言語まとめ C#

[言語 まとめ][C#][C# サンプルコード][Effective C# 4.0][Universal Windows Platform][Visual Studio][プログラミング C# 第 7 版]

1. プログラミング C# 第 7 版 (1)
2. プログラミング C# 第 7 版 (2)

準備

参照サイト

■ C# 言語仕様

・C Sharp Language Specification version 3.0
・言語仕様 3.0
・MSDN C# プログラミングガイド

■ MSDN Library

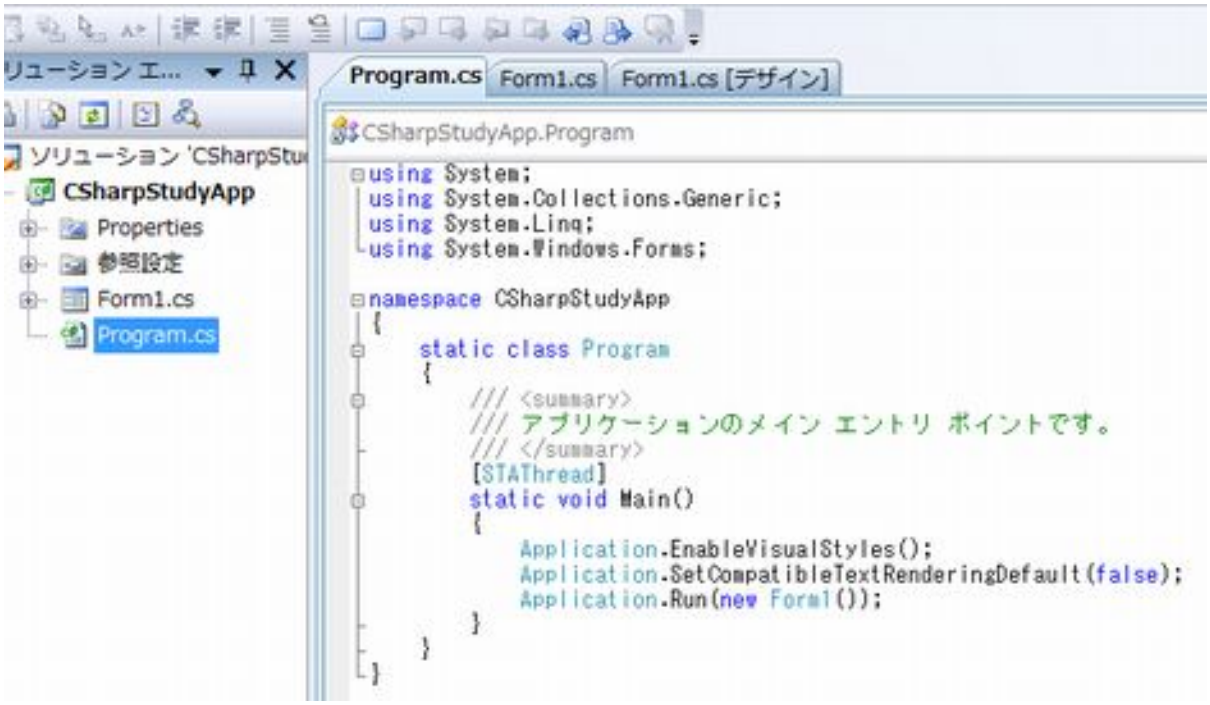
・Visual Studio 2008 Visulal C#

環境

エントリーポイント

Main メソッド

・オブジェクトを作成し、他のメソッドを呼び出す、プログラムのエントリ ポイント。
・C# プログラムでは、エントリ ポイントは 1 つだけ。
・クラスまたは構造体の内部で宣言。静的である必要がある。パブリックにはしない。
・戻り値の型は、void か int
・パラメータなしでも宣言可能
・パラメータは、ゼロから始まるインデックス付きのコマンド ライン引数として読み取れる



データ型

・C# には値型と参照型がある。
・値型の変数は直接値を保持し、参照型はデータ自体への参照を保持する。
・値型は、組み込み数値型、列挙型、構造体型、null 許容型に分かれる
・参照型はクラス型、インターフェイス型、配列型、デリゲート型に分かれる

値型

■数値型

内容	型
符号付整数	sbyte short int long
符号なし整数	byte ushort uint ulong
ユニコード文字	char
IEEE 浮動小数点数	float double
高有効桁数 10 進数	decimal 実数値リテラルを decimal として扱うには、サフィックス m または M を使用
真偽値	bool

列挙型、構造体型、null 許容型

内容	型
列挙型	enum E { ... }
構造体型	struct S { ... }

null 許容型	null 許容型は、基礎となる値型の正しい範囲の値だけでなく、null 値も表すことができる。構文 T? は、Nullable<(Of <(T>)>) の省略表現

構造体の例

```
public struct Person
{
    public string name;
    public int age;
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person myself;
        myself.name = "yagi";
        myself.age = 38;

        Person someone = new Person("hoge", 20);

        System.Console.WriteLine(myself.name + "," + myself.age);
        System.Console.WriteLine(someone.name + "," + someone.age);
    }
}
```

列挙型の例

```
enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat }
static void Main(string[] args)
{
    System.Console.WriteLine(Days.Sun);
    int sun = (int)Days.Sun;
    System.Console.WriteLine(sun);
}
```

・結果

```
Sun
0
```

null 許容型の例

・int? は、Nullable<int> の省略表現

```
static void Main(string[] args)
{
    PrintValue(null);
    PrintValue(12345);
}
static void PrintValue(int? num)
{
    if (num.HasValue)
    {
        System.Console.WriteLine("num's value is : " + num.Value);
    }
    else
    {
        System.Console.WriteLine("num is null");
```

```
    }
}
```

構造体の場合も同様に利用

・http://stackoverflow.com/questions/109859/what-does-datetime-mean-in-c

```
DateTime? a = null;
if (!a.HasValue)
{
    a = DateTime.Now;
    if (a.HasValue)
    {
        Console.WriteLine(a.Value);
    }
}
```

・結果

```
num is null
num's value is : 12345
```

参照型

■クラス型

内容	型
すべての型の根本的な基底クラス	object
ユニコード文字列	string
ユーザー定義クラス	class C { ... }

■インターフェース型、配列型、デリゲート型

内容	型
インターフェース型	interface I { ... }
配列型	一次元 : int[]、多次元 : int[，]、ジャグ (配列の配列): int[][]
デリゲート型	メソッドを参照する型 例 delegate int D(...)

