この項目は書きかけの項目です

C++ 基礎文法

INDEX

データ型

データ型の種類

- 論理型
    - bool型：true または false
- 整数型
    - unsigned short 型：0～65535（16bit）←約6万
    - short int 型：-32768～32767（16bit）←約±3万
    - unsigned int 型：0～4294967295（32bit）←約40億
    - int 型：-2147483648～2147483647（32bit）←約±20億／16進数（0xFFCC00等）も可／初期値
- 浮動小数点数型
    - float 型：有効数字7桁
    - double 型：有効数字15桁 ←初期値

- 文字型
    - char 型：1文字（シングルクォーテーションで囲む）
    - string 型：2文字以上（ダブルクォーテーションで囲む）

上記の他に以下のデータ型などもあり

- 列挙型（enum）：内部的には0、1、2...（int型）で処理
- 構造体（struct）：継承が出来ない、クラスに似たもの
- クラス（class）：classを使った参照型（データそのものではなくアドレスを保持）

検証

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo> //typeid()に必要
using namespace std;
class MyClass {}; //前方宣言が必要
int main() {
    //====================
    // 論理型 (bool型)
    //====================
    bool _bool = true;
    cout << _bool << "¥n"; //1
    cout << typeid(_bool).name() << "¥n"; //b (bool)
    //===============
    // 整数型
    //===============
    // ①unsigned short型 (0～65,535)
    unsigned short int _uShortInt = 65535;
    cout << _uShortInt << "¥n"; //65535
    cout << typeid(_uShortInt).name() << "¥n"; //t (unsigned
short int)

    // ②short int型 (-32,768～32,767)
    short int _shortInt = -32768;
    cout << _shortInt << "¥n"; //-32768
```

```
    cout << typeid(_shortInt).name() << "¥n"; //s (short int)

    // ③unsigned int型 (0～4,294,967,295)
    unsigned int _uInt = 4294967295;
    cout << _uInt << "¥n"; //4294967295
    cout << typeid(_uInt).name() << "¥n"; //j (unsigned int)

    // ④int型 (-2,147,483,648～2,147,483,647)
    int _int = -2147483648;
    cout << _int << "¥n"; //-2147483648
    cout << typeid(_int).name() << "¥n"; //i (int)

    int _int16 = 0xFFCC00; //16進数の場合
    cout << _int16 << "¥n"; //16763904
    cout << typeid(_int16).name() << "¥n"; //i (int)

    //================
    // 浮動小数点数型
    //================
    // ①float型 (有効数字7桁)
    float _float = 3.14159265358979323846264338327950288;
    cout << _float << "¥n"; //3.14159
    cout << typeid(_float).name() << "¥n"; //f (float)

    // ②double型 (有効数字15桁)
    double _double = 3.14159265358979323846264338327950288;
    cout << _double << "¥n"; //3.14159
    cout << typeid(_double).name() << "¥n"; //d (double)

    //================
    // 文字型
    //================
    // ①char型 (1文字)
    char _char = 'a';
    cout << _char << "¥n"; //a
    cout << typeid(_char).name() << "¥n"; //c (char)

    // ②string型 (2文字以上)
    string _string = "999";
```

```
    cout << _string << "¥n"; //999
    cout << typeid(_string).name() << "¥n"; //NSt7__cxx1112...

    //================
    // その他
    //================
    //列挙型（enum）
    enum ABC {AAA=1, BBB=2, CCC=3}; //列挙の定義（関数の外で定義
可）
    ABC _bbb = BBB; //列挙以外の値だとエラー
    cout << _bbb << "¥n"; //2（値を省略すると0から始まる順番が返
る）

    //構造体型（注意：private/publicを省略した場合public扱い）
    struct Name { //定義は先に記述する必要あり（関数の外で定義
可）
        string romaji;
        string kanji;
    };
    Name _taro = {"TARO", "太郎"};
    Name _hanako = {"HANAKO", "花子"};
    cout << _taro.romaji << "¥n" << _taro.kanji << "¥n";
    cout << _hanako.romaji << "¥n" << _hanako.kanji << "¥n";

    //クラス
    MyClass _myClass; //MyClassのオブジェクトを生成
    cout << typeid(_myClass).name() << "¥n"; //7MyClass

    //配列
    int _array[4]; //4個の空の要素を持つ配列の場合
    cout << _array << "¥n"; //0x7fff9849e280
    cout << typeid(_array).name() << "¥n"; //A4_i

    return 0;
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura

作成日:2016年05月12日
更新日:2017年04月24日

データ型の操作

データ型の調べ方：typeid()

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
class MyClass {}; //前方宣言が必要

int main() {
    cout << typeid(true).name() << "¥n"; //b (bool)
    cout << typeid(100).name() << "¥n"; //i (int)
    cout << typeid(2147483648).name() << "¥n"; //l (long int)
    cout << typeid(0.1).name() << "¥n"; //d (double)
    cout << typeid('1').name() << "¥n"; //c (char)
    cout << typeid("1").name() << "¥n"; //A2_c ←string
    MyClass _myClass;
    cout << typeid(_myClass).name() << "¥n"; //7MyClass
    return 0;
}
```

データ型のキャスト

1. 数値→bool型へ変換

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
int main() {
```

```
    bool _tmp = (bool)1;
    cout << _tmp << "¥n"; //1
    cout << typeid(_tmp).name() << "¥n"; //b (bool)
    return 0;
}
```

2. bool型→数値へ変換

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
int main() {
    int _tmp = (int)true;
    cout << _tmp << "¥n"; //1 (falseの場合0)
    cout << typeid(_tmp).name() << "¥n"; //i (int)
    return 0;
}
```

3. 数値→数値(整数の縮小変換)

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
int main() {
    int _tmp = 3350000; //intは-2,147,483,648～
2,147,483,647
    short int _tmp2= (short int)_tmp;
    cout << _tmp2 << "¥n"; //776 ←元のデータが失われる
}
```

4. 数値→数値(浮動小数点数の縮小変換)

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
int main() {
    double _double =
3.14159265358979323846264338327950288;
    float _float = (float)_double;
    cout << _float << "¥n"; //3.14159 ←元のデータが失われ
る
}
```

5. 数値⇔数値(拡張変換)

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo>  //typeid()に必要
using namespace std;
int main() {
    short int _tmp = 32767; //short int型は-32,768～32,767
    int _tmp2 = (int)_tmp + 1; //short int型→int型へ変換
    cout << _tmp2 << "¥n"; //32768
    cout << typeid(_tmp2).name() << "¥n"; //i (int)
}
```

6. string型→数値

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo> //typeid()に必要
#include <cstdlib> //atoiに必要
using namespace std;
```

```
int main() {
    string _tmp = "001";
    int _tmp2 = atoi(_tmp.c_str()); //string型→int型に変
換
    cout << _tmp2 << "¥n"; //1
    cout << typeid(_tmp2).name() << "¥n"; //i (int)
    return 0;
}
```

7. 数値→string型

```
//test.cpp
#include <iostream> //coutに必要
#include <typeinfo> //typeid()に必要
#include <sstream> //ostringstreamに必要
using namespace std;

int main() {
    int _tmp = 100;
    ostringstream _stream;
    _stream << _tmp;
    string _tmp2 = _stream.str();
    cout << _tmp2 << "¥n"; //"100"
    cout << typeid(_tmp2).name() << "¥n";
//NSt7__cxx1112basic…
    return 0;
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2016年05月12日
更新日:2017年04月25日

クラス

```
//test.cs
using System; //Console.WriteLine()に必要
class Test { //Mainは不可
    static void Main() { //自動的に最初に実行される
        //①インタンスの生成
        Rectangle _rectangle = new Rectangle();

        //②プロパティの更新
        _rectangle.width = 1920;
        _rectangle.height = 1080;
        //③プロパティの取得
        Console.WriteLine(_rectangle.width); //1920
        Console.WriteLine(_rectangle.height); //1080

        //④メソッドの実行
        Console.WriteLine(_rectangle.getArea()); //2073600
    }
}

class Rectangle { //長方形クラス
    //プロパティの定義･初期値の設定
    private int _width = 0; //privateは省略可
    private int _height = 0; //privateは省略可

    //コンストラクタは省略可（初期値はここで設定してもよい）
    public Rectangle() {}

    //メソッド群の定義
    public int width {
        get { return this._width; } //thisは省略可
        set { this._width = value; } //valueは決め打ち
    }
    public int height {
        get { return this._height; } //thisは省略可
        set { this._height = value; } //valueは決め打ち
    }
    public int getArea() { //面積を計算して値を返す
        return this._width * this._height; //thisは省略可
```

```
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月02日
更新日:2017年04月17日

基本クラスと派生クラス

```
//test.cs
using System; //Console.WriteLine()に必要

class Test { //メインクラス（Main()メソッドを含む）←Mainは不可
    static void Main() { //自動的に最初に実行される
        //派生クラスＡのインスタンス
        SubClassA _subclassA = new SubClassA();
        Console.WriteLine(_subclassA.pSuperClass); //”基本クラスのプロパティ”
        Console.WriteLine(_subclassA.pSubClassA); //”派生クラスＡのプロパティ”
        Console.WriteLine(_subclassA.mSuperClass()); //”基本クラスのメソッド”
        Console.WriteLine(_subclassA.mSubClassA()); //”派生クラスＡのメソッド”

        //派生クラスＢのインスタンス
        SubClassB _subclassB = new SubClassB();
        Console.WriteLine(_subclassB.pSuperClass); //”基本クラスのプロパティ”
        Console.WriteLine(_subclassB.pSubClassB); //”派生クラスＢのプロパティ”
        Console.WriteLine(_subclassB.mSuperClass()); //”基本クラスのメソッド”
        Console.WriteLine(_subclassB.mSubClassB()); //”派生クラ
```

```
スＢのメソッド”
    }
}

//基本クラス（スーパークラス）
class SuperClass {
    //①プロパティの定義
    string _pSuperClass = ”基本クラスのプロパティ”; //privateは
省略
    //②メソッド群の定義
    public string pSuperClass {
        get { return _pSuperClass; } //thisは省略
    }
    public string mSuperClass() {
        return ”基本クラスのメソッド”;
    }
}

