case 2:
        Console.WriteLine("Two");
        break;
    default:
        Console.WriteLine("Many");
        break;
}
```

goto文

```
    // ラベルの箇所にジャンプする
    err = Func();
    if (err != 0)
    {
        goto Error;
    }
    ...
Error:
    return err;
```

yield文

```
// foreach (int n in Range(0, 10)) { ... } で使用可能な IEnumerableインタフェースを提供
static IEnumerable<int> Range(int from, int to)
{
    for (int i = from; i < to; i++)
    {
        yield return i;
    }
    yield break;
}
```

checked, unchecked文

```
unchecked
{
    // オーバーフローして-2147483648になる(デフォルト動作)
    int n = int.MaxValue;
    Console.WriteLine(n + 1);
}
checked
{
    // OverflowException例外が発生するようになる
    int n = int.MaxValue;
    Console.WriteLine(n + 1);
}
```

lock文

```
static void MyThread(Counter counter)
{
    for (int i = 0; i < 100; i++)
    {
        // マルチスレッド環境でcounterオブジェクトをロックする
```

```
        lock (counter)
        {
            var count = counter.getCount();
            counter.setCount(count + 1);
        }
    }
}
```

例外処理(try...catch)

try { ... } の中で発生した例外は **catch** で受け取ることができます。**throw** は例外を生成して **catch** に投げつけます。**fainally** には例外の有無に関わらず実行する後処理機記述します。

```
try
{
    if (/* エラーが発生したら */)
    {
        // 例外を投げる
        throw new Exception("Message...");
    }
    // 例外が発生するとこれ以降は実行されない
}
catch (Exception e)
{
    // 例外を受け取る
    ...
}
finally
{
    // 例外の有無に関われず実行する
    ...
}
```

クラス(class)

class はクラスを定義します。**new** はクラスのインスタンスを生成します。クラス名と同じ名前のメソッドはコンストラクタとして生成時に呼び出されます。クラスメソッドでは自分自身のオブジェクトを **this** で参照します。~クラス名() のメソッドはファイナライザー(デストラクタ)として、インスタンスがガベージコレクションによって完全に破棄されるタイミングで呼び出されます。

```
class Animal
{
    public string type;

    // クラス名と同じメソッドはコンストラクタとして呼び出される
    public Animal(string type)
    {
        this.type = type;
    }

    // ~クラス名のメソッドはファイナライザーとして呼び出される
    ~Animal()
    {
        // 破棄時の処理...
    }
}
```

```
Animal a1 = new Animal("dog");
```

構造体(struct)

構造体とクラスは似ていますが、クラスの方が継承できるなど高機能、構造体の方が機能は少ないけど高速という特徴があります。クラスは通常ポインタを引き渡し、構造体は中身を引き渡します。通常のプログラミングではクラスを使用することが多く、16バイト程度以下の小さなデータ構造や、Win32 API に引き渡すデータを定義する際に構造体を用います。

```
struct Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

Point p1 = new Point(100, 200);
```

フィールドとプロパティ

クラスは **フィールド** や **プロパティ** を持つことができます。フィールドはメンバ変数とも呼ばれ、値を直接設定・参照します。プロパティは **get**, **set** の **アクセッサー** を用いて値を設定・参照します。**value** は **set** における値を示します。

```
class Person
{
    private string _name;               // フィールド
    public string Name                  // プロパティ
    {
        get { return _name; }           // アクセッサー(get)
        set { _name = value; }  // アクセッサー(set)
    }
}

class MyClass
{
    static void Main()
    {
        Person p1 = new Person();
        p1.Name = "Yamada";
        Console.WriteLine(p1.Name);
    }
}
```

メソッド

クラスや構造体はメソッドを持つことができます。メソッドは0個以上の引数を持つことができます。**return** にはメソッドの戻り値を指定します。