■多次元配列

例

```
int[,] ma = { { 1, 2, 3 }, { 4, 5, 6 } };
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
    {
        System.Console.WriteLine(ma[i, j]);
    }
}
```

■ジャグ配列

例

```
int[][] ja = new int[2][];
ja[0] = new int[] { 1, 2, 3};
ja[1] = new int[] { 4, 5 };
for (int i = 0; i < ja.Length; i++)
{
    for (int j = 0; j < ja[i].Length; j++)
    {
        System.Console.WriteLine(ja[i][j]);
    }
}
```

■デリゲート型

例

```
public static void PrintMessage(string msg)
{
    System.Console.WriteLine(msg);
}
public delegate void MessageHandler(string msg);

public static void SomeProcess(MessageHandler handler) {
    handler(" デリゲートする処理によりハンドらの挙動を変更可能 ");
}
static void Main(string[] args)
{
    MessageHandler handler = PrintMessage;
    SomeProcess(handler);
}
```

暗黙の型

■ var

・ローカル変数には、明示的な型ではなく、推論される " 型 " var を設定できる
・初期化ステートメントの右側の式から変数の型を推測することをコンパイラに指示

バリアント型 を意味するものではなく、変数の弱い型指定や遅延バインディングを示すものでもない。最適な型をコンパイラが決定し割り当てることを意味するだけ。

例

```
var i = 10;                 // int 型
var s = "hello.";           // string 型
var ia = new[] { 1, 2, 3 }; // int 配列型

var list = new List<int>();
list.Add(2);
list.Add(4);
foreach (var e in list)
{
    Console.WriteLine(e);
}
```

匿名型

・object から直接派生する参照型
・1 つ以上の読み取り専用のパブリック プロパティを持つクラス型。それ以外のクラス メンバ (メソッド、イベントなど) は使用できない。
・型名はコンパイラによって生成され、ソース コード レベルでは利用できない

・作成するには、new 演算子とオブジェクト初期化子を使用
・共通言語ランタイムから見た匿名型は、object 以外の型にキャストできないことを除き、他の参照型と同じ
・プロパティの型はコンパイラによって推論される。

例

```
var me = new { Name = "Yagi", Age = 38 };
Console.WriteLine("{0} {1}", me.Name, me.Age);
```

・通常、クエリ式の select 句で使用される。

例 2

```
var m = new Dictionary<string, string>();
m.Add("abc", "10");
m.Add("def", "11");
m.Add("hij", "20");
m.Add("klm", "30");

var query =
    from e in m
    where e.Value.EndsWith("0")
    select new { e.Key, e.Value };

foreach (var e in query)
{
    Console.WriteLine("{0},{1}", e.Key, e.Value);
}
```

・結果

```
abc,10
hij,20
klm,30
```

型の判定

■オブジェクトと、指定した型との間に互換性があるかどうかをチェック

・java での instanceof
・is

```
if (obj is MyObject)
{
}
```

■式の実行時の型を取得する

・GetType()

```
int i = 0;
System.Type type = i.GetType();
```

■型の System.Type オブジェクトを取得

・typeof()

```
System.Type type = typeof(int);
```

■指定した複数の Object インスタンスが同一かどうかを判断

```
boolean isSameRef = Object.ReferenceEquals(o1, o2);
```

データ変換

■ string を int に変換

```
int i = Convert.ToInt32("29");
int i = Int32.Parse("-105");
```

文字列

■ @ と二重引用符を使用

・リテラル文字列は string 型であり、二重引用符で囲む形式と、@ と二重引用符で囲む形式の 2 通りある。

Raw 文字列

・@ と二重引用符を使用を使用すると、エスケープシーケンスが処理されないため、正規表現やファイル名が記述しやすい。
・二重引用符を書く場合、2 つ続ける

```
Console.WriteLine(@"c:\work\test.txt");
```

ヒアドキュメント

・ヒアドキュメント的な使い方もできる。

```
        static void Main(string[] args)
        {
            Console.WriteLine(
@" このように、
ヒアドキュメント的な
使い方が、
できる。
");
        }
```

■文字列挿入 ($)

・http://ufcpp.net/study/csharp/st_string.html#FormatableString
・C# 6.0 で追加

```
var formatted = $"({x}, {y})";
```

このような書き方を文字列挿入 (string interpolation) といいます。 文字列挿入の結果は、単純に string.Format メソッドの呼び出しに置き替えられます。 例えば、最初の例は以下のコードと同じ

意味なります。

```
var formatted = string.Format("({0}, {1})", x, y);
```

■書式

・C# 書式

演算子

・https://msdn.microsoft.com/ja-jp/library/6a71f45d.aspx

??(null 合体演算子)

・null 許容値型および参照型の既定値を定義するために使用。
・左側のオペランドが null 値でない場合にはこのオペランドを返し、null 値である場合には右側のオペランドを返す

例

```
static int? hoge(int? x)
{
    return x;
}
static void Main(string[] args)
{
    int? x = hoge(10) ?? -1;
    int? y = hoge(null) ?? -1;
    Console.WriteLine("{0}, {1}", x, y);
}
```

・結果

```
10, -1
```

?(null 条件演算子)

・https://msdn.microsoft.com/en-us/magazine/dn802602.aspx
・C#6.0 から

これまで

```
if (value != null)
{
    result = value.Substring(0, Math.Min(value.Length, length));
}
return result;
```

null 条件演算子を利用

```
return value?.Substring(0, Math.Min(value.Length, length));
```

??(null 合体演算子) と合わせて使用

```
return value?.Substring(0, Math.Min(value.Length, length)) ?? 0;
```

nameof

・C#6.0 から
・http://ufcpp.net/study/csharp/ap_ver6.html#nameof-operator
・変数、型、またはメンバーの単純な (修飾されていない) 文字列名を取得するのに使用