//派生クラスＡ
class SubClassA : SuperClass { //基本クラスを継承（多重継承は不
可）
    //①プロパティの定義
    string _pSubClassA = ”派生クラスＡのプロパティ”; //privateは
省略
    //②メソッド群の定義
    public string pSubClassA {
        get { return _pSubClassA; } //thisは省略
    }
    public string mSubClassA() {
        return ”派生クラスＡのメソッド”;
    }
}

//派生クラスＡ
class SubClassB : SuperClass { //基本クラスを継承（多重継承は不
可）
    //①プロパティの定義
    string _pSubClassB = ”派生クラスＢのプロパティ”; //privateは
省略
```

```
    //②メソッド群の定義
    public string pSubClassB {
        get { return _pSubClassB; } //thisは省略
    }
    public string mSubClassB() {
        return ”派生クラスＢのメソッド”;
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月05日
更新日：2017年04月17日

名前空間

概要

- フォルダによる階層構造でファイルを管理するかのようにクラスを管理（但し論理的）
- .NET Framework にある System 名前空間はその下に Text、IO、Drawing などあり
- デフォルトでは無名の名前空間（global名前空間）が使用される
- 1つの名前空間の中に、クラス/構造体/デリゲート/列挙/インターフェース等を宣言できる他、別の名前空間を宣言することも可能

書式

```
namespace 名前空間名 {
    class ○○ {
    }
    ......
}
```

例文

```
//test.cs
using System; //Console.WriteLine()に必要

class Test { //メインクラス（Main()メソッドを含む）←Mainは不可
    static void Main() { //自動的に最初に実行される
        MyLibrary.MyClass _myClass = new MyLibrary.MyClass();
        Console.WriteLine(_myClass); //MyLibrary.MyClass
    }
}

namespace MyLibrary {
    //インターフェース等
    class MyClass {
        public MyClass() { //コンストラクタ
            //いろいろな処理
        }
        //いろいろなメソッド
    }
    //いろいろなクラス
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月23日
更新日：2017年04月17日

継承と委譲

概要

- GoF デザインパターンの Adapter パターン等で利用される
- 継承の場合は **:クラス名** を使い、委譲の場合は **new クラス名()** を使ってオブジ

ェクトを生成し、他のクラスの機能を利用する

継承版

```
//test.cs
using System;

class Test {
    static void Main() {
        ClassB _classB = new ClassB();
        _classB.MyMethod();
    }
}

class ClassA {
    public void MyMethod() {
Console.WriteLine("ClassA.MyMethod()"); }
}
class ClassB : ClassA {} //ClassAを継承
```

委譲版

```
//test.cs
using System;

class Test {
    static void Main() {
        ClassB _classB = new ClassB();
        _classB.MyMethod();
    }
}

class ClassA {
    public void MyMethod() {
Console.WriteLine("ClassA.MyMethod()"); }
```

```
}
class ClassB { //この内容だけが継承と異なる
    private ClassA _classA;
    public ClassB() { _classA = new ClassA(); } //コンストラクタ
でオブジェクト生成
    public void MyMethod() { _classA.MyMethod(); }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月26日
更新日：2017年04月17日

変数とスコープ

変数の種類

1. public 変数：全クラスからアクセス可能
2. protected 変数：同じクラスおよび派生クラス内でのみアクセス可能
3. private 変数：同じクラス内のみアクセス可能（省略すると private 扱い）
4. ローカル変数：メソッド内でのみアクセス可能（メソッド内で宣言したもの）

- その他「ブロックスコープ」等あり

public 変数

- 特徴
    - 全クラスからアクセスが可能
    - クラス定義の直後、コンストラクタの直前に定義
    - 通常は private 変数を利用し、アクセスには「get / set アクセサ」を使用する
- 書式

```
class クラス名 { //クラス定義
public データ型 変数名; //public変数宣言（初期化も可）
    public クラス名() {} //コンストラクタ（省略可）
    ......
```

- 悪い例

```
//test.cs
using System;
class Test {
    static void Main() {
        MyClass _myClass = new MyClass();
        Console.WriteLine(_myClass._p); //アクセス可（他人の変数
を勝手にいじる行為）
     }
}
class MyClass {
    public string _p = ”public変数”; //public宣言は冒頭でおこな
う
}
```

protected 変数

- 特徴
    - 同じクラスおよび派生クラス内でのみアクセス可能
    - 基本クラス（スーパークラス）の定義の直後、コンストラクタの直前に定義
- 書式

```
class 基本クラス { //スーパークラス定義
    protected データ型 変数名; //protected変数宣言（初期化も可）
    public クラス名() {} //コンストラクタ（省略可）
```

```
    ......
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        SubClass _subClass = new SubClass();
        Console.WriteLine(_subClass); //SubClass
        //Console.WriteLine(_subClass._pSuperClass); //error (アクセス不可)
     }
}

class SuperClass { //基本クラス
    protected string _pSuperClass = "SuperClass変数";
//protected変数宣言
}

class SubClass : SuperClass { //派生クラス
    public SubClass() {
        Console.WriteLine(_pSuperClass); //アクセス可能
    }
}
```

private 変数

- 特徴
    - 同じクラス内のみアクセス可能（省略すると private 扱い）
    - クラス定義の直後、コンストラクタの直前に定義
    - 「他人の変数を勝手にいじってはいけない」というルールに則り、インスタンス変数は通常、private 変数とし、外部からは「get / set アクセサ」を使ってアクセスする

- 書式

```
class クラス名 { //クラス定義
private データ型 変数名; //private変数宣言（初期化も可）
←privateは省略可
    public クラス名() {} //コンストラクタ（省略可）
    ......
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        MyClass _myClass = new MyClass();
        Console.WriteLine(_myClass.P); //アクセス可（≠他人の変
数を勝手にいじる行為）
     }
}

class MyClass {
    private string _p = ”private変数”; //private宣言は冒頭でおこ
なう
    public string P {
        get { return _p; }
        set { _p = value; }
    }
}
```

ローカル変数

- 特徴
    - ①メソッド ② for ③ foreach 文内で宣言
    - 宣言した ①メソッド ② for ③ foreach 文内でのみアクセス可能

1. メソッド内で宣言する場合

```
//test.cs
using System;
class Test {
    static void Main() {
        MyClass _myClass = new MyClass();
        _myClass.MyMethod();
    }
}

class MyClass {
    private string _string = ”private変数”;
    public MyClass() { //コンストラクタ
        Console.WriteLine(_string); //private変数（ここは
thisは無くても良い）
    }
    public void MyMethod() {
        string _string = ”ローカル変数”; //ローカル変数宣
言
        Console.WriteLine(_string); //ローカル変数
        Console.WriteLine(this._string); //private変数（こ
こではthisが必須）
    }
}
```

2. for 文内で宣言する場合（foreach 文も同様）

```
//test.cs
using System;
class Test {
    static void Main() {
        new MyClass();
     }
}
class MyClass {
```

```
    private int _i = 999; //private変数
    public MyClass() { //コンストラクタ
        for (int _i=0; _i<=5; _i++) { //ローカル変数宣言
            Console.WriteLine("A: " + _i); //0、1、2、...、5
            Console.WriteLine("B: " + this._i); //999 (private変
数)
        }
        //Console.WriteLine("C: " + _i); //error (ロカール変数は
アクセス不可)
        Console.WriteLine("C: " + this._i); //999 (private変数は
thisが必須)
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月20日
更新日:2017年04月17日

アクセサ（getter / setter）

読み書き可能なプロパティ

```
//test.cs
using System;
class Test {
    static void Main() {
        Nishimura _nishimura = new Nishimura();
        Console.WriteLine(_nishimura.Age); //49
        _nishimura.Age = 50; //値を変更可能
        Console.WriteLine(_nishimura.Age); //50
    }
}

class Nishimura {
```

```
    int _age = 49; //privateは省略
    public int Age {
        get { return _age; } //thisは省略
        set { _age = value; } //thisは省略 ←valueは予め定義された変数（決め打ち）
    }
}
```

読み取り専用のプロパティ

```
//test.cs
using System;
class Test {
    static void Main() {
        Nishimura _nishimura = new Nishimura();
        Console.WriteLine(_nishimura.Age); //49
        //_nishimura.Age = 50; //error（値の変更は不可）
        _nishimura.NextYear();
        Console.WriteLine(_nishimura.Age); //50
    }
}

class Nishimura {
    int _age = 49; //privateは省略
    public int Age {
        get { return _age; } //thisは省略
        private set {} //外部からアクセス不可（読み取り専用にする）
    }
    public void NextYear() { _age += 1; } //クラスの内部からに限り変更可能
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura

作成日:2015年11月09日
更新日:2017年04月17日

演算子

```
//test.cs
using System;

class Test {
    static void Main() {
        Console.WriteLine(3 + 2); //5（可算）
        Console.WriteLine(5 - 8); //-3（減算）
        Console.WriteLine(3 * 4); //12（乗算）
        Console.WriteLine(1 + 2 * 3 - 4 / 2); //5（複雑な計算）
        Console.WriteLine(63 % 60); //3（余剰）

        // 除算（注意が必要です）
        Console.WriteLine(8 / 3); //2(除算）←整数同士の場合、余
りは切り捨てられる
        Console.WriteLine(8 / 3.0); //2.66666666666667（小数点第
14位までの値＝double型）

        float _float = (float)8.0 / 3;
        Console.WriteLine(_float); //2.666667（小数点第6位までの
値）

        decimal _decimal = (decimal)8.0 / 3;
        Console.WriteLine(_decimal);
//2.6666666666666666666666666667（第28位まで）

        // 後ろに付けるインクリメント（デクリメント）
        // _hoge++（_hoge--）が返す値は、加算（減算）する前の
_hogeの値です
        int _hoge = 0;
        int _piyo = _hoge++; //デクリメントの場合_hoge--
        Console.WriteLine(_hoge); //1（デクリメントの場合-1）
```

```
        Console.WriteLine(_piyo); //0（デクリメントの場合0）

        // 前に付けるインクリメント（デクリメント）
        // ++_hoge（--_hoge）が返す値は、加算（減算）後の_hogeの
値です
        _hoge = _piyo = 0;
        _piyo = ++_hoge; //デクリメントの場合--_hoge
        Console.WriteLine(_hoge); //1（デクリメントの場合-1）
        Console.WriteLine(_piyo); //1（デクリメントの場合-1） ←
注目
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月06日
更新日:2017年04月17日

定数

通常の定数

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        const float PI = 3.14159f; //staticは記述しない（注意）
        Console.WriteLine(PI); //=> 3.14159
        //PI = 3.14; //error（変更不可）
    }
}
```