```
class MyClass
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}

class MainClass
{
    static void Main()
    {
        var p = new MyClass();  // クラスをインスタンス化して
        var ans = p.Add(5, 3);  // 呼び出す
        Console.WriteLine(ans);
    }
}
```

static 宣言したメソッドは、クラスをインスタンス化しなくても呼び出すことができます。

```
class MyClass
{
    public static int Add(int x, int y)
    {
        return x + y;
    }
}

class MainClass
{
    static void Main()
    {
        var ans = MyClass.Add(5, 3);
        Console.WriteLine(ans);
    }
}
```

可変引数(params)

params を用いることで可変引数のメソッドを定義できます。

```
static void Func(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.WriteLine(list[i]);
    }
}
static void Main()
{
    Func(1, 2, 3, 4, 5);
}
```

クラスの継承

クラスを継承するには下記の様に宣言します。

```
// Animalクラスを継承する Catクラスを定義する
class Cat : Animal
{
    // 親クラスのコンストラクタを呼び出す
    public Cat() : base("cat")
    {
    }
}

Cat c1 = new Cat();
```

親が持つメソッドを上書きする際は **new** をつけます。

```
class Cat : Animal
{
    // 親クラスが実装済みのCry()メソッドを上書きする
    new public void Cry()
    {
        Console.WriteLine("miaow miaow");
    }
}
```

親が持つメソッドを呼び出す場合は **base** を用います。

```
class Cat : Animal
{
    // 親クラスが実装済みのCry()メソッドを呼び出す
    new public void Cry()
    {
        base.Cry();
    }
}
```

親クラスのメソッドをオーバーライドするには、親クラスのメソッドのメソッドに **virtual**、子クラスのメソッドに **override** を使用します。

```
class Animal
{
    public virtual void Cry() { ... }
}
class Cat : Animal
{
    public override void Cry() { ... }
}
```

sealed をつけたクラスは継承することができません。**sealed** をつけたメソッドは親クラスのメソッドが **virtual** を指定していても、子クラスはオーバーライドすることができません。

```
sealed class Animal { ... }

class Cat : Pet {
    sealed public override void Cry() { ... }
}
```

アクセス修飾子

private, **protected**, **internal**, **public**, **internal** のアクセス修飾子を用いて、クラス、フィールド、プロパティ、メソッドなどを参照可能なスコープを制御することができます。アセンブリは .exe または .dll の単位。

```
private                 // 自クラスからのみ参照可能。
protected               // 自クラスおよび派生クラスから参照可能。
internal                // 自アセンブリから参照可能。
public                  // 自アセンブリ、および他アセンブリから参照可能。
protected internal      // 自クラスおよび派生クラス、および自アセンブリから参照可能。
private protected       // 自クラスおよび派生クラス、ただし自アセンブリに限る。
```

抽象クラス(abstract)

abstract は抽象クラスを定義します。抽象クラスはそれ自体ではインスタンス化することができず、継承するクラスが実装すべきメソッドなどを定義します。Animal を継承するクラスであれば、ちゃんと Cry() メソッドを実装しろという作法を定義するようなものですね。**override** は親クラスのメソッドをオーバーライドします。

```
// Animalを抽象クラスとして定義する
abstract class Animal
{
    // Aminalを実装するクラスはCry()というメソッドを実装すること
    public abstract void Cry();
}

// Aminalを実装するクラスはCry()というメソッドを実装すること
class Cat : Animal
{
    public override void Cry()
    {
        Console.WriteLine("miaow miaow");
    }
}
```

インタフェース(interface)

interface はインタフェースを定義します。インタフェースはそれ自体ではインスタンス化することができず、継承するクラスが実装すべきメソッドを定義します。抽象クラス と似ていますが、抽象クラスは抽象メソッドだけではなく実メソッドも実装できるのに対し、インタフェースは実メソッドは実装できませんが、カンマ(,)で連結することにより多重継承を実装することができます。

```
interface Criable<T>
{
    void Cry();
}

class Cat : Criable<Cat>
{
    public void Cry()
    {
        Console.WriteLine("miaow miaow");
    }
}

class MyClass
{
    static void Main()
    {
```

```
        Cat c1 = new Cat();
        c1.Cry();
    }
}
```

演算子のオーバーロード(operator)

operator を用いて + などの演算子をオーバーロードすることができます。