ステートメント

選択

■ if、else

・if else

```
if (condition)
{
    // 真のときの処理
}
else
{
    // 偽のときの処理
}
```

・if else if

```
Days today = Days.Mon;

if (today == Days.Sun)
{
    System.Console.WriteLine("休み");
}
else if (today == Days.Sat)
{
    System.Console.WriteLine("半休");
}
else
{
    System.Console.WriteLine("仕事");
}
```

■ switch、case

```
Days today = Days.Mon;

switch (today) {
    case Days.Sun:
        System.Console.WriteLine("休み");
        break;
    case Days.Sat:
        System.Console.WriteLine("半休");
        break;
    default:
        System.Console.WriteLine("仕事");
        break;
}
```

繰り返し

■ do

```
int i = 0;
do
{
    System.Console.Write(i++ + "\t");
} while (i < 5);
```

■ for

```
for (int i = 0; i < 5; i++)
{
    System.Console.Write(i + "\t");
}
```

■ foreach

・配列またはコレクションのの要素に対してステートメントを繰り返す。

```
int[] ary = { 1, 2, 3, 4 };
foreach (int i in ary)
{
    System.Console.Write(i + "\t");
}
```

■ while

```
int i = 0;
while (i < 5)
{
    System.Console.Write(i++ + "\t");
}
```

ジャンプ

■ break、continue

```
int i = 0;
while (true)
{
    i++;
    if (i % 2 == 0)
    {
        continue;
    }
    if (i > 10)
    {
        break;
    }
    System.Console.Write(i + "\t");
}
```

■ goto、default

・通常、goto は switch ステートメントの特定の switch-case ラベルまたは default ラベルに制御を移動するのに使用
・階層の深い入れ子のループから抜ける際にも便利

```
int num = 4;
int ttl = 0;
switch (num)
{
    case 1:
        ttl += 1;
        break;
```

```
    case 2:
        ttl += 2;
        goto case 1;
    case 3:
        ttl += 3;
        goto case 2;
    case 4:
        ttl += 4;
        goto case 3;
    default:
        break;
}
System.Console.Write(ttl);
```

■ return

・return ステートメントは、メソッドの実行を終了し、呼び出し側のメソッドに制御を戻す

■ yield

・列挙子オブジェクトに値を挿入する場合、または反復処理の終了を通知する場合に、iterator ブロック内で使用
・yield ステートメントは、メソッド、演算子、またはアクセサの本体として使用される iterator ブロック内でのみ使用できる

```
public static IEnumerable Range(int size)
{
    int result = 0;
    while (result < size)
    {
        yield return result;
        result++;
    }
}
public static void Main(string[] args)
{
    foreach (int i in Range(5))
    {
        System.Console.Write(i + "\t");
    }
}
```

・結果

```
0       1       2       3       4
```

例外

■ try、catch、finally、throw

・catch 句は、引数なしで使用できる。引数を指定せずに使用すると、すべての種類の例外がキャッチされ、汎用的な catch 句と見なされる。
・throw ステートメントは、catch ステートメントでキャッチされた例外を再びスローするために catch ブロックで使用できる。
・パラメータのない catch 句で処理された例外を再スローする場合、引数のない throw ステートメントを使用

```
try
{
    int[] num = { 1, 2, 4 };
    System.Console.WriteLine(num[10]);
}
catch (IndexOutOfRangeException ioe)
{
    System.Console.WriteLine("{0} : Exception Occured.", ioe);
```

```
}
catch (NullReferenceException nre)
{
    throw (nre);
}
catch
{
    System.Console.WriteLine("Exception Occured.");
    throw;
}
finally
{
    System.Console.WriteLine("Finally Block.");
}
```

■例外を明示的にスローする

・例外を明示的にスローするには、throw ステートメントを使用
・キャッチした例外を再びスローするときにも、throw ステートメントを使用

```
throw new FileNotFoundException(@"[data.txt not in c:\temp directory]",e);
```

Checked と Unchecked

・ステートメントは、checked または unchecked のいずれかのコンテキストで実行されます。checked コンテキストでは、算術オーバーフローの例外が発生します。unchecked コンテキストでは、算術オーバーフローは無視され、結果は切り捨てられます

```
try
{
    int max_val = Int32.MaxValue;
    System.Console.WriteLine(max_val);
    max_val = max_val + 1;
    System.Console.WriteLine(max_val);

    // オーバーフローチェック
    int checked_max_val = Int32.MaxValue;
    checked_max_val = checked(checked_max_val + 1);
    System.Console.WriteLine(checked_max_val);
}
catch (OverflowException e)
{
    System.Console.WriteLine(e);
}
```

・結果

```
2147483647
-2147483648
System.OverflowException: 算術演算の結果オーバーフローが発生しました。
               場　所　　CSharpConsoleStudyApp.Program.Main(String[]　　args)　　場　所
C:\...\CSharpConsoleStudyApp\Program.cs: 行 21
```

fixed

・マネージ変数へのポインタを設定し、実行時にマネージ変数を " 固定 " します。fixed がない場合、ガベージ コレクションが予期できないかたちで移動可能なマネージ変数を再配置するため、マネージ変数へのポインタはほとんど役に立ちません。

lock

・lock キーワードは、指定のオブジェクトに対する相互排他ロックを取得し、ステートメン

トを実行し、ロックを解放するステートメント ブロックをクリティカル セクションとしてマークします。

式

リテラルと簡易名、呼び出し式

リテラルと簡易名

- 5 と "Hello World" は共にリテラル
- i と s は、共にローカル変数を識別する簡易名
- これらの変数を式で使用すると、変数の値が取得され、式で使用される

```
int i = 5;
string s = "Hello World";
```

呼び出し式

- DoWork への呼び出しも、呼び出し式と呼ばれる式の一種

```
int var = 5;
DoWork(var);
```

クエリ式 (LINQ クエリ式)

- 統合言語クエリ (LINQ: Language-Integrated Query) は、クエリ機能を言語に直接統合する技術。
- LINQ を使用すると、クエリは、クラス、メソッド、イベントなどと同じように、高度な機能を備えた言語構成要素になる。
- 一般的な式の規則と同じ規則がクエリ式に適用される。