静的定数(メンバ定数)

- 構文

```
class クラス名 {
    public const float 定数名 = 値; //staticは記述しない
    ...
}

#アクセス方法
クラス名.定数名
```

- 例文

```
//test.cs
using System;

//メインクラス
class Test {
    static void Main() { //自動的に最初に実行される
        Console.WriteLine(MyMath.PI); //=> 3.14159
        //MyMath.PI = 3.14; //error（変更不可）
    }
}

//カスタムクラス（MyMath）
class MyMath {
    public const float PI = 3.14159f; //staticは記述しない（注
意）
    public MyMath() {} //コンストラクタ
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月06日
更新日：2017年04月17日

メソッド

基本構文

```
アクセス修飾子 [static] 戻り値のデータ型 メソッド名([データ型 引数, ...]) {
    [return 戻り値;]
}
```

アクセス修飾子

1. public : 全クラスからアクセス可能
2. protected : 同じクラスおよび派生クラス内でのみアクセス可能
3. private : 同じクラス内のみアクセス可能（省略すると private 扱い）
4. internal : アセンブリ内でのみアクセス可能

- static : 静的メソッド＝クラスメソッド

基本例文

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        MyClass _myClass = new MyClass();
        Console.WriteLine(_myClass.Tashizan(1,10)); //55
        Console.WriteLine(_myClass.Tashizan(1,100)); //5050
    }
}

class MyClass {
    //〇～〇までの値を足した合計を返す
    public int Tashizan(int _start, int _end) {
        int _result = 0; //ローカル変数
```

```
        for (int _i = _start; _i <= _end; _i++) {
            _result += _i;
        }
        return _result;
    }
}
```

Main()メソッド

- 特徴
    - 記述できるクラスは1つだけ（複数存在するとエラー）
    - 自動的に最初に実行
    - static キーワードが必須（オブジェクトの生成をしなくても Main() を呼び出す必要がある為）
    - 値を返したり、引数を渡すことも可能
- 例文

```
//test.cs
using System;
class Test { //メインクラス（Mainは不可）
    static void Main() { //自動的に最初に実行される
        Method(); //”Test.Method()”
    }
    static void Method() { //staticなメソッドならMain()から呼び出せる
        Console.WriteLine(”Test.Method()”);
    }
}
```

コンストラクタ

- 書式】

```
class クラス名 {
    public クラス名([型① 引数①, 型② 引数②, ...]) { //コンス
トラクタは省略可
        ......
    }
    ......
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        Point _point = new Point(100,150); //ここでコンストラク
タを呼び出す
        Console.WriteLine(_point.X); //100
        Console.WriteLine(_point.Y); //150
     }
}

class Point {
    private int _x, _y;
    public Point(int _x=0, int _y=0) { //コンストラクタ
        this._x = _x;
        this._y = _y;
    }
    public int X {
        get { return _x; }
        set { _x = value; }
    }
    public int Y {
        get { return _y; }
        set { _y = value; }
    }
}
```

静的メソッド（クラスメソッド）

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        Console.WriteLine(Math.Pow(2,0)); //1 (2の0乗)
        Console.WriteLine(Math.Pow(2,1)); //2 (2の1乗)
        Console.WriteLine(Math.Pow(2,8)); //256 (2の8乗)
    }
}

class Math {
    public static long Pow(int arg1, int arg2) {
        if (arg2 == 0) { return 1; } //0乗対策
        long _result = arg1;
        for (int i=1; i<arg2; i++) {
            _result = _result * arg1;
        }
        return _result;
    }
}
```

デフォルト値付き引数

- オプション引数（引数は省略可）

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        MyClass _myClass = new MyClass();
        _myClass.AddPoint(); //→1
        _myClass.AddPoint(10); //→11
    }
```

```
}

class MyClass {
    private int _point = 0;
    public void AddPoint(int arg = 1) { //初期値を1とした場合
        _point += arg;
        Console.WriteLine(_point);
    }
}
```

可変長引数

- 引数を固定の数ではなく任意の数にすることが可能

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        MyClass _myClass = new MyClass();
        _myClass.Sum(1,1); //2 (1+1)
        _myClass.Sum(1,2,3,4,5); //15 (1+2+3+4+5)
    }
}

class MyClass {
    public void Sum(params int[] args) {
        int _result = 0; //ローカル変数
        foreach (int tmp in args) {
            _result += tmp;
        }
        Console.WriteLine(_result);
    }
}
```

名前付き引数

- 引数名を指定してメソッドを呼び出す（任意の順序で引数を渡すことが可能）

```
//test.cs
using System;
class Test {
    static void Main() { //自動的に最初に実行される
        MyClass _myClass = new MyClass();
        _myClass.Rect(endX:100, endY:100); //面積:10000m2
        _myClass.Rect(10,10,100,100); //面積:8100m2
    }
}

class MyClass {
    public void Rect(int startX=0, int startY=0, int endX=0, int
endY=0) {
        int _result = (endX - startX) * (endY - startY);
        Console.WriteLine("面積:" + _result + "m2");
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2016年11月25日
更新日:2017年04月18日

匿名メソッド

```
//test.cs
using System;
class Test { //メインクラス
    static void Main() { //自動的最初に実行される
        MyClass _myClass = new MyClass();
        _myClass.Move(1); //→
        _myClass.change();
```

```
        _myClass.Move(3); //←←←
    }
}
class MyClass {
    public delegate void Method(int arg); //デリゲートの宣言（名
前＝Methodは任意）
    public Method Move; //匿名メソッドを格納する変数Move（＝メソ
ッド名）
    private bool _right = true;
    public MyClass() { //コンストラクタ
        //匿名メソッドの定義
        Move = delegate(int arg) {
            string _tmp = "";
            for (int i=0; i<arg; i++) _tmp += "→";
            Console.WriteLine(_tmp);
        };
    }
    public void change() {
        _right = ! _right;
        if (_right) {
            Move = delegate(int arg) {
                string _tmp = "";
                for (int i=0; i<arg; i++) _tmp += "→";
                Console.WriteLine(_tmp);
            }; //...匿名メソッドの再定義（メソッドの内容を変更）
        } else {
            Move = delegate(int arg) { //匿名メソッドの再定義
（メソッドの内容を変更）
                string _tmp = "";
                for (int i=0; i<arg; i++) _tmp += "←";
                Console.WriteLine(_tmp);
            };
        }
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura

作成日:2015年11月09日
更新日:2017年04月18日

ラムダ式

- 匿名メソッドを「ラムダ式」に置き換えたバージョン

```
//test.cs
using System;
class Test { //メインクラス
    static void Main() { //自動的最初に実行される
        MyClass _myClass = new MyClass();
        _myClass.Move(1); //→
        _myClass.change();
        _myClass.Move(3); //←←←
    }
}
class MyClass {
    public delegate void Method(int arg); //デリゲートの宣言（名前＝Methodは任意）
    public Method Move; //匿名メソッドを格納する変数Move（＝メソッド名）
    private bool _right = true;
    public MyClass() { //コンストラクタ
        Move = (int arg) => { //匿名メソッドの代りにラムダ式を利用
            string _tmp = "";
            for (int i=0; i<arg; i++) _tmp += "→";
            Console.WriteLine(_tmp);
        }; //メソッドの内容を変更
    }
    public void change() {
        _right = ! _right;
        if (_right) {
            Move = (int arg) => { //匿名メソッドの代りにラムダ式を利用
```

```
                string _tmp = "";
                for (int i=0; i<arg; i++) _tmp += "→";
                Console.WriteLine(_tmp);
            }; //メソッドの内容を変更
        } else {
            Move = (int arg) => { //匿名メソッドの代りにラムダ式
を利用
                string _tmp = "";
                for (int i=0; i<arg; i++) _tmp += "←";
                Console.WriteLine(_tmp);
            }; //メソッドの内容を変更
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月09日
更新日:2017年04月18日

静的メンバ(static)

- 静的メンバはクラスをインスタンス化せずにアクセスが可能

```
//test.cs
using System;

class Test {
    static void Main() { //インスタンス化せずにOSから自動的に呼
ぶ出すため
        Console.WriteLine(Math.PI); //3.14159265358979 ←静的変
数の呼び出し
        Console.WriteLine(Math.Pow(2,8)); //256 (2の8乗)  ←静的
メソッドの実行
    }
```

```
}

class Math { //独自クラス
    //静的変数
    public static double PI = 3.14159265358979;

    //静的メソッド
    public static long Pow(int arg1, int arg2) {
        if (arg2 == 0) { return 1; } //0乗対策
        long _result = arg1;
        for (int i=1; i<arg2; i++) {
            _result = _result * arg1;
        }
        return _result;
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月21日
更新日:2017年04月18日

if 文

基本例文

```
//test.cs
using System;
class Test { //メインクラス
    static void Main() { //自動的最初に実行される
        int _age = 49;
        if (_age <= 20) {
            Console.WriteLine("20歳以下");
        } else if (_age <= 40) {
            Console.WriteLine("21～40歳");
```

```
        } else if (_age <= 60) {
            Console.WriteLine("41～60歳"); //これが出力される
        } else {
            Console.WriteLine("61歳以上");
        }
    }
}
```

論理積（AND）

1. 論理演算子（&&）を使う方法

```
if (条件式① && 条件②) {
    処理A ←条件式① かつ 条件式② の両方がtrueの場合に実行
} else {
    処理B
}
```

2. ifのネストを使う方法

```
if (条件式①) {
    if (条件②) {
        処理A ←条件式① かつ 条件式② の両方がtrueの場合に実行
    } else {
        処理B
    }
} else {
    処理B
}
```

論理和（OR）

1. 論理演算子(||)を使う方法

```
if (条件式① || 条件②) {
    処理A ←条件式①または条件式②の両方がtrueの場合に実行
} else {
    処理B
}
```

2. ifのネストを使う方法