```
class String
{
    string _str;

    public String(string str)
    {
        this._str = str;
    }
    public override string ToString()
    {
        return this._str;
    }
    public static String operator+ (String x, String y)
    {
        return new String(x._str + y._str);
    }
}

class MyClass
{
    static void Main()
    {
        String s1 = new String("ABC");
        String s2 = new String("XYZ");
        String s3 = s1 + s2;
        Console.WriteLine(s3.ToString());
    }
}
```

インデクサー

下記の様に、インスタンスに対して [...] を付けた時の振る舞いを定義することができます。

```
class SampleCollection<T>
{
    private T[] _arr = new T[100];

    public T this[int i]
    {
        get { return _arr[i]; }
        set { _arr[i] = value; }
    }
}

class MyClass
{
    static void Main()
    {
        SampleCollection<string> arr = new SampleCollection<string>();
```

```
        arr[0] = "Hello world!";
        Console.WriteLine(arr[0]);
    }
}
```

型判定・型変換

.GetType() で型を取得し、**typeof()** と比較します。**in** はその型に変換可能かどうかを調べます。**as** 型 や (型) はその型に変換します。**as** 変換では変換できない場合にコンパイルの警告がでて null が返却されるのに対し、(...) 変換(キャスト)ではコンパイルエラーが発生します。

```
class Animal { public string Cry() { return "(unknown...)"; } }
class Cat : Animal { new public string Cry() { return "miaow miaow"; } }
class Book {}

class MainClass
{
    static void Main()
    {
        var c1 = new Cat();

        // .GetType()とtypeof()で型を比較する
        Console.WriteLine(c1.GetType() == typeof(Animal));      // False
        Console.WriteLine(c1.GetType() == typeof(Cat)); // True
        Console.WriteLine(c1.GetType() == typeof(Book));        // False

        // isでその型に変換できるか確認する
        Console.WriteLine(c1 is Animal);                        // True
        Console.WriteLine(c1 is Cat);                           // True
        Console.WriteLine(c1 is Book);                  // False

        // asでその型に変換する
        Console.WriteLine(c1.Cry());                            // miaow miaow
        Console.WriteLine((c1 as Animal).Cry());                // (unknown...)

        // (...)でその型に変換(キャスト)する
        Console.WriteLine(c1.Cry());                            // miaow miaow
        Console.WriteLine(((Animal)c1).Cry());          // (unknown...)
    }
}
```

明示的・暗黙的型変換(explicit・implicit)

explicit は明示的、**implicit** は暗黙的な型変換を指定します。下記の例ではミリメートルからメートルの変換は桁落ちするので明示的、メートルからミリメートルの変換は桁落ちしないので暗黙的な型変換の手法を用いています。

```
class Meter
{
    public int value;
    public Meter(int value)
    {
        this.value = value;
    }
    public static explicit operator Meter(MilliMeter mm)
    {
        return new Meter(mm.value / 1000);
```

```
    }
}

class MilliMeter
{
    public int value;
    public MilliMeter(int value)
    {
        this.value = value;
    }
    public static implicit operator MilliMeter(Meter m)
    {
        return new MilliMeter(m.value * 1000);
    }
}

class MyClass
{
    static void Main()
    {
        MilliMeter mm1 = new MilliMeter(12345);
        Console.WriteLine(mm1.value + "mm");
        Meter m1 = (Meter)mm1;          // 明示的型変換
        Console.WriteLine(m1.value + "m");
        MilliMeter mm2 = m1;                    // 暗黙的型変換
        Console.WriteLine(mm2.value + "mm");
    }
}
```

デリゲート(delegate)

delegate はメソッドの形式(戻り値の型や引数の個数・型)を型として定義します。