例

```
string[] member = { "asano", "inoue", "uesugi", "okada", "onoue", "katou"};
IEnumerable<string> choiced_member =
    from member_name in member
    where member_name.StartsWith("o")
    select member_name;

foreach (string choiced_member_name in choiced_member)
{
    System.Console.WriteLine(choiced_member_name);
}
```

- 結果

```
okada
onoue
```

ラムダ式

- ラムダ式は式とステートメントを含めることができる匿名関数であり、デリゲート型または式ツリー型を作成するために使用できる
- すべてのラムダ式でラムダ演算子 => が使用され、ラムダ演算子の左辺で入力パラメータ

を指定し (ある場合)、右辺には式ブロックまたはステートメント ブロックが入ります

```
引数リスト => 式
```

```
delegate int del(int num);
public static void Main(string[] args)
{
    del pow = x => x * x;
    System.Console.WriteLine(pow(5));
}
```

ラムダ式は、次のように省略できる

```
x => x + 1                  // 暗黙に型付けされた、式本体
x => { return x + 1; }      // 暗黙に型付けされた、文本体
(int x) => x + 1            // 明示的に型付けされた、式本体
(int x) => { return x + 1; } // 明示的に型付けされた、文本体
(x, y) => x * y             // 複数のパラメータ
() => Console.WriteLine()   // パラメータなし
```

■パラメータがアンダースコア

・パラメータを利用しない場合、パラメータ名をアンダースコアとすることで表現する
・https://dismantledtech.wordpress.com/2014/06/07/using-underscore-to-denote-unused-parameters-in-c-lambdas/

```
this.Updated += (sender, args) =>
{
    // 通常
}
```

```
this.Updated += (_, args) =>
{
    // args しか使わない
}
```

```
this.Updated += (_, __) =>
{
    // sender も args も使わない
}
```

空のラムダ式の場合、有用

```
DoExport((_, __) => { });
```

式ツリー

・式をデータ構造体として表すことができます。クエリ式を、SQL データベースなどの他のコンテキストで意味を持つコードに変換するために、LINQ プロバイダにより広く使用されています。

```
TODO
```

名前空間

名前空間を使用して多くのクラスを編成する

・System が名前空間で、Console がこの名前空間のクラス

```
System.Console.WriteLine("Hello C# World!");
```

using

・using キーワードを使用すると、完全な名前を記述する必要がなくなる

```
using System;
    :
Console.WriteLine("Hello C# World!");
```

クラス名とメソッド名のスコープの管理を容易にする

・独自の名前空間を宣言すると、大型のプログラミング プロジェクトでクラス名とメソッド名のスコープの管理が容易になる
・namespace キーワードを使用して名前空間を宣言する

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

名前空間エイリアス修飾子 (::)

```
using System;

namespace CSharpConsoleStudyApp
{
    namespace System
    {
        public class Cosole
        {
            public static void Writeln(string msg)
            {
                // 名前空間エイリアス修飾子 (::) を使用して識別子を検索する
                global::System.Console.Write("[debug] : " + msg);
            }
        }
    }
    class Program
    {
        static void Main()
        {
            // System 名前空間が CSharpConsoleStudyApp.System 名前空間によって隠されている
            System.Cosole.Writeln("message");
        }
    }
}
```

using

・アンマネージ リソースや、それをカプセル化するクラス ライブラリ型はすべて、IDisposable インターフェイスを実装する必要があります。(File や Font など)
・一般に、IDisposable オブジェクトを使用するときは、それを using ステートメントで宣言して、インスタンス化する
・using ステートメントは、オブジェクトで正しく Dispose メソッドを呼び出す

・using ステートメントを使うと、オブジェクトでのメソッドの呼び出し中に例外が発生した場合でも Dispose が必ず呼び出される
・オブジェクトを try ブロックに配置し、finally ブロックで Dispose を呼び出すのと同じ結果

```
using (Font font3 = new Font("Arial", 10.0f),
            font4 = new Font("Arial", 10.0f))
{
    // Use font3 and font4.
}
```

クラス

宣言

・class キーワードを使用して定義

```
public class Foo
{
    public void hoge()
    {
        System.Console.WriteLine("Foo.hoge()");
    }
}
class Program
{
    static void Main()
    {
        Foo foo = new Foo();
        foo.hoge();
    }
}
```

コンストラクタ

■インスタンスコンストラクタ

・インスタンスを作成および初期化するために使用

```
public class Ham
{
    public Ham()
    {
    }
    public Ham(int weight)
      : this()
    {
    }
}
```

■プライベートコンストラクタ

・通常は、静的メンバだけを含むクラスで使用。
・クラスに 1 つ以上のプライベート コンストラクタがあり、パブリック コンストラクタがない場合、他のクラス (入れ子になったクラスを除く) はこのクラスのインスタンスを作成できない。

静的コンストラクタ

・静的データを初期化したり、1 回限りの特定の処理を実行したりする際に使用
・最初のインスタンスを作成する前、または静的メンバが参照される前に自動的に呼び出される。

・静的コンストラクタはアクセス修飾子をとらず、パラメータはありません

```
public class Ham
{
    public static int initial_num;
    static Ham()
    {
        initial_num = 100;
    }
}
class Program
{
    static void Main()
    {
        Console.WriteLine(Ham.initial_num);
    }
}
```

デストラクタ

・構造体には定義できない。クラスでだけ使用可能。
・修飾子をとらず、パラメータはない。

```
public class Person
{
    ~Person()
    {
        Console.WriteLine("I have destructed.");
    }
}
```

オブジェクト初期化子

・オブジェクトの初期化で使われる、public なプロパティに対して名前付きの値を渡すことができる。

```
Customer customer = new Customer
{
   ID = 1001,
   CompanyName = "foo",
   CompanyAddress = new Address()
};
```

コレクション初期化子

・複数のオブジェクトでコレクションを初期化して生成する処理を 1 行で済ませることができる。

```
List<Customer> custList = new List<Customer>
{
  new Customer {ID=1001, CompanyName="Foo"},
  new Customer {ID=1002, CompanyName="Goo"},
  new Customer {ID=1003, CompanyName="Hoo"},
}
```