```
if (条件式①) {
    処理A ←条件式①がtrueの場合に実行
} else if (条件②) {
    処理A ←条件式②がtrueの場合に実行
} else {
    処理B
}
```

排他的論理和(XOR)

1. ^ 演算子を使う方法

```
//test.cs
using System;
class Test { //メインクラス
    static void Main() { //自動的最初に実行され
        bool _a = true, _b = false;
        if (_a ^ _b) {
            Console.WriteLine("どちらか一方だけtrueです");
        } else {
            Console.WriteLine("両方共にtrueかfalseです");
        }
    }
```

```
}
```

2. ^演算子を使わない方法

```
//test.cs
using System;
class Test {
    static void Main() {
        bool _a = true, _b = false;
        if ((_a || _b) && !(_a && _b)) {
            Console.WriteLine(”どちらか一方だけtrueです”);
        } else {
            Console.WriteLine(”両方共にtrueかfalseです”);
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月09日
更新日:2017年04月18日

三項演算子

比較式が1つの場合

- 構文

```
データ型 変数名 =（比較式）？（true時の返り値）:（false時の返り値);
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _age = 49;
        string _result = (_age < 60) ? ”現役” : ”退職”;
        Console.WriteLine(_result); //”現役”
    }
}
```

比較式が複数の場合

- 構文

```
データ型 変数名 =（比較式①）?（①がtrueの場合の返り値）: //①がfalseの場合↓
変数名 =（比較式②）?（②がtrueの場合の返り値）: //②がfalseの場合↓
変数名 =（①②の全てがfalseの場合の返り値);
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _age = 49;
        string _result = (_age < 20) ? ”未成年” :
        _result = (_age < 60) ? ”現役” :
        _result = ”退職”;
        Console.WriteLine(_result); //”現役”
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月09日
更新日:2017年04月18日

switch 文

基本サンプル

```
//test.cs
using System;
class Test {
    static void Main() {
        string _name = "TAKASHI";
        switch (_name) { //判別式には「整数型」「文字型」しか使
えない!
            case "TAKASHI" :
                Console.WriteLine("父");
                break;
            case "HANAKO" :
                Console.WriteLine("母");
                break;
            case "TARO" :
                Console.WriteLine("長男");
                break;
            case "JISO" :
                Console.WriteLine("次男");
                break;
            default:
                Console.WriteLine("家族以外");
                break; //defaultのbreakは省略不可（注意）
        }
    }
}
```

注意その1：判別式にbool型が使えない

- 判別式に指定可能なもの
    - byte 型、short 型、int 型 などの整数型（浮動小数点型は指定不可）
    - char 型、string 型といった文字型
- 悪い例（エラー発生）

```
//test.cs
using System;
class Test {
    static void Main() {
        int _age = 49;
        switch (true) { //bool型はエラー（注意）
            case _age < 20 :
                Console.WriteLine(”未成年”);
                break;
            default:
                Console.WriteLine(”成人”);
                break;
        }
    }
}
```

注意その2：フォールスルーの禁止規則

- C#では、下記のように case で何か処理をしておきながら break 文を書かないで次の case の処理に入っていくことは不可（フォールスルーの禁止規則）

```
case ”〇〇” : 何か処理; //何か処理をしておきながらbreakを書かないとエラー
case ”□□” : 何か処理; break;
```

- 良い例

```
//test.cs
using System;
class Test {
    static void Main() {
        string _name = ”JIRO”;
        switch (_name) {
            case ”TAKASHI” : //breakが無いと次のcaseも処理
            case ”HANAKO” :
                Console.WriteLine(”親”);
                break;
            case ”TARO” : //breakが無いと次のcaseも処理
            case ”JIRO” :
                Console.WriteLine(”子”);
                break;
            default:
                Console.WriteLine(”家族以外”);
                break; //defaultのbreakは省略不可
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月10日
更新日:2017年04月18日

for 文

基本構文

```
for（①初期化; ②ループ判定式; ③更新処理）{
    繰り返す処理
```

```
}
```

ループカウンタ(ループ制御変数)の宣言位置

1. for 文の中で宣言

```
//test.cs
using System;
class Test {
    static void Main() {
        for (int i=0; i<10; i++) { //ここでint型を宣言すると...
            Console.WriteLine(i); //0,1,2,3,4,5,6,7,8,9
        }
        //Console.WriteLine(i); //error (for文の外では使用不可)
    }
}
```

2. for 文の外でループ制御変数を宣言する

```
//test.cs
using System;
class Test {
    static void Main() {
        int _i; //ここでint型を宣言すると...
        for (_i=0; _i<10; _i++) {
            Console.WriteLine(_i); //0,1,2,3,4,5,6,7,8,9
        }
        Console.WriteLine(_i); //10 (for文の外でも有効)
    }
}
```

ループカウンタを〇つずつアップする

```
//test.cs
using System;
class Test {
    static void Main() {
        for (int i=0; i<50; i+=5) { //5つずつアップする場合...
            Console.WriteLine(i); //0,5,10,15,20,25,30,35,40,45
        }
    }
}
```

for 文のネスト

```
//test.cs
using System;
class Test {
    static void Main() {
        for (int i=1; i<=5; i++) {
            for (int j=1; j<=5; j++) {
                Console.WriteLine("x" + i + "y" + j);
//x1y1,x1y2,....,x5y4,x5y5
            }
        }
    }
}
```

無限ループと break 文

```
//test.cs
using System;
class Test {
    static void Main() {
```

```
        int _count = 0;
        for (;;) { //①初期化 ②ループ判定式 ③更新処理...の全て
を省略すると無限ループ
            _count++;
            if (_count > 100) break; //ループを終了
            Console.WriteLine(_count); //1,2,....,99,100
        }
        Console.WriteLine("for文終了"); //★
    }
}
```

for 文と continue 文

```
//test.cs
using System;
class Test {
    static void Main() {
        for (int i=1; i<=20; i++) { //iは1,2,...19,20
            if ((i % 3) != 0) { //3で割って余りが0ではない（＝3
の倍数ではない）場合
                continue; //for文の残処理をスキップしてfor文の次
の反復を開始する
            }
            Console.WriteLine(i); //3,6,9,12,15,18（3の倍数）
        }
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月10日
更新日：2017年04月19日

foreach 文

基本構文

```
foreach (データ型 変数名 in 配列等) {
    Console.WriteLine(変数名);
}
```

配列(1次元)の場合

```
//test.cs
using System;
class Test {
    static void Main() {
        string[] _array = {"A","B","C","D"};
        foreach (string value in _array) {
            Console.WriteLine(value); //"A"→"B"→"C"→"D"
        }
    }
}
```

配列(2次元)の場合

```
//test.cs
using System;
class Test {
    static void Main() {
        string[,] _array = {
            {"x0y0","x1y0","x2y0"}, //0行目
            {"x0y1","x1y1","x2y1"}  //1行目
        };
        foreach (string value in _array) {
            Console.WriteLine(value);
//"x0y0"→"x1y0"→"x2y0"→"x0y1"→"x1y1"→"x2y1"
        }
```

```
    }
}
```

配列（ジャグ配列）の場合

```
//test.cs
using System;
class Test {
    static void Main() {
        dynamic[][] _array = new dynamic[2][];
            _array[0] = new dynamic[]{"A","あ"};
            _array[1] = new dynamic[]{"I","い"};
        foreach (object[] theArray in _array) {
            foreach (object value in theArray) {
                Console.WriteLine(value); //"A"→"あ"、"I"→"い"
            }
            Console.WriteLine(); //オプション（改行）
        }
    }
}
```

動的配列（ArrayList）の場合

```
//test.cs
using System;
using System.Collections; //ArrayListに必要
class Test {
    static void Main() {
        ArrayList _array = new ArrayList();
        _array.Add("TAKASHI");
        _array.Add(49);
        foreach (object value in _array) {
            Console.WriteLine(value); //"TAKASHI"→49
        }
```

```
    }
}
```

動的配列(List)の場合

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { ”A”, ”B” };
        foreach (string value in _list) {
            Console.WriteLine(value); //”A”→”B”
        }
    }
}
```

連想配列の場合

```
//test.cs
using System;
using System.Collections.Generic; //Dictionaryに必要
class Test {
    static void Main() {
        Dictionary<string, string> _dic = new Dictionary<string,
string>() {
            {”A”, ”あ”},    {”I”, ”い”}
        };
        foreach (KeyValuePair<string, string> tmp in _dic) {
            Console.WriteLine(tmp.Key + ”:” + tmp.Value); //A:あ
→ I:い
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2016年01月21日
更新日:2017年04月19日

while 文

while 文

- 構文

```
while（ループ判定式）{
    繰り返す処理
}
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _i = 0;
        while (_i < 10) { //ループ判定式にはbool型しか使えない
            Console.WriteLine(_i); //0,1,2,3,4,5,6,7,8,9
            _i++;
        }
        Console.WriteLine(_i); //10（変数はまだ有効）
    }
}
```

do...while 文

- 構文

```
do {
    繰り返す処理 ←ループ判定式がfalseの場合でも最低１回は実行さ
れる
} while(ループ判定式);
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _i = 0;
        do {
            Console.WriteLine(_i); //0 ←ループ判定式はfalseだが
１回実行される
            _i++;
        } while(_i < 0);
    }
}
```

while 文と break 文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _count = 0;
        while (true) { //ループ判別式をtrueにすると無限ループに
            _count++;
            if (_count > 100) {
                break; //break文を使ってループを終了→★
```

```
            }
            Console.WriteLine(_count); //1,2,....,99,100 (1～100
までを出力)
        }
        Console.WriteLine("while文終了"); //★
    }
}
```

while 文と continue 文

```
//test.cs
using System;
class Test {
    static void Main() {
        int _i = 1;
        while (_i <= 20) {
            if ((_i % 3) != 0) { //3で割って余りが0ではない（＝3
の倍数ではない）場合
                _i++;
                continue; //while文の残処理をスキップしてwhile文
の次の反復を開始する
            }
            Console.WriteLine(_i); //3,6,9,12,15,18 (3の倍数を出
力)
            _i++;
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月10日
更新日:2017年04月19日

配列

- C# では配列宣言後の要素数変更は不可

1次元配列の作成

- 構文（他にも var キーワードを使ってデータ型を省略した定義も可能）