```
// 「int引数を受け取り値は返却しないメソッド」という型を定義する
delegate void SampleDelegate(int n);

class MainClass
{
    static void Main()
    {
        // SampleDelegate型変数としてfuncを定義する
        SampleDelegate func = MyFunc;
        // funcをメソッドとして呼び出す
        func(123);
    }

    static void MyFunc(int n)
    {
        Console.WriteLine(n);
    }
}
```

イベント(event)

event を用いて、JavaScript の **addEventListener()** の様なイベントハンドラを定義することができます。
delegate によってイベントハンドラの戻り値の型や引数も定義できる点が JavaScript よりも柔軟になっています。

```
using System;

// イベントハンドラの型や引数を定義する
public delegate void OnClickHandler(object sender, EventArgs e);

class MyButton
{
    // イベントを定義する
    public event OnClickHandler onclick;

    public void Click()
    {
        // イベントを発行する
        this.onclick(this, EventArgs.Empty);
    }
}

class MyClass
{
    // イベントハンドラ
    static public void OnClick(object o, EventArgs e)
    {
        Console.WriteLine("Clicked!!");
    }

    static void Main(string[] args)
    {
        MyButton btn = new MyButton();
        // イベントにイベントハンドラを追加する(+=演算子により複数設定可)
        btn.onclick += new OnClickHandler(MyClass.OnClick);
        btn.Click();
    }
}
```

add と **remove** キーワードは、複数のインタフェースから同名のイベントを継承する際に、どちらの実装を優先するかを決める際に使用しますが、通常は使用する必要はありません。詳細は Microsoft のリファレンスを参照してください。(→ add, remove)

スレッド

スレッドは下記の様に使用します。

```
using System;
using System.Threading;

class MyClass
{
    static void Main()
    {
        var threadA = new Thread(() => {
            for (int i = 0; i < 100; i++)
            {
                Console.Write("a");
                Thread.Sleep(100);
            }
        });
        var threadB = new Thread(() => {
            for (int i = 0; i < 100; i++)
            {
```

```
                Console.Write("b");
                Thread.Sleep(100);
            }
        });
        threadA.Start();        // スレッドAを開始する
        threadB.Start();        // スレッドBを開始する
        threadA.Join(); // スレッドAの終了を待つ
        threadB.Join(); // スレッドBの終了を待つ
    }
}
```

タスク

タスクは下記の様に使用します。0 から 100 までの合計を求めるタスクを二つ起動しています。**WhenAll()** は引数で指定したすべてのタスクが完了するのを待ちます。**.Result** はタスクの戻り値を参照します。

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

class MyClass
{
    // 0から100までの合計を求めるタスク
    static int CountUpTask()
    {
        int total = 0;
        for (int i = 0; i <= 100; i++) { total += i; Thread.Sleep(10); }
        return total;
    }

    static void Main()
    {
        // タスクA と タスクB を実行する
        var taskA = Task.Run(() => {
            return CountUpTask();
        });
        var taskB = Task.Run(() => {
            return CountUpTask();
        });
        // タスクA と タスクB 両方の完了を待つ
        Task.WhenAll(taskA, taskB);
        // タスクA と タスクB の戻り値を表示する
        Console.WriteLine(taskA.Result);
        Console.WriteLine(taskB.Result);
    }
}
```

async と await

async と **await** は、同期関数を非同期関数にしたり、それをあたかも同期関数の様に呼び出す際に使用します。REST API を呼び出したり、ファイルを読み書きしたりなど時間のかかる処理を非同期化し、それを同期的に呼び出すことにより、プログラムは同期的で分かりやすいけど、時間がかかっている間、フォームの更新を行うなどの別処理を行うことができるようになります。(フォームを使用しないサンプルを示したいけど思いつかない...)