オブジェクト指向

継承

・派生クラス名の後に、コロンと基本クラス名を追加して指定
・単一継承のみをサポート

```
public class Bar : Foo
{
}
```

基底クラス

■ base

・派生クラス内で基本クラスのメンバにアクセス

■ this

・クラスの現在のインスタンスを参照

構造体

・構文上ではクラスとほとんど変わらないが、クラスよりも制限される
・const または static と宣言されているフィールド以外は初期化できない
・既定のコンストラクタやデストラクタを宣言できない
・構造体は値型。オブジェクトを構造体から作成し、変数に代入すると、その変数には、構造体の値全体が含まれる。構造体を含む変数をコピーすると、すべてのデータがコピーされ、新しいコピーを変更しても、以前のコピーのデータは変更されない
・ラスで new 演算子を呼び出すと、ヒープに割り当てられます。ただし、構造体をインスタンス化した場合は、スタックに作成されます。

宣言

```
public struct Spam
{
    public int x, y;
    public Spam(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Program
{
    static void Main()
    {
        Spam spam;
        spam.x = 1;
        spam.y = 2;
        Console.WriteLine(spam.x);
    }
}
```

メソッド

拡張メソッド

・静的メソッドとして定義するが、呼び出すときはインスタンスのメソッドのように呼び出すことができる。
・最初のパラメータに、メソッドが操作する型を指定し、前には this 修飾子を付加する
・メソッドが定義されている名前空間に関する using ディレクティブを追加するだけで、特定の型の拡張メソッドを使用できるようになる
・標準クエリ演算子を使用するには、using System.Linq をコードに追加

必要な場合に限り注意して実装すること。既存の型を拡張する必要がある場合、可能であれば既存の型から派生した新しい型を作成する

拡張メソッドの例

```
using System;

// 静的メソッドとして定義するが、インスタンス メソッドの構文を使用して呼び出せる
// 入れ子になっていない、非ジェネリックの静的クラス内で定義
namespace ExtensionMethodTest
{
    static class Foo
    {
        // 引数の this に注目
        public static string hogehoge(this string s)
        {
            return "hogehoge" + s + "hogehoge";
        }
    }
}
namespace ConsoleApplicationTest
{
    using ExtensionMethodTest;  // using を使用して拡張メソッドをスコープに取り込む
    class Program
    {
        static void Main(string[] args)
        {
            string s = "Hello";
            Console.WriteLine(s.hogehoge()); // インスタンスのメソッドのように扱える
        }
    }
}
```

結果

```
hogehogeHellohogehoge
```

メソッドのオーバーライド

■ new キーワード

・基本クラス同名のメソッドを宣言する場合、new キーワードで明示する
・変数についても同様

```
 public partial class HogeForm : System.Windows.Forms.Form
{
    // Forms の ShowDialog メソッドを隠す new を使って宣言する
    public new void ShowDialog()
    {
        MessageBox.Show("");
        base.ShowDialog();
    }
```

メソッドのパラメータ

■ ref

・ref キーワードをつけると、引数の値ではなく参照が渡されます。
・メソッド パラメーターは、値型であるか参照型であるかにかかわらず、ref で修飾できます。 値型が参照渡しにされるときには、ボックス化が行われません。
・ref のパラメーターを使用するには、メソッド定義と呼び出し元のメソッドの両方に明示的に ref のキーワードを使用する必要があります。

```
class RefExample
{
```

```
    static void Method(ref int i)
    {
        i = i + 44;
    }
    static void Main()
    {
        int val = 1;
        Method(ref val);
        Console.WriteLine(val);
        // Output: 45
    }
}
```

■ out

・out キーワードを使用すると、引数が参照渡しされます。
・ref キーワードに似ていますが、ref の場合は、変数を初期化してから渡す必要があります。
・out パラメーターを使用するには、メソッド定義と呼び出し元のメソッドの両方で out キーワードを明示的に使用する必要があります。
・out 引数として渡す変数は、渡す前に初期化する必要はありませんが、呼び出されたメソッドでは、メソッドから制御を戻す前に値を代入する必要があります。

```
class OutExample
{
    static void Method(out int i)
    {
        i = 44;
    }
    static void Main()
    {
        int value;
        Method(out value);
        // value is now 44
    }
}
```

メソッドの呼び出し元

・メソッドの引数に属性をつけておくと、その引数に対して、コンパイラーが診断情報を渡してくれます。

```
public void TraceMessage(string message,
        [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
        [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
        [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    :
}
```

フィールド

const

・定数フィールドまたはローカル定数を宣言するには、const キーワードを使用
・定数には、数字、ブール値、文字列、または null 参照が含まれます

いずれかの時点で変わることが予想される情報を表すために定数を作成してはなりません。コンパイラは定数を伝達するため、ライブラリでコンパイルされた他のコードを再コンパイルして、変更点を反映することが必要になってしまいます

readonly

・フィールド宣言が readonly 修飾子を含む場合、宣言によって導入されるフィールドへの代

入は、宣言の一部として、または同じクラスのコンストラクター内でだけ行うことが出来る。
・使用するコンストラクターに応じて異なる値を持つことができます

プロパティ

・プロパティは、get アクセサと set アクセサのいずれか、または両方を表すコード ブロック
・set アクセサなしのプロパティは、読み取り専用、get アクセサなしのプロパティは、書き込み専用

get アクセサ、set アクセサ

・get アクセサ メソッドの呼び出しは、コンパイルでインライン展開される
・set アクセサは、プロパティの型の value という名前の暗黙のパラメータを使用す

```
public class Person
{
    private string name;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

自動実装

・自動実装するプロパティを使用すると、プロパティのアクセサで追加のロジックが必要ない場合に、プロパティ宣言がより簡単になる

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; private set; } // 読み取り専用
}
```

インターフェースのプロパティ

・アクセサには、本体がありません。
・アクセサの目的は、プロパティが読み取り / 書き込み、読み取り専用、または書き込み専用のいずれであるかを示す

```
public interface INameable
{
    string Name
    {
        get;
        set;
    }
}
```

デリゲート

・http://ufcpp.net/study/csharp/sp_delegate.html#definition

・デリゲートは用途も関数ポインターとほとんど同じで、 述語やイベントハンドラ等に利用
・関数ポインターと違い、 インスタンスメソッドを参照したり、 複数のメソッドを同時に参照する事が出来る
・デリゲートとはメソッドを参照するための型