```
データ型[] 変数名 = new データ型[要素数];
データ型[] 変数名 = new データ型[]{要素①,要素②,...};
データ型[] 変数名 = {要素①,要素②,...}; //簡単
```

- 例文

```
dynamic[] _array1 = new dynamic[4]; //4つの空の要素（動的型）を
持つ配列を作成
string[] _array2 = new string[]{”A”,”B”,”C”,”D”};
string[] _array3 = {”A”,”B”,”C”,”D”}; //簡単
```

2次元配列（四角配列）の作成

- 構文

```
データ型[,] 変数名 = new データ型[行数,列数]; //縦x横の空の要素
を持つ２次元配列
データ型[,] 変数名 = {{1行目の配列},{2行目の配列},...};
```

1. new 演算子を使う方法（≒5行x4列のコインロッカー）

```
//test.cs
```

```
using System;
class Test {
    static void Main() {
        string[,] _coinlocker = new string[5,4];
        _coinlocker[0,0] = "1083"; //0,0の値
        _coinlocker[0,1] = "7777"; //0,1の値
        _coinlocker[2,1] = "0135"; //2,1の値
        _coinlocker[4,3] = "1234"; //4,3の値
    }
}
```

2. 配列リテラルを使う方法（≒5行x4列のコインロッカー）

```
//test.cs
using System;
class Test {
    static void Main() {
        string[,] _coinlocker =
        {{"1083","7777","",""}, //0行目
        {"","","",""},          //1行目
        {"","0135","",""},      //2行目
        {"","","",""},          //3行目
        {"","","","1234"}};     //4行目

        //確認
        Console.WriteLine(_coinlocker[0,0]); //"1083"
        Console.WriteLine(_coinlocker[0,1]); //"7777"
        Console.WriteLine(_coinlocker[2,1]); //"0135"
        Console.WriteLine(_coinlocker[4,3]); //"1234"
    }
}
```

配列の配列（ジャグ配列）の作成

- 構文（それぞれの配列の長さは異なるものにできる）

```
①データ型[][] 変数名 = new データ型[要素数][];
②データ型[][] 変数名 = new データ型[][]{new データ型[]{配列
①},...};
```

1. ジャグ配列の宣言→後で値を割り当てる方法

```
//test.cs
using System;
class Test {
    static void Main() {
        dynamic[][] _array = new dynamic[4][];
        _array[0] = new dynamic[]{"A","あ","ア"}; //配列リ
テラルは不可
        _array[1] = new dynamic[]{"I","い","イ"};
        _array[2] = new dynamic[]{"U","う","ウ"};
        _array[3] = new dynamic[]{"E","え","エ"};
    }
}
```

2. ジャグ配列の宣言と同時に値を割り当てる方法

```
//test.cs
using System;
class Test {
    static void Main() {
        dynamic[][] _array = new dynamic[][]{
        new dynamic[]{"A","あ","ア"},
        new dynamic[]{"I","い","イ"},
        new dynamic[]{"U","う","ウ"},
        new dynamic[]{"E","え","エ"}
        };
        foreach (dynamic[] theArray in _array) { //確認
(コマンドライン版の例)
```

```
                foreach (object theValue in theArray) {
                    Console.WriteLine(theValue);
                }
            }
        }
    }
```

配列の Length プロパティ

```
//test.cs
using System;
class Test {
    static void Main() {
        string[] _array = {”A”,”B”,”C”,”D”};
        for (int i=0; i<_array.Length; i++) { //配列の要素の数
            Console.WriteLine(_array[i]);
        }
    }
}
```

文字列→配列

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = ”A,B,C,D”; //「,」区切りの文字列
        string[] _array = _string.Split(’,’); //「,」区切りで分
割して配列化
        foreach (string value in _array) {
            Console.WriteLine(value); //”A”→”B”→”C”→”D”
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2016年01月21日
更新日:2017年04月19日

動的配列(List)

概要

- 配列と異なり List は要素の数を変更したり追加・削除などが可能
- 動的配列(ArrayList)の.NET framework 2.0 対応版

作成

- 構文

```
List<データ型> 変数名 = new List<データ型>(); //空のListを作成
List<データ型> 変数名 = new List<データ型>(数); //指定数の空の要素を持つList作成
List<データ型> 変数名 = new List<データ型>() { 要素①,要素②,... };
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { "A", "B" };
        foreach (object value in _list) {
            Console.WriteLine(value); //"A"→"B"
```

```
        }
    }
}
```

追加(最後)

- 構文

```
List.Add(値); //値はobject型（文字型、数値型等）で混在不可
（dynamic型は除く）
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //空 → "A" → "A","B"
        List<string> _list = new List<string>();
        _list.Add("A");
        _list.Add("B");
        foreach (object value in _list) {
            Console.WriteLine(value); //"A"→"B"
        }
    }
}
```

追加(指定位置)

- 構文

```
List.Insert(インデックス番号,値); //先頭（0）～最後
（List.Capacity-1）まで指定可能
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //”A”,”B” → ”C”,”A”,”B”
        List<string> _list = new List<string>() { ”A”, ”B” };
        _list.Insert(0,”C”); //先頭に追加する場合は0
        foreach (object value in _list) {
            Console.WriteLine(value); //”C”→”A”→”B”
        }
    }
}
```

更新(任意の値)

- 構文

```
List[インデックス番号] = 値;
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
```

```
    static void Main() {
        //”A”,”B” → ”C”,”B”
        List<string> _list = new List<string>() { ”A”, ”B” };
        _list[0] = ”C”; //0番目を変更する場合
        foreach (object value in _list) {
            Console.WriteLine(value); //”C”→”B”
        }
    }
}
```

###更新(null型)

- 構文

```
List[インデックス番号] = null;
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //”A”,”B”,”C” → ”A”,”B”,null
        List<string> _list = new List<string>() { ”A”, ”B”, ”C”
};
        _list[2] = null;
        foreach (object value in _list) {
            Console.WriteLine(value); // ”A”→”B”→ (null)
        }
    }
}
```

削除(指定のオブジェクト)

- 構文

```
List.Remove(object); //最初に見つかった指定のオブジェクトを削除
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //"A","B","C" → "A","C"
        List<string> _list = new List<string>() { "A", "B", "C"
};
        _list.Remove("B");
        foreach (object value in _list) {
            Console.WriteLine(value); // "A"→"C"
        }
    }
}
```

削除(指定のインデックス)

- 構文

```
List.RemoveAt(インデックス番号); //先頭 (0) ～最後
 (List.Capacity-1) まで指定可能
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //”A”,”B”,”C” → ”B”,”C”
        List<string> _list = new List<string>() { ”A”, ”B”, ”C”
};
        _list.RemoveAt(0); //先頭を削除する場合
        //_list.RemoveAt(_list.Capacity-1); //最後を削除する場合
        foreach (object value in _list) {
            Console.WriteLine(value); // ”B”→”C”
        }
    }
}
```

削除(〇番目から□個)

- 構文

```
List.RemoveRange(開始, 削除する個数); //開始＝削除開始したいインデックス番号
List.RemoveRange(開始, List.Capacity-開始); //〇番目から最後まで削除
List.Clear(); //全て削除
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //”A”,”B”,”C”,”D” → ”A”,”B”
```

```
        List<string> _list = new List<string>() { "A", "B", "C",
"D"};
        _list.RemoveRange(2, 2); //2番目から2個削除
        //]_list.RemoveRange(1, _list.Capacity-1); //1番目～最後
を削除する場合
        //_list.Clear(); //全て削除する場合
        foreach (object value in _list) {
            Console.WriteLine(value); // "A"→"B"
        }
    }
}
```

抽出(〇番目から□個)

- 構文

```
List.GetRange(開始, 抜き出す個数); //開始＝抜出しを開始したいイ
ンデックス番号
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //"A","B","C","D" → "C","D"を返す
        List<string> _list = new List<string>() { "A", "B", "C",
"D"};
        List<string> _result = _list.GetRange(2, 2); //2番目から
2個抽出する場合
        //List<string> _result = _list.GetRange(1,
_list.Capacity-1);//1番目～最後を抽出
        foreach (object value in _result) {
```

```
            Console.WriteLine(value); // ”C”→”D”
        }
    }
}
```

検索(前から)

- 構文

```
List.IndexOf(object [,検索開始するインデックス番号]);
//最初に見つかったインデックス番号を返す（無い場合-1）
//第2引数を省略すると最初（0）から検索
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { ”A”, ”B”, ”C”,
”D”};
        Console.WriteLine(_list.IndexOf(”C”,0)); //2
        //最初から検索する場合（第2引数が0の場合は省略可能）
    }
}
```

検索(後ろから)

- 構文

```
List.LastIndexOf(object [,検索開始するインデックス番号]);
```

```
//最後に見つかったインデックス番号を返す（無い場合-1）
//第2引数を省略すると最後（List.Capacity-1）から検索
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { ”A”, ”B”, ”C”,
”D”};
        Console.WriteLine(_list.LastIndexOf(”C”)); //2
        //最初から検索する場合（第2引数が0の場合は省略可能）
    }
}
```

要素の数

- 構文

```
List.Count; //実際に格納されている要素の数
List.Capacity; //格納可能な要素の数
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //List<string> _list = new List<string>() { ”A”, ”B”,
```

```
”C”};
        List<string> _list = new List<string>(3); //空の
ArrayListを作成
        Console.WriteLine(_list.Count); //0 ←実際に格納されてい
る要素の数
        Console.WriteLine(_list.Capacity); //3 ←格納可能な要素
の数
    }
}
```

並べ替え(反転)

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { ”A”, ”B”, ”C”,
”D”};
        _list.Reverse();
        foreach (object value in _list) {
            Console.WriteLine(value); // ”D”→”C”→”B”→”A”
        }
    }
}
```

並べ替え(ソート)

- 構文

```
List.Sort(); //引数で範囲や比較方法を指定することも可能
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        //"C", "02", "A", "01", "03", "B" → "01", "02", "03",
"A", "B", "C"
        List<string> _list = new List<string>() { "C", "02",
"A", "01", "03", "B" };
        _list.Sort();
        foreach (object value in _list) {
            Console.WriteLine(value); //
"01"→"02"→"03"→"A"→"B"→"C"
        }
    }
}
```