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

```
class MyClass
{
    // 同期関数(0から100までを合計した結果を返す同期処理)
    static int CountUpTask(string name)
    {
        int total = 0;
        for (int i = 0; i <= 100; i++)
        {
            total += i;
            Thread.Sleep(10);
            Console.Write(name);
        }
        return total;
    }

    // 非同期関数(上記の関数を非同期関数化したもの)
    static async Task<int> CountUpAsync(string name)
    {
        int total = 0;
        for (int i = 0; i <= 100; i++)
        {
            total += i;
            await Task.Delay(10);
            Console.Write(name);
        }
        return total;
    }

    // awaitを使用するメソッドには必ずasyncをつける
    static async Task Main()
    {
        // 定常処理(ひたすら"."を書き出す)
        bool stop = false;
        var taskX = Task.Run(() => { while (!stop) { Console.Write("."); Thread.Sleep(10);

        // 同期関数をタスクとして非同期的に呼び出す
        var taskA = Task.Run(() => CountUpTask("A"));
        Task.WhenAll(taskA);            // タスクが完了するのを待つ
        var countA = taskA.Result;      // タスクの結果を取り出す

        // 非同期関数を同期的に呼び出す(1)
        // 非同期関数とawaitを使用すれば、上記の処理を同期関数の様に記述することができる
        var countB = await CountUpAsync("B");

        // でも、同期関数を同期的に呼び出すのとどう違うのかを示すサンプルを思いつかない...
        var countC = CountUpTask("C");

        stop = true;
        Console.WriteLine("\n{0} {1} {2}", countA, countB, countC);
    }
}
```

ポインタ

C# でポインタを使用するには、コンパイルオプションに **/unsafe** をつけ、**unsafe** ブロックの中で使用する必要があります。さらに、ガベージコレクションによってポインタが変動してしまうのを防ぐために **fixed** を使用します。

```
class MyClass
{
    // unsafeでメモリアンセーフな状態にする
    unsafe static void Main()
    {
        byte[] arr = {100, 200, 300};
        // fixedでポインタが変動してしまうのを防ぐ
        fixed (byte *p = arr)
        {
            Console.WriteLine(*(p + 0));        // => 100
            Console.WriteLine(*(p + 1));        // => 200
            Console.WriteLine(*(p + 2));        // => 300
        }
    }
}
```

パッケージ

パッケージファイル mypkg.cs と、パッケージを利用するメインプログラム main.cs を作ります。**namespace** はパッケージの名前空間を指定します。

mypkg.cs

```
using System;

namespace MyPackages
{
    public class MyClass
    {
        public void MyMethod()
        {
            Console.WriteLine("Hello!!");
        }
    }
}
```

main.cs

```
using MyPackages;

class Hello
{
    static void Main()
    {
        MyClass p = new MyClass();
        p.MyMethod();
    }
}
```

パッケージファイル(*.dll)にコンパイルし、それを利用するメインプログラムをコンパイルします。

```
csc /t:library mypkg.cs # mypkg.dllが生成される
csc /r:mypkg.dll main.cs
```

using文

using を用いることで、**System.Console.WriteLine()** を **Console.WriteLine()** と記述できるようになります。

```
System.Console.WriteLine(...);
    ↓
using System;
Console.WriteLine(...);
```

使用する名前空間に別名をつけることもできます。

```
using sys = System;
sys::Console.WriteLine(...);
```

部分クラス(partial)

partial は大規模開発において一つのクラスを複数のファイルに分離して開発することを可能にします。下記の例では、MyPkg パッケージの中の MyClass という一つのクラスを mypkgA.cs と mypkgB.cs という二つのファイルに分割して開発しています。

mypkgA.cs

```
namespace MyPkg
{
    public partial class MyClass
    {
        public void MethodA() { ... }
    }
}
```

mypkgB.cs

```
namespace MyPkg
{
    public partial class MyClass
    {
        public void MethodB() { ... }
    }
}
```

コンパイル

```
csc /t:library -out:mypkg.dll mypkgA.cs mypkgB.cs
```

globalエイリアス

global エイリアスは **System** などを持つグローバルなエイリアスとして使用できます。下記の場合、global:: が無いと Console は System.Console ではなく MyProduct.System.Console として解釈されます。

```
amespace MyProduct.System
{
    class MyClass
    {
        static void print(string msg)
        {
            global::System.Console.Write(msg);
        }
        static void Main()
        {
            print("Hello\n");
        }
```

```
    }
}
```