定義

・定義したデリゲート型は、ユーザ定義のクラスや構造体と同じ 1 つの “ 型 ” として扱われる
・デリゲート型の変数には、 デリゲートの定義時に指定した物と同じ戻り値と引数リストを持つメソッドを代入する事が出来る

```
delegate 戻り値の型 デリゲート型名 ( 引数リスト );
using System;

// SomeDelegate という名前のデリゲート型を定義
delegate void SomeDelegate(int a);

class DelegateTest
{
  static void Main()
  {
    // SomeDelegate 型の変数にメソッドを代入。
    SomeDelegate a = new SomeDelegate(A);

    a(256); // デリゲートを介してメソッドを呼び出す。
            // この例では A(256) が呼ばれる。
  }
  static void A(int n)
  {
    Console.Write("A({0}) が呼ばれました。\n", n);
  }
}
```

機能

■インスタンスメソッドの代入

・デリゲートにはクラス (static) メソッドとインスタンス (非 static) メソッドのどちらでも代入する事が出来る

■複数のメソッドを代入

・デリゲートには += 演算子を用いることで、複数のメソッドを代入する事が出来る
・複数のメソッドを代入した状態で、デリゲート呼び出しを行うと、代入した全てのメソッドが呼び出される
・マルチキャストデリゲートの呼び出しは、+= で代入した順に逐次実行される

```
using System;

/// <summary>
/// メッセージを表示するだけのデリゲート
/// </summary>
delegate void ShowMessage();

class DelegateTest
{
  static void Main()
  {
    ShowMessage a = new ShowMessage(A);
    a += new ShowMessage(B);
    a += new ShowMessage(C);

    a();
  }

  static void A(){Console.Write("A が呼ばれました。\n");}
```

```
  static void B(){Console.Write("B が呼ばれました。\n");}
  static void C(){Console.Write("C が呼ばれました。\n");}
}
```

イベント

・.NET Framework クラス ライブラリでは、イベントは EventHandler デリゲートと EventArgs 基本クラスに基づいています。

パブリッシャとサブスクライバ

・イベントを送信をクラスをパブリッシャ、イベントを受信するクラスをサブスクライバと呼ぶ

IDE により自動で生成される

・イベントハンドラ

```
private void button1_Click(object sender, EventArgs e)
{
}
```

・Form1.Designer.cs ファイルの InitializeComponent メソッド内に、イベントをサブスクライブする必要があるコード行も自動生成される

this.button1.Click += new System.EventHandler(this.button1_Click);

シグネチャ

・戻値は Void
・1 つ目のパラメータは、sender という名前の Object 型。これは、イベントを発生させたオブジェクト
・2 つ目のパラメータは、e という名前の EventArgs 型または EventArgs の派生クラス。これは、イベント固有データ。

event

・event キーワードを使用して、パブリッシャー クラス内にイベントを宣言
・イベントは、宣言元 (パブリッシャー クラス) のクラスまたは構造体内でしか呼び出せない特殊なマルチキャスト デリゲート
・その他のクラスまたは構造体のイベントをサブスクライブすると、パブリッシャー クラスがイベントを発生させるときにイベント ハンドラー メソッドが呼び出されます

```
public class SampleEventArgs
{
    public SampleEventArgs(string s) { Text = s; }
    public String Text {get; private set;} // readonly
}
public class Publisher
{
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);
    public event SampleEventHandler SampleEvent;
    protected virtual void RaiseSampleEvent()
    {
        if (SampleEvent != null)
            SampleEvent(this, new SampleEventArgs("Hello"));
    }
}
```

EventHandler パターンに基づいてイベントを発行

- http://msdn.microsoft.com/ja-jp/library/w369ty8x.aspx
- ユーザー定義のクラス内のイベントは、値を返すデリゲートを含む、あらゆる有効なデリゲートを基にすることができますが、一般には、EventHandler を使用して、.NET Framework のパターンを基にすることを推奨

パブリッシャー クラスとサブスクライバー クラスの両方から参照できるスコープで、カスタム データのクラスを宣言

- イベントと共にカスタム データを送信する必要がない場合は、この手順を省略

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string s)
    {
        msg = s;
    }
    private string msg;
    public string Message
    {
        get { return msg; }
    }
}
```

パブリッシャー クラス内にデリゲートを宣言

- ジェネリック バージョンの EventHandler<TEventArgs> を使用する場合、この手順は省略

```
public delegate void CustomEventHandler(object sender, CustomEventArgs a);
```

パブリッシャー クラス内にイベントを宣言

- カスタムの EventArgs クラスがない場合、Event 型は非ジェネリック バージョンの EventHandler デリゲートになります。

```
public event EventHandler RaiseCustomEvent;
```

- 非ジェネリック バージョンの EventHandler を使用し、EventArgs から派生したカスタム クラスがある場合は、パブリッシャー クラス内でイベントを宣言

```
public event CustomEventHandler RaiseCustomEvent;
```

- ジェネリック バージョンを使用する場合、カスタム デリゲートは不要です。 代わりに、パブリッシャー クラス内でイベントの種類として EventHandler<CustomEventArgs> を指定

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

使用例

```
public event CustomEventHandler RaiseCustomEvent;
namespace DotNetEvents
{
```

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

    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
        }
    }
    class Publisher
    {
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;
        public void DoSomething()
        {
            OnRaiseCustomEvent(new CustomEventArgs("Did something"));
        }
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            EventHandler<CustomEventArgs> handler = RaiseCustomEvent;
            if (handler != null)
            {
                e.Message += String.Format(" at {0}", DateTime.Now.ToString());
                handler(this, e);
            }
        }
    }
    class Subscriber
    {
        private string id;
        public Subscriber(string ID, Publisher pub)
        {
            id = ID;
            pub.RaiseCustomEvent += HandleCustomEvent;
        }
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine(id + " received this message: {0}", e.Message);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Publisher pub = new Publisher();
            Subscriber sub1 = new Subscriber("sub1", pub);
            Subscriber sub2 = new Subscriber("sub2", pub);

            pub.DoSomething();

            Console.WriteLine("Press Enter to close this window.");
            Console.ReadLine();
        }
    }
}
```