結合

- 構文

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list1 = new List<string>() { "A", "B", "C"
};
        List<string> _list2 = new List<string>() { "D", "E", "F"
};

        //_list1の末尾に_list2を結合
        _list1.AddRange(_list2);

        foreach (object value in _list1) {
```

```
            Console.WriteLine(value); //
"A"→"B"→"C"→"D"→"E"→"F"
        }
    }
}
```

複製

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { "A", "B", "C"
};
        List<string> _listCopy = new List<string>(_list); //簡易
型コピー方法
        _list[0] = "X";
        Console.WriteLine(_list[0]); //"X"
        Console.WriteLine(_listCopy[0]); //"A (参照コピーではな
い)
    }
}
```

文字列→ List

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        string _string = "A,B,C,D"; //①元となる文字列
        string[] _array = _string.Split(','); //②文字列→配列に
変換 (「配列」参照)
```

```
        List<string> _list = new List<string>(); //③空のListを
作成
        foreach (string _tmp in _array) { //データ型に注意
            _list.Add(_tmp); //④配列の要素を1つずつListに追加
        }

        //確認
        foreach (object value in _array) {
            Console.WriteLine(value); // ”A”→”B”→”C”→”D”
        }
    }
}
```

全要素を取り出す

1. foreach 文を使う方法

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { ”A”,
”B”, ”C” };

        //全要素を取り出す
        foreach (object value in _list) {
            Console.WriteLine(value); // ”A”→”B”→”C”
        }
    }
}
```

2. for 文を使う方法

```
//test.cs
using System;
using System.Collections.Generic; //Listに必要
class Test {
    static void Main() {
        List<string> _list = new List<string>() { "A",
"B", "C" };

        //全要素を取り出す
        for (int i=0; i < _list.Count; i++) {
            Console.WriteLine(_list[i]); // "A"→"B"→"C"
        }
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月14日
更新日：2017年04月19日

連想配列（Dictionary）

概要

- ディクショナリ、ハッシュとも呼ばれる「キー」と「値」の組み合わせを格納するデータ構造
- 匿名型クラスは、同様のデータ構造を持てるが読取り専用

作成方法

```
Dictionary<キーの型, 値の型> 変数名 = new Dictionary<キーの型,
値の型>();
Dictionary<キーの型, 値の型> 変数名 = new Dictionary<キーの型,
```

```
値の型>() {
    {”キー①”, 値①},
    {”キー②”, 値②},
    ……
};
```

- 例文

```
//test.cs
using System;
using System.Collections.Generic; //Dictionaryに必要
 (Unity版でも必要)
class Test {
    static void Main() {
        //①作成（空のDictionaryを作成する場合、{}は不要）
        Dictionary<string, string> _dic = new
Dictionary<string, string>() {
            {”A”, ”あ”},
            {”I”, ”い”}
        };

        //②追加
        _dic.Add(”U”, ”う”);
        _dic.Add(”E”, ”え”);

        //③更新
        _dic[”A”] = ”ア”; //上書き変更

        //④取得
        Console.WriteLine(_dic[”A”]); //”ア”
    }
}
```

キー、値の検索

```
```

```
//test.cs
using System;
using System.Collections.Generic; //Dictionaryに必要（Unity版でも必要）
class Test {
    static void Main() {
        //①作成（空のDictionaryを作成する場合、{}は不要）
        Dictionary<string, string> _dic = new Dictionary<string, string>() {
            {"A", "あ"},{"I", "い"},{"U", "う"},{"E", "え"},{"O", "お"}
        };

        Console.WriteLine(_dic.ContainsKey("B")); //任意のキーがあるか否か（True／False）
        Console.WriteLine(_dic.ContainsValue("え")); //任意の値があるか否か（True／False）
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年12月11日
更新日：2017年04月19日

this

thisが必要な場合

1. 「引数」と「インスタンス変数」が同じ場合
2. 「ローカル変数」と「インスタンス変数」が同じ場合

- this は、this を記述したメソッドを所有するクラス（オブジェクト）を指す

例文

```
//test.cs
using System;

//メインクラス
class Test {
    static void Main() {
        Robot _robot = new Robot(500);
        _robot.Move();
        Console.WriteLine(_robot.X); //510
        //Console.WriteLine(this); //error（staticメソッド内では
参照できず）
    }
}

//カスタムクラス
class Robot {
    private int _x; //インスタンス変数（thisは不要）

    public Robot(int _x) { //引数
        this._x = _x; //①thisが無いとWarning（引数を参照してし
まう）
        Console.WriteLine(this); //Robot（このメソッドが実行され
たオブジェクト）
    }

    public void Move() {
        int _x; //ローカル変数
        _x = this._x + 10; //②thisが無いとerror（ローカル変数を
参照してしまう）
        if (_x >= 1920) _x = 0;
        this._x = _x; //②thisが無いとWarning（ローカル変数を参
照してしまう）
        Console.WriteLine(this); //Robot（このメソッドが実行され
たオブジェクト）
    }

    public int X {
        get { return _x; } //thisを付けてもよい（通常は省略）
```

```
        private set {}
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月15日
更新日:2017年04月19日

文字列の操作

string オブジェクトの作成

- 構文

```
string 変数名 = ”〇〇”; //文字列リテラルを使う方法
string 変数名 = new string(new char[]{’〇’,’〇’,...}); //new演算子とchar型配列を使う方法
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        //①文字列リテラルを使う
        string _string1 = ”ABCDE”;
        Console.WriteLine(_string1); //”ABCDE”

        //②new演算子とchar型配列を使う
        string _string2 = new string(new char[]
{’A’,’B’,’C’,’D’,’E’});
        Console.WriteLine(_string2); //”ABCDE”
```

```
    }
}
```

長さを調べる

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = ”ABCDE”;
        Console.WriteLine(_string.Length); //5
    }
}
```

一部分を取得

- 構文

```
String[番号] ←0（最初）～String.Length-1（最後）
String.Substring(開始 [,文字数])
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = ”0123456789”;
        Console.WriteLine(_string[4]); //”4”
        Console.WriteLine(_string.Substring(4)); //”456789”
        Console.WriteLine(_string.Substring(4,3)); //”456”
    }
```

```
}
```

一部分を削除

- 構文

```
String.Remove(開始位置, 削除する文字数);
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = ”にしむらたかし”;
        Console.WriteLine(_string.Remove(0, 4)); //”たかし”
    }
}
```

置換

- 構文

```
String.Replace(”置換前の文字列”, ”置換後の文字列”);
String.Replace('置換前の文字', '置換後の文字');
```

- 例文

```
//test.cs
using System;
```

```
class Test {
    static void Main() {
        string _string = ”2017年04月19日”;
        Console.WriteLine(_string.Replace(”2017年”, ”平成29
年”)); //”平成29年04月19日”
    }
}
```

検索

- 構文

```
String.IndexOf(”検索したい文字列”, 開始位置);
String.IndexOf('検索したい文字', 開始位置);
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = ”ABCDEFG-ABCDEFG”;
        string _word = ”CD”;
        int _i = 0;
        while (_string.IndexOf(_word, _i) != -1) { //見つからな
い場合「-1」
            int _num = _string.IndexOf(_word, _i);
            Console.WriteLine(_num); //2、10 ←”CD”が見つかった
位置を出力
            Console.WriteLine(_string.Substring(_num,
_word.Length)); //”CD”、”CD”
            _i = _num + 1;
        }
    }
```

```
}
```

文字列→配列

- 構文

```
String.Split('区切り文字');
```

- 例文

```
//test.cs
using System;
class Test {
    static void Main() {
        string _string = "A,B,C,D"; //「,」区切りの文字列
        string[] _array = _string.Split(','); //「,」区切りで分
割して配列化
        foreach (object value in _array) {
            Console.WriteLine(value); // "A"→"B"→"C"→"D"
        }
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月17日
更新日:2017年04月19日

正規表現

- C# には以下のサンプル以外にも多くの正規表現の機能が用意されています

マッチした数

```
//test.cs
using System;
using System.Text.RegularExpressions; //Regexに必要
class Test {
    static void Main() {
        string _string = ”cabacbbacbcba”;
        //”a”がいくつ含まれるか
        MatchCollection _mc = Regex.Matches(_string, ”a”);
        Console.WriteLine(_mc.Count); //4
    }
}
```

パスワード

```
//test.cs
using System;
using System.Text.RegularExpressions; //Regexに必要
class Test {
    static void Main() {
        string _string = @”U7eLoERa”; //任意のパスワード（@を付
ける）
        /* 条件
        8文字以上（全て半角）
        1文字以上の「数字」を含む
        1文字以上の大文字および小文字の「英字」を含む
        */
        Regex _regex = new Regex(@”^(?=.*¥d)(?=.*[a-z])(?=.*[A-
Z])¥w{8,}$”);//②
        Match _match = _regex.Match(_string);
        Console.WriteLine(_match.Success); //True ←パスワードと
して条件に合致
    }
}
```

郵便番号(7桁)

```
//test.cs
using System;
using System.Text.RegularExpressions; //Regexに必要
class Test {
    static void Main() {
        string _string = "156-0057"; //任意の郵便番号
        Regex _regex = new Regex("¥¥d{3}-¥¥d{4}");
        Match _match = _regex.Match(_string);
        Console.WriteLine(_match.Success); //True ←郵便番号とし
て条件に合致
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月19日
更新日:2017年04月19日

インターフェース

概要

- クラスにどのような機能(メソッド)を持たせるか、ということだけを定める
- 抽象クラスと似ているが、抽象クラスとは異なり、実際の処理は一切記述できない
- 実際の処理はインターフェースを継承したクラスで定義(実装しないとエラー)
- 多重実装(複数のインターフェースを同時に指定)や多重継承も可能

構文

```
//インターフェースの宣言
interface Iインターフェース名 { //慣例的にインターフェース名の先頭にIを付けます
    戻り値の型 メソッド名A([型① 引数①, 型② 引数②,...]); //暗黙的にpublic扱い
    ......
}
//インターフェースの実装
class クラス名 : Iインターフェース名 { ......
```