外部関数(extern)

extern は Win32 API など外部の関数を呼び出す際に使用します。

```
using System.Runtime.InteropServices;

class MyClass
{
    User32.dllで定義された外部の関数を呼び出す
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        return MessageBox((IntPtr)0, "Hello!", "My Message Box", 0);
    }
}
```

alias は、外部DLL に別名をつけます。コンパイルオプションに /r:GridV1=grid.dll, /r:GridV2=grid20.dll と指定し、これをプログラム中で、GridV1::Grid() と GridV2::Grid() の様に呼び出すことが可能となります。

```
extern alias GridV1;
extern alias GridV2;
```

変数名参照(nameof)

nameof は変数名を文字列として返します。

```
string name = "Tanaka";
int age = 123;
Console.WriteLine(nameof(name));        // => "name"
Console.WriteLine(nameof(age)); // => "age"
```

クエリ(LINQ)

プログラム中に **LINQ** と呼ばれる SQLライクな構文を使用することができます。**from**, **where**, **select**, **into**, **orderby**, **join**, **let**, **in**, **on**, **equals**, **group**, **by**, **ascending**, **desending** などが使用できます。

```
List<int> Scores = new List<int>() { 97, 92, 81, 60 };
IEnumerable<int> queryHighScores =
    from score in Scores
    where score > 80
    select score;
foreach (int n in queryHighScores)
{
    Console.Write(n + " ");
}
```

in・ref・out修飾子

in修飾子は、引数が ReadOnly であり変更が禁止されることを明示します。

```
static void Func(in num)
{
    // num = 999;                // 変更しようとするとコンパイルエラーとなる
}
static void Main()
{
    int num = 123;
    Func(num);
}
```

ref修飾子は、引数の**参照渡し**に使用されます。通常の引数は **値渡し** で値のコピーが渡されますが、参照渡しではそのポインターが渡されます。大きなデータの場合コピーが発生しない分高速になったり、メソッドの中で元のデータを変更することが可能となります。

```
static void Func(ref int num)
{
    num = 999;
}
static void Main()
{
    int num = 123;
    Func(ref num);
    Console.WriteLine(num);     // => 999
}
```

out修飾子は、**ref** と似ていますが、メソッドの中で必ず値が変更される必要があります。

```
static void Func(out int num)
{
    num = 999;                  // 変更されない場合はエラーとなる
}
static void Main()
{
    int num = 123;
    Func(out num);
    Console.WriteLine(num);     // => 999
}
```

ref修飾子はまた、**参照戻り値** にも使用されます。戻り値を値渡しではなく参照渡しにすることで、巨大データのコピーを防ぎます。

```
static int num = 123;
static ref int Func()
{
    return ref num;
}
static void Main()
{
    ref int num = ref Func();
    Console.WriteLine(num);
}
```

readonly修飾子

readonly修飾子は、対象が読み取り専用で変更が禁止されることを宣言します。フィールド、構造体、構造体メンバ、ref参照戻り値に対して使用できます。

```
class Cat
{
    readonly string name;               // フィールド
    public Cat(string name)
    {   // コンストラクタでのみ値を設定可能
        this.name = name;
    }
}

readonly struct Point                   // 構造体
{
    readonly int x, y;                  // 構造体メンバ
    public Point(int x, int y)
    {   // コンストラクタでのみ値を設定可能
        (this.x, this.y) = (x, y);
    }
}

static int num = 123;
static ref readonly int Func2() // ref参照戻り値
{
    return ref num;
}
```

sizeof演算子

sizeof演算子で、型が占有するバイト数を得ることができます。

```
// byteやintに対しては常に使用可。stringやobjectは不可
var n1 = sizeof(int);

// ポインタに対してはunsafeな場合に使用可
unsafe { var n2 = sizeof(int*); }

// クラス自体のサイズを得るにはunmanaged制約を用いる
n3 = MySizeOf<Point>();
static unsafe int MySizeOf<T>() where T : unmanaged { return sizeof(T); }
```