インデクサ

・クラスまたは構造体のインスタンスに、配列と同様にインデックスを付けることができる
・プロパティと似ているが、インデクサのアクセサはパラメータを受け取る点が異なる

```
public class Foo<T>
{
    private T[] ary = new T[5];
    public T this[int i]
    {
        get
        {
            return ary[i];
```

```
        }
        set
        {
            ary[i] = value;
        }
    }
    public int Count { get { return 5; } }
}
static void Main()
{
    Foo<int> fint = new Foo<int>();
    for (int i = 0; i < fint.Count; i++)
    {
        fint[i] = i * i;
    }
    for (int i = 0; i < fint.Count; i++)
    {
        Console.Write(fint[i] + "\t");
    }
}
```

ジェネリック

・http://ufcpp.net/study/csharp/sp2_generics.html

・クラスやメソッドがクライアント コードで宣言され、インスタンス化されるまで、1 つ以上の型の指定を遅延させるクラスとメソッドを設計できる
・ランタイムのキャストやボックス化操作のコストやリスクを負わずに他のクライアント コードで使用できる単一のクラスを記述できる
・コードの再利用性、タイプ セーフ性、およびパフォーマンスを最大化するために使用する
・.NET Framework クラス ライブラリには、複数の新しいジェネリック コレクション クラスが System.Collections.Generic 名前空間に含まれています。これらのクラスは、System.Collections 名前空間の ArrayList などのクラスの代わりとして、できる限り使用する

```
public class GenelicList<T>
{
    T[] elements = new T[5];
    int count = 0;
    public void Add(T element)
    {
        if (count >= elements.Length)
        {
            T[] new_elements = new T[elements.Length * 2];
            for (int i = 0; i < elements.Length; i++)
            {
                new_elements[i] = elements[i];
            }
            elements = new_elements;
            new_elements = null;
        }
        elements[count] = element;
        count++;
    }
    public T Get(int index)
    {
        return elements[index];
    }
    public int Count
    {
        get
        {
            return count;
        }
    }
}
static void Main()
{
    GenelicList<int> int_list = new GenelicList<int>();
    for (int i = 0; i < 100; i++)
    {
        int_list.Add(i * i);
    }
```

```
    for (int i = 0; i < 100; i++)
    {
        System.Console.Write(int_list.Get(i) + "\t");
    }

    GenelicList<string> str_list = new GenelicList<string>();
    str_list.Add("abc");
    str_list.Add("def");
    str_list.Add("ghi");

    for (int i = 0; i < str_list.Count; i++)
    {
        System.Console.Write(str_list.Get(i) + "\t");
    }
}
```

型引数で与えた型でメソッド呼び出しをしたい場合などには、 where キーワードを使って型に制約条件を付加します

制約の与え方	説明
where T : struct	型 T は値型である
where T : class	型 T は参照型である
where T : new()	引数なしのコンストラクタを持つ。他の制約条件と同時に課す場合には、一番最後に指定する必要がある。
where T : [base class]	型 T は [base class] で指定された型を継承する。
where T : [interface]	型 T は [interface] で指定されたインターフェースを実装する。

```
static Type Max<Type>(Type a, Type b)
  where Type : IComparable
{
    return a.CompareTo(b) > 0 ? a : b;
}
```

ジェネリックインターフェース

・http://msdn.microsoft.com/ja-jp/library/kwtft8ak.aspx
・ジェネリック コレクション クラス、またはコレクション内の項目を表すジェネリック クラスにインターフェイスを定義すると便利。
・ジェネリック クラスでは、値型に対するボックス化およびボックス化解除の操作を回避するために、IComparable よりも IComparable<T> などのジェネリック インターフェイスを使用することをお勧めします。

■単一の型に対する制約として、次のように複数のインターフェイスを指定できます。

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

■１つのインターフェイスで、次のように複数の型パラメーターを定義できます。

```
interface IDictionary<K, V>
{
}
```

■クラスに適用される継承の規則が、インターフェイスにも適用されます。

```
interface IMonth<T> { }

interface IJanuary    : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>   : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

■具象クラスでは、次のように閉じた構造のインターフェイスを実装できます。

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

■ジェネリック クラスでは、インターフェイスで必要なすべての引数がクラスのパラメーター リストに指定されている場合、次のように、ジェネリック インターフェイスまたは閉じた構造のインターフェイスを実装できます。

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

■規定値 (default)

・変数を初期化するとき、 数値型の場合は 0 で、 参照型の場合は null で初期化する事がよくあります。 これら、0 や null などの値を既定値（default value）と呼びます。
・C# ジェネリックでは、既定値を得るために、 default(Type) というキーワードを用意しています。
・default(Type) は、 数値型に対しては 0、 参照型に対しては null になります。 また、構造体に対しては、 構造体の全てのメンバーに対して 0 または null で初期化したものを与えます。

アクセス修飾子

アクセス修飾子が指定されていない場合は、internal が既定値です。

private

・メンバ アクセス修飾子
・クラスの本体内か、メンバが宣言されている構造体の内部でだけアクセス可能
・同じ本体にある入れ子になったクラスも、プライベートなメンバにアクセスできる

public

・型および型メンバのためのアクセス修飾子
・パブリック メンバへのアクセスに関する制限はありません。

internal

・型および型メンバのためのアクセス修飾子
・同じアセンブリのファイル内でのみアクセス可能

protected

・メンバ アクセス修飾子

・そのクラス内で派生クラスからアクセスできる
・基本クラスのプロテクト メンバに派生クラスでアクセス可能なのは、派生したクラス型を使ってアクセスが行われる場合だけ

partial (**部分型定義**)

・部分型定義では、クラス、構造体、またはインターフェイスを複数のファイルに分割することを定義できる

Form1.cs

```
using System.Windows.Forms;
namespace TaskTraySample
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Program.cs

```
namespace TaskTraySample
{
    partial class Form1
    {
        /// <summary>
        /// 必要なデザイナ変数です。
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// 使用中のリソースをすべてクリーンアップします。
        /// </summary>
        /// <param name="disposing"> マネージ リソースが破棄される場合 true、破棄されない場合は false
です。</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows フォーム デザイナで生成されたコード

        /// <summary>
        /// デザイナ サポートに必要なメソッドです。このメソッドの内容を
        /// コード エディタで変更しないでください。
        /// </summary>
        private void InitializeComponent()
        {
        }
    }
}
```