例文

```
//test.cs
using System;
class Test {
    static void Main() {
        Moneybox _moneybox = new Moneybox();
        _moneybox.Add(5000);
        Console.WriteLine(_moneybox.Total); //5000
    }
}

//インターフェースの宣言
interface IMoneybox {
    void Add(int _money); //通常のメソッド（暗黙的にpublicになる）
    int Total { get; set; } //get/setアクセサ（暗黙的にpublicになる）
}

//インターフェースの実装（継承との併用は,を使う）
class Moneybox : IMoneybox {
    private int _total = 0;
    public void Add(int _money) { _total += _money; }
    public int Total {
        get { return _total; }
```

```
        set { _total = value; }
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月21日
更新日：2017年04月19日

抽象クラス（abstract）

概要

- 派生クラスに"実装しなければならないメソッド"を抽象クラスで定義する
- 実際の処理は、抽象クラスを継承した派生クラスで、抽象メソッドを override して記述

構文

```
abstract class Abstract〇〇 { //抽象クラスの定義
    public abstract 戻り値の型 メソッド名A([型① 引数①, 型②
引数②,...]);
}
class SubClass : Abstract〇〇 { //抽象クラスの継承
    public override 戻り値の型 メソッド名A([型① 引数①, 型②
引数②,...]) {
        //実際の処理
    }
    ……
```

例文

```
//test.cs
using System;
class Test {
    static void Main() {
        SubClass _subClass = new SubClass();
        _subClass.Common(); //”AbstractClass.Common()”
        _subClass.Method(); //”SubClass.Method()”
    }
}
abstract class AbstractClass { //「抽象クラス」の定義
    public void Common() { //共通のメソッド
        Console.WriteLine(”AbstractClass.Common()”);
    }
    public abstract void Method(); //「抽象メソッド」の宣言（実
際の処理は書かない）
}
class SubClass : AbstractClass { //抽象クラスを継承
    public override void Method() { //オーバーライドして実際の処
理を記述
        Console.WriteLine(”SubClass.Method()”); //実際の処理
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月24日
更新日：2017年04月21日

base キーワード

概要

基本クラスに定義されたコンストラクタ（private 以外）は、派生クラスのコンストラクタが実行される直前に必ず実行されるその際、基本クラスのコンストラクタへ、派生クラ

スのコンストラクタから引数を渡すことがbaseを使うことで可能になる（≒ super）
base.メソッド() で基本クラスのメソッドを呼び出す事が可能（「オーバーライド」参照）

書式

```
class 基本クラス名 {
    public 基本クラス名(型 引数) { //基本クラスのコンストラクタ
        //引数を使った処理etc.
    }
}
class 派生クラス名 : 基本クラス名 { //派生クラス（基本クラスを継承）
    public 派生クラス名() : base(引数) { //派生クラスのコンストラクタ
    }
}
```

例文

```
//test.cs
using System;
class Test {
    static void Main() {
        SubClass _subClass = new SubClass(”A”);
    }
}
class SuperClass { //基本クラス
    public SuperClass(string p1, string p2) { //基本クラスのコンストラクタ
        Console.WriteLine(”SuperClass:” + p1 + ”:” + p2); //①番目（p1は”A”、p2は”B”）
    }
}
class SubClass : SuperClass { //派生クラス
    public SubClass(string p) : base(p, ”B”) { //派生クラスのコ
```

```
ンストラクタ
        Console.WriteLine("SubClass:" + p); //②番目 (pは"A")
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月24日
更新日:2017年04月21日

オーバーライド

概要

- 基本クラス(または抽象クラス)で定義したメソッドを、派生クラスで再定義することをオーバーライドと呼ぶ
- オーバーライドできるメソッドは、基本クラスの場合 virtual 、抽象クラスの場合 abstract キーワードが付加されたものに限る
- 基本クラスのメソッドを、オーバーライドによって拡張する場合などで、基本クラスのメソッドを呼び出したい場合は、base.メソッド名() を使用する(「base キーワード」参照)

「仮想メソッド」のオーバーライド

- 書式

```
class 基本クラス名 {
    アクセス修飾子 virtual 戻り値の型 メソッド名([型 引数]) {
        ……
    }
    ……
}
class 派生クラス名 : 基本クラス { //派生クラス (基本クラスを継承)
```

```
    アクセス修飾子 override 戻り値の型 メソッド名([型 引数]) {
        base.メソッド名(引数); //基本クラスのメソッドを呼び出す
（オプション）
        ……
    }
    ……
}
```

- 例文

```
//test.cs
using System;

class Test {
    static void Main() {
        SubClass _subClass = new SubClass();
        _subClass.Method();
    }
}

class SuperClass { //基本クラス
    public virtual void Method() { //オーバーライドを許可
        Console.WriteLine(”SuperClass.Method”);
    }
}

class SubClass : SuperClass { //派生クラス（基本クラスを継承）
    public override void Method() { //基本クラスのメソッドのオー
バーライド
        Console.WriteLine(”SubClass.Method”);
        base.Method(); //”SuperClass.Method”←基本クラスのメソッ
ド実行（オプション）
    }
}
```

「抽象メソッド」のオーバーライド

- 書式

```
abstract class 抽象クラス名 { //抽象クラスの定義
    アクセス修飾子 abstract 型 抽象メソッド名([型 引数]); //抽象
メソッド宣言
    ……
}
class 派生クラス : 抽象クラス名 { //抽象クラスを継承
    アクセス修飾子 override 型 抽象メソッド名([型 引数]) { //オ
ーバーライド
        //実際の処理
    }
    ……
}
```

- 例文

```
//test.cs
using System;

class Test {
    static void Main() {
        SubClass _subClass = new SubClass();
        _subClass.Method();
    }
}

abstract class AbstractClass { //抽象クラス
    public abstract void Method(); //抽象メソッドの宣言
}

class SubClass : AbstractClass { //派生クラス（抽象クラスを継
承）
    public override void Method() { //オーバーライドして実際の処
```

```
理を記述
        Console.WriteLine("AbstractClass.Method");
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura
作成日：2015年11月24日
更新日：2017年04月21日

カスタムイベント

概要

イベントとは、あるアクションが発生したことを自動的に通知する仕組みカスタムクラス内で何か処理をし終えた際、別のオブジェクトにそのことを知らせる場合に、このイベント機能を使用イベントを設定したカスタムクラスからは、情報（イベント）を発信するだけ情報を受けたいオブジェクトは、リスナーメソッドを準備して待ち受ける…このことにより、カスタムクラスを汚さずに済む、というメリットが生まれるC# に用意された event は、特殊なデリゲート（delegate）と言えるものですデリゲートとの違いは、event 宣言した変数（イベント名）には、イベントハンドラ（≒リスナー関数）の追加（+=）または削除（-=）のみ可能ということ等

書式

- イベントの設定

```
class クラス名 {
    public delegate void デリゲート名([型 引数]); //デリゲート宣
言
    public event デリゲート名 イベント名; //これにイベントハンド
ラを登録
    public 戻り値の型 メソッド名([型 引数]) { //イベントを発生さ
せたいメソッド
```

```
        ……
        イベント名([引数]); //ここでイベントハンドラを呼出す!
    }
    ……
}
```

- イベントハンドラの登録

```
クラス名 〇 = new クラス名();
〇.イベント名 += イベントハンドラ名; //イベントハンドラを削除する場合「-=」
……
static 戻り値の型 イベントハンドラ名([型 引数]) {
    //イベントが発生した際に処理すること
}
```

例文

```
//test.cs
using System;

class Test {
    static void Main() {
        MyGame _myGame = new MyGame();
        _myGame.GameOverEvent += GameOverHandler_myGame; //複数登録可能（+=、-=のみ）
        //_myGame.GameOverEvent -= GameOverHandler_myGame; //イベントハンドラの削除
        for (int i=0; i<10; i++) { //10回繰返す場合…
            Console.WriteLine("得点:" + _myGame.Point);
            _myGame.AddPoint();
        }
    }
```

```
    static void GameOverHandler_myGame(object arg) { //イベント
ハンドラ
        Console.WriteLine("ゲームオーバー！ " + arg); //"ゲームオ
ーバー! MyGame"
    }
}

//イベントを設定するクラス
class MyGame {
    private int _point = 0;
    public delegate void MyEventHandler(object arg); //デリゲー
ト宣言
    public event MyEventHandler GameOverEvent; //これにイベント
ハンドラを登録
    public void AddPoint() { //イベントを発生させたいメソッド
        if (++_point >= 10) {
            if (GameOverEvent != null) {
                GameOverEvent(this); //イベントハンドラの呼出し
            }
        }
    }
    public int Point {
        get { return _point; }
        private set {} //読取専用
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月25日
更新日:2017年04月21日

数学関数（Math）

Math.Sin()：サイン（正弦）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Sin(0)); //0 ←0°
        Console.WriteLine(Math.Sin(Math.PI/2)); //1 ←90°
        Console.WriteLine(Math.Sin(Math.PI));
//1.22460635382238E-16 (≒0) ←180°
        Console.WriteLine(Math.Sin(Math.PI*3/2)); //-1 ←270°
        Console.WriteLine(Math.Sin(Math.PI*2));
//-2.44921270764475E-16 (≒0) ←360°
    }
}
```

Math.Cos() : コサイン（余弦）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Cos(0)); //1 ←0°
        Console.WriteLine(Math.Cos(Math.PI/2));
//6.12303176911189E-17 (≒0) ←90°
        Console.WriteLine(Math.Cos(Math.PI)); //-1 ←180°
        Console.WriteLine(Math.Cos(Math.PI*3/2));
//-1.83690953073357E-16 (≒0) ←270°
        Console.WriteLine(Math.Cos(Math.PI*2)); //1 ←360°
    }
}
```

Math.Atan2() : アークタンジェント2

```
//test.cs
```

```
using System;
class Test {
    static void Main() {
        //2つの値のアークタンジェント（逆タンジェント）X、Y座標
の角度をラジアン単位で返す
        //πラジアン（3.141592…）は180°
        //三角形の各辺が1:2:√3の場合、2:√3の間の角度は30°であ
ることを検証
        double _disX = Math.Sqrt(3); //√3のこと
        int _disY = 1;
        Console.WriteLine(Math.Atan2(_disY, _disX));
//0.523598775598299（ラジアン）
        Console.WriteLine(180*Math.Atan2(_disY, _disX)/Math.PI);
//30（度）
    }
}
```

Math.PI：円周率

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.PI); //3.14159265358979（Math.PI
ラジアン=180°）
    }
}
```