stackalloc式

stackalloc式を用いることにより、ガベージコレクションの対象となるヒープ領域ではなく、高速なスタック領域にメモリを確保することができます。スタック領域ですのでメソッドが終了した時点で自動的に開放されます。

```
// ヒープ領域に確保する例
static void HeapAlloc()
{
    int length = 1024;
    int[] numbers = new int[length];
    for (var i = 0; i < length; i++) { numbers[i] = i; }
}
// スタック領域に確保する例。面倒だが高速。あまり大きなデータは不可
static void StackAlloc()
{
    int length = 1024;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++) { numbers[i] = i; }
}
```

volatileキーワード

volatileキーワードは、そのフィールドが複数のスレッドからアクセスされる可能性があることを示します。int やポインタなど、1回のCPU命令で読み書き可能なフィールドに対してのみ使用できます。double や long などには使用できません。ローカル変数にも指定できません。**volatile** を付与することで、コンパイラやランタイムやハードウェアによっては、最適化のアルゴリズムを変動させるかもしれませんが、lock の様にスレッドセーフな変数にはる訳ではありません。

```
public volatile int counter = 0;
```

dynamic型

dynamic 型は静的な型でありながら、静的型チェックが行われない変数を定義します。型チェックが行われないため、どんな型の値でも代入して使用することが可能です。

```
dynamic obj;
obj = new ClassA();
obj = new ClassB();
```

whenキーワード

C# 6.0 から、**catch** ステートメントに **when** で条件を記述できるようになりました。

```
var err = 0;
try
{
    err = 300;
    throw new Exception();
}
catch  (Exception e) when (err == 300)
{
    Console.WriteLine("Error 300");
    Console.WriteLine(e);
}
catch  (Exception e) when (err == 400)
{
    Console.WriteLine("Error 400");
    Console.WriteLine(e);
}
```

C# 7.0 からは **case** ステートメントにも **when** を記述できるようになりました。

```
string lang = "uk";
int floor = 1;
switch (floor)
{
    case 1 when lang == "us":
        Console.WriteLine("First floor");
        break;
    case 1 when lang == "uk":
        Console.WriteLine("Grand floor");
        break;
    case 2 when lang == "us":
        Console.WriteLine("Second floor");
        break;
```

```
    case 2 when lang == "uk":
        Console.WriteLine("First floor");
        break;
}
```

ジェネリック型制約(where, unmanaged)

下記の例で Max1() は int 型引数を比較して大きい方を返却しますが、double 型引数では使用できません。
Max2() は任意の型を受け取ることができるジェネリックメソッドとして実装しています。**where** は、その条件として IComparable<T> インタフェースを実装しているものであることを示しています。

```
static int Max1(int x, int y)
{
    return x > y ? x : y;
}

static T Max2<T>(T x, T y) where T : IComparable
{
    return x.CompareTo(y) > 0 ? x : y;
}
```

ジェネリック型制約に **unmanaged** を使用することで、ジェネリックメソッドにアンマネージドなポインタを使用することも可能になりました。

```
struct Point
{
    public byte x, y;
    public Point(byte x, byte y) { (this.x, this.y) = (x, y); }
}

class MyClass
{
    unsafe static void MemSet0<T>(ref T x, byte c) where T : unmanaged
    {
        fixed (T* p = &x)
        {
            var b = (byte *)p;
            var size = sizeof(T);
            for (int i = 0; i < size; i++) { b[i] = c; }
        }
    }

    static void Main()
    {
        var p = new Point(10, 20);
        MemSet0<Point>(ref p, 30);
        Console.WriteLine("{0} {1}", p.x, p.y);
    }
}
```

リンク

- C#関連ドキュメント (Microsoft)
- C#6.0ドラフト仕様 (Microsoft)