アセンブリ

・アセンブリは、.NET Framework アプリケーションの基本ビルド ブロック
・簡単な C# アプリケーションを構築する場合、Visual Studio は、単一のポータブル実行可能 (PE) ファイルの形式 (具体的には EXE または DLL) でアセンブリを作成する
・リフレクションを使用すると、アセンブリに関する情報をプログラムによって取得できる

extern

同じアセンブリの 2 つのバージョンを 1 つのアプリケーションで使用できる

アセンブリ マニフェスト

- 静的であるか動的であるかにかかわらず、すべてのアセンブリは、アセンブリ内の要素の相互関係を記述したデータのコレクションを含んでいる
- 固有の内部バージョン番号と、格納されているすべてのデータとオブジェクト型の詳細を表すメタデータが含まれる

グローバル アセンブリ キャッシュ

- アセンブリをグローバル アセンブリ キャッシュに配置すると、複数のアプリケーションで共有

厳密な名前付きアセンブリ

- グローバル アセンブリ キャッシュに含める場合は、厳密な名前を付ける必要がある

頻用クラスライブラリ

IO

■ System.IO 名前空間

class	note
File	ファイルの作成、コピー、削除、移動、オープンのための静的メソッドを提供
FileInfo	ファイルを作成、コピー、削除、移動、および開くためのプロパティおよびインスタンス メソッドを提供し、FileStream オブジェクトを作成できるように
FileSystemInfo	FileInfo オブジェクトと DirectoryInfo オブジェクトの両方の基本クラス
FileStream	読み取り操作と書き込み操作をサポートするファイル用の Stream
Directory	
StringReader	文字列から読み取る TextReader

コレクション

■ System.Collections

コレクションを定義

・ArrayList
・Hashtable

■ System.Collections.Generic

ジェネリック コレクションを定義

・List
・Dictionary

同時実行

・lock ステートメント
・スレッドからコントロールを操作する

■ System.Threading.Tasks

同時実行コードおよび非同期コードを簡単に記述できるようにする

class	note
Task	非同期操作

■ System.Collections.Concurrent

スレッド セーフなコレクション

class	note
ConcurrentQueue<T>	スレッド セーフな先入れ先出し (FIFO) コレクション

正規表現

■ System.Text.RegularExpressions

■ Regex

```
var ptnLink = new Regex(@".+ad=(?<AREA_ID>[0-9]+).*");
var matche = ptnLink.Match(link);
if (matche.Groups.Count > 0)
{
   Debug.WriteLine(match.Groups["AREA_ID"].Value);
}
```

```
var ptnLink = new Regex(@".+ad=(?<AREA_ID>[0-9]+).*");
var matches = ptnLink.Matches(link);
if (matches.Count > 0)
{
   var match = matches[0];
   Debug.WriteLine(match.Groups["AREA_ID"].Value);
}
```

■正規表現による置換

```
var output = System.Text.RegularExpressions.Regex.Replace(input, @"^[ ]", "");
```

書式

■日付書式

標準の日付

・標準の日付と時刻の書式指定文字列

カスタム書式

・カスタム書式

```
// 書式付け
DateTime.Now.ToString("yyyyMMddHHmmss")
```

非同期プログラミング

・https://msdn.microsoft.com/ja-jp/library/hh191443.aspx?f=255&MSPPError=-2147217396
・非同期プログラミングを使用することによって、ボトルネックとなる幾つかの部分に、全体のパフォーマンスが引きずられる事のないように改良することができます
・しかしながら、従来の非同期アプリケーションのプログラミングはその性質から、コードの記述が難しく、デバッグでは問題の再現が難しくなりがちで、またそのコードのメンテナンスも複雑になっていました。
・開発者は同期処理と似たプログラム構造を維持したままこの非同期プログラムを実現することができます。

非同期による応答性の改善

・.NET Framework 4.5 および Windows ランタイム の API の一覧には、非同期のプログラミングをサポートするメソッドが含まれます。

アプリケーション領域	サポートされている、非同期のメソッドを含む API
Web アクセス	HttpClient、SyndicationClient
ファイルの処理	StorageFile、StreamWriter、StreamReader、XmlReader
イメージの処理	MediaCapture、BitmapEncoder、BitmapDecoder
WCF プログラミング	同期操作と非同期操作

非同期メソッドを使用すると、アプリケーションは UI に応答し続けます。たとえば、ウィンドウのサイズ変更や最小化を実行したり、アプリケーション処理の完了待たずに、アプリケーションを閉じたりできます。

```
//  - メソッド シグネチャは Async または async 修飾子を含みます
//  - 戻り値の型は次のいずれかになります Task もしくは Task<T>
//    Task<int> は整数値を返します
//  - 非同期メソッドの名前は、慣例により「Async」というサフィックスで終わります。
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    // GetStringAsync は Task<string> を返す。
    // getStringTask が待機しないため、AccessTheWebAsync は GetStringAsync からの最終結果に依存しない他の作業を続行できます。
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    // GetStringAsync に依存しない、何かしらの作業をここで行うことができる。
    // DoIndependentWork は、作業を実行し、呼び出し元に戻る同期メソッド
```

```
        DoIndependentWork();

        // await 演算子を使用してその進行を中断
        //  - AccessTheWebAsync は次に、ダウンロードする文字列の長さを計算しますが、メソッドに文字列が戻さ
れるまで、メソッドはその値を計算できません。
        //  - AccessTheWebAsync を呼び出したメソッドにコントロールを戻します
        //  - getStringTask が完了すると制御が再開する
        //  - タスクは、ダウンロードされた文字列の長さの整数値を生成することの保証を表します
        string urlContents = await getStringTask;

        // return 文は 整数値を指定する
        // どのメソッドも AccessTheWebAsync が桁数を参照するのを待つ
        return urlContents.Length;
    }
```

・非同期プログラムにおける制御フロー

非同期メソッド作成の簡素化

・ async と await のキーワードは非同期プログラミングの中核
・async および await を使用して定義する非同期メソッドは、async メソッドとして参照されます。