Math.Floor()：切り捨て

```
//test.cs
using System;
class Test {
    static void Main() {
```

```
        Console.WriteLine(Math.Floor(1.001)); //1
        Console.WriteLine(Math.Floor(1.999)); //1
    }
}
```

Math.Ceiling() : 切り上げ

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Ceiling(1.001)); //2
        Console.WriteLine(Math.Ceiling(1.999)); //2
    }
}
```

Math.Round() : 四捨五入

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Round(1.499)); //1
        Console.WriteLine(Math.Round(1.500)); //2
    }
}
```

Math.Abs() : 絶対値

```
//test.cs
using System;
class Test {
```

```
    static void Main() {
        Console.WriteLine(Math.Abs(100)); //100
        Console.WriteLine(Math.Abs(-100)); //100
    }
}
```

Math.Pow() : 累乗（〇の□乗）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Pow(2, 0)); //1 (2の0乗)
        Console.WriteLine(Math.Pow(2, 8)); //256 (2の8乗)
    }
}
```

Math.Sqrt() : 平方根（√〇）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Sqrt(2)); //1.4142135623731 (一夜一夜にひとみごろ)
        Console.WriteLine(Math.Sqrt(3)); //1.73205080756888 (人並みに奢れや)
        Console.WriteLine(Math.Sqrt(4)); //2
        Console.WriteLine(Math.Sqrt(5)); //2.23606797749979 (富士山麓オウム鳴く)
    }
}
```

Math.Max() : 比較（最大値）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Max(5.01, -10)); //5.01 ←「2つ」
の数値の比較
    }
}
```

Math.Min() : 比較（最小値）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Min(5.01, -10)); //-10 ←「2つ」
の数値の比較
    }
}
```

Math.Sign() : 符号（正か負の値か）

```
//test.cs
using System;
class Test {
    static void Main() {
        Console.WriteLine(Math.Sign(-0.1)); //-1 (負の値)
        Console.WriteLine(Math.Sign(0)); //0 (0)
        Console.WriteLine(Math.Sign(0.1)); //1 (正の値)
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月26日
更新日:2017年04月21日

乱数

- システム時間を元に発生させているためfor文で同時に異なる乱数を発生できない

書式

```
Random ○ = new Random([seed値]);
//↑引数（シード値）を省略するとEnvironment.TickCount（システム時間）を利用
○.NextDouble(); //0～1.0までの浮動小数点数の乱数
○.Next(); //整数値の乱数（百万～数十億等）
○.Next(整数値); //0～整数値の値の乱数（整数）
```

例文

```
//test.cs
using System;
class Test {
    static void Main() {
        Random _random = new Random();

        //0～1.0までの乱数
        Console.WriteLine(_random.NextDouble());
//0.0432652673350072
        Console.WriteLine(_random.NextDouble());
```

```
//0.78664848541429
        Console.WriteLine(_random.NextDouble());
//0.545385330900118

        //整数値の乱数
        Console.WriteLine(_random.Next()); //369339869
        Console.WriteLine(_random.Next()); //1966699381
        Console.WriteLine(_random.Next()); //6900123

        //任意の値までの整数値の乱数
        Console.WriteLine(_random.Next(10)); //0
        Console.WriteLine(_random.Next(10)); //3
        Console.WriteLine(_random.Next(10)); //9
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月26日
更新日:2017年04月21日

日時情報

書式

```
DateTime ○ = DateTime.Now; //DateTimeは構造体
○.Year; //年（2017等）
○.Month; //月（1～12）
○.Day; //日（1～31）
○.DayOfYear; //元日から日数（1～366）
○.DayOfWeek; //曜日（Sanday～Saturday）
○.Hour; //時間（0～23）
○.Minute; //分（0～59）
○.Second; //秒（0～59）
○.Millisecond; //ミリ秒（0～999）
```

```
〇.Ticks; //0001年1月1日00:00:00からの経過時間（ナノ秒）：但し精度は10ミリ秒
```

例文

```
//test.cs
using System;
class Test {
    static void Main() {
        DateTime _now = DateTime.Now;
        Console.WriteLine(_now); //4/21/2017 10:16:04 AM
        Console.WriteLine(_now.Year); //2017
        Console.WriteLine(_now.Month); //4
        Console.WriteLine(_now.Day); //21
        Console.WriteLine(_now.DayOfYear); //111（元日からの日数）
        Console.WriteLine(_now.DayOfWeek); //Friday
        Console.WriteLine(_now.Hour); //10
        Console.WriteLine(_now.Minute); //16
        Console.WriteLine(_now.Second); //4
        Console.WriteLine(_now.Millisecond); //337
        Console.WriteLine(_now.Ticks); //636283665643372990（100ナノ秒単位）
        //”hh:mm:ss”で現在の時間を表示する方法
        string _h = (_now.Hour < 10) ? ”0” + _now.Hour : _now.Hour.ToString();
        string _m = (_now.Minute < 10) ? ”0” + _now.Minute : _now.Minute.ToString();
        string _s = (_now.Second < 10) ? ”0” + _now.Second : _now.Second.ToString();
        Console.WriteLine(_h + ”:” + _m + ”:” + _s); //”10:16:04”
    }
}
```

実行環境:Ubuntu 16.04.2 LTS、C++14
作成者:Takashi Nishimura
作成日:2015年11月27日
更新日:2017年04月21日

タイマー

スレッドタイマー(System.Threading.Timer)を使う方法

```
//test.cs
/*
システムタイマー（後述）と比較すると軽量
Windows Formでの使用は非推奨
*/
using System;
using System.Threading; //System.Threading.Timerに必要

class Test {
    private static Timer _timer; //privateは省略可

    static void Main() {
        _timer = new Timer(new TimerCallback(Loop)); //タイマー
の生成
        _timer.Change(0, 1000); //0ミリ秒後から、1000ミリ秒間隔
で開始!
        Console.ReadLine(); //ここでは必須（要注意）
    }

    static void Loop(object arg) { //1000ミリ秒毎に実行される
        Console.WriteLine(arg); //System.Threading.Timer
        //_timer.Change(Timeout.Infinite, Timeout.Infinite); //
停止 ←力技
    }
}
```

システムタイマー（System.Timers.Timer）を使う方法

```
//test.cs
/*
サーバベース・タイマーとも呼ばれる
スレッドタイマー（前述）と比較すると重いが精度が高い
スレッドの経過時間とは独立した時間監視をする
Windows Formでの使用もＯＫ
*/
using System;
using System.Timers; //System.Timers.Timerに必要

class Test {
    private static Timer _timer; //privateは省略可

    static void Main() {
        _timer = new Timer(); //タイマーの生成
        _timer.Interval = 1000; //1000ミリ秒間隔
        _timer.Elapsed += new ElapsedEventHandler(Loop); //イベ
ントハンドラの追加
        _timer.Start(); //開始!
        Console.ReadLine(); //ここでは必須（要注意）
    }

    static void Loop(object arg1, EventArgs arg2) { //1000ミリ秒
毎に実行される
        Console.WriteLine(arg1); //System.Timers.Timer（タイマー
本体）
        Console.WriteLine(arg2);
//System.Timers.ElapsedEventArgs（各種情報）
        //_timer.Stop(); //停止 ←この場合１回で停止
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成者：Takashi Nishimura

作成日:2015年11月27日
更新日:2017年04月21日

処理速度計測

DateTime構造体を使う方法

```
//test.cs
//日時情報を得るためのDatetime構造体を利用して計測する方法
using System; //DateTimeに必要
class Test {
    static void Main() {
        long _start = DateTime.Now.Ticks; //100ナノ秒単位（精度
は10ミリ秒）
        for (long i=0; i<10000000000; i++) { //100億回繰り返す場
合…
            //速度計測したい処理
        }
        Console.WriteLine(DateTime.Now.Ticks - _start);
//33060210 (≒3.3秒)
    }
}
```

Stopwatchクラスを使う方法

```
//test.cs
/*
.NET Framework 2.0から追加された機能
Stopwatchクラスのインスタンスを生成しStart/Stopメソッドを実行す
るだけで可能
*/
using System;
using System.Diagnostics; //Stopwatchに必要
class Program     {
```

```
    static void Main() {
        Stopwatch _stopWatch = new Stopwatch(); //インスタンスの
生成
        _stopWatch.Start(); //計測開始
        for (long i=0; i<10000000000; i++) { //100億回繰り返す場
合…
            //速度計測したい処理
        }
        _stopWatch.Stop(); //計測終了
        Console.WriteLine(_stopWatch.ElapsedMilliseconds);
//3230（ミリ秒）
        Console.WriteLine(_stopWatch.Elapsed);
//00:00:03.2302265（秒）
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14
作成日：2015年11月28日
更新日：2017年04月21日

外部テキストの読み込み

- Web サーバを稼働する必要はない

テキストファイルの用意（sample.txt/UTF-8として保存）

```
あいうえお
かきくけこ
さしすせそ
```

例文（StreamReader クラスを使う方法）

```
//test.cs
using System;
using System.IO; //StreamReaderに必要
class Test {
    static void Main() {
        string _path = ”sample.txt”;
        //↓Shift-JISなどUTF-8以外の場合、第2引数で指定します
        StreamReader _stream = new StreamReader(_path); //.txt以
外も可能
        string _string = _stream.ReadToEnd(); //全ての内容を読み
込む
        _stream.Close(); //閉じる
        Console.WriteLine(_string); //結果を出力
    }
}
```

例文（File.OpenTextメソッドを使う方法）

```
//test.cs
using System;
using System.IO; //StreamReaderに必要
class Test {
    static void Main() {
        string _path = ”sample.txt”;
        StreamReader _stream = File.OpenText(_path); //.txt以外
も可能（UFT-8限定）
        string _string = _stream.ReadToEnd(); //全ての内容を読み
込む
        _stream.Close(); //閉じる
        Console.WriteLine(_string); //結果を出力
    }
}
```

実行環境：Ubuntu 16.04.2 LTS、C++14

作成者:Takashi Nishimura
作成日:2015年11月30日
更新日:2017年04月21日