講義メモ

・p.58「リテラル」から

提出フォロー：アレンジ演習：**p.57 bool01.cs**

・bool型の変数aに整数値を代入するとどうなるか確認し、コメントアウトしよう
・bool型の変数bに文字列”true”を代入するとどうなるか確認し、コメントアウトしよう
・int型の変数c、double型の変数d、float型の変数e、decimal型の変数f、uint型の変数gを定義し、それぞれの型情報をGetType()で表示する処理を追加しよう
⇒ cはSystem.Int32, dはSystem.Double, eはSystem.Single, fはSystem.Decimal, gはSystem.UInt32 となる

作成例

```
//アレンジ演習：p.57 bool01.cs
using System;
class bool01
{
    public static void Main()
    {
        bool a = true; //論理型の変数aをtrueで初期化
        bool b = false; //論理型の変数aをfalseで初期化
        Console.WriteLine("a = {0}, b = {1}", a, b); //True、False
        Console.WriteLine("aは{0}", a.GetType()); //System.Boolean
        Console.WriteLine("aは文字列にすると「{0}」", //True
            a.ToString());
        Console.WriteLine("bは文字列にすると「{0}」", //False
            b.ToString());
        //【以下追加】
        // a = 0; //「変換できません」エラーになる
        // b = "true"; //同上
        int c = 0;
        double d = 0.0;
        float e = 0.0f;
        decimal f = 0.0M;
        uint g = 0U;
        Console.WriteLine("cは{0},dは{1},eは{2},fは{3},gは{4}",
            c.GetType(), d.GetType(), e.GetType(), f.GetType(), g.GetType());
    }
}
```

p.58 リテラル：整数リテラル

・プログラムソースの中に記述した値のことをリテラルという
・リテラルの表現方法は型によって異なるので注意。
・整数リテラル：数字列で、先頭のみ「-」も指定可能。int型扱いになるので、表現できる範囲はint型の範囲になる
・uint型整数リテラル：整数リテラルの末尾にUまたはuをつけたもの。uint型型扱いになり、マイナス値ではないことを明示でき、正の数の表現できる範囲はint型の倍になる
・long型整数リテラル：整数リテラルの末尾にLまたはl(非推奨)をつけたもの。long型扱いになり、int型の範囲を超える値を表現できる
・ulong型整数リテラル：整数リテラルの末尾にULまたはulをつけたもの。ulong型扱いになり、マイナス値ではないことを明示でき、正の数の表現できる範囲はlong型の倍になる

・末尾に与える文字をサフィックスという。なお、byte型、sbyte型、short型、ushort型のサフィックスはない。

p.58 リテラル：実数リテラル

・実数リテラル：整数リテラルに1つだけ「.」を加えたもの（先頭でもOK）。double型扱いになるので、表現できる範囲はdouble型の範囲になる
・float型実数リテラル：実数リテラルの末尾にFまたはfをつけたもの。float型扱いになり、メモリの利用量を削減できるが、精度が下がり、表現できる範囲はflort型の範囲になる(※ Unityでよく使われる)
・decimal型実数リテラル：実数リテラルの末尾にMまたはmをつけたもの。decimal扱いになり、メモリの利用量が倍に増えるが、精度が最大まで向上する

p.58 リテラル：補足**-**その他のリテラル

・論理値リテラル：bool型のリテラルで、true、falseのみ
・文字リテラル：char型のリテラルで、1文字をシングルコーテーションで囲ったもの
・文字列リテラル：string型のリテラルで、0文字以上をダブルクォーテーションで囲ったもの

p.58 Object.GetTypeメソッド

・p.45にあるように、データ型はC#における構造体で定義されている
・そのため、内部的にObjectクラスが提供するメソッドが自動的に実行可能になっている
・その一つがGetTypeメソッドで「変数等やリテラル.GetType()」で、.NET型情報がType型で返される
・これをConsole.WriteLine/Console.Writeに与えると、文字列と表示できる

p.60 literal01.cs 補足

・「string format = "{0}の型は.NET型で{1}です";」でフォーマット文字列を変数に与えている
・これを以降のConsole.WriteLineで使いまわしている
・この用に、フォーマット文字列は事前にしておくことが可能
・なお、実数リテラルのサフィックスを整数リテラルの末尾につける事で、実数リテラルになる
・通常用いないdouble型のサフィックスDまたはdは、このような場合に用いる事ができる

アレンジ演習：**p.60 literal01.cs**

・「(-10).GetType()」を「-10.GetType()」とすると、どうなるか確認しよう
　⇒「.」演算子が「-」演算子より優先なので、先に「10.GetType()」が動作してしまい、結果の文字列に対して「-」演算子を動作させようとしてしまうので、文法エラーになる

作成例

```
//アレンジ演習：p.60 literal01.cs
using System;
class literal01
{
    public static void Main()
    {
        string format = "{0}の型は.NET型で{1}です"; //フォーマット文字列
        Console.WriteLine(format, "100", 100.GetType()); //int
        Console.WriteLine(format, "100U", 100U.GetType()); //uint
        Console.WriteLine(format, "100L", 100L.GetType()); //long
        Console.WriteLine(format, "100UL", 100UL.GetType()); //ulong
        Console.WriteLine(format, "1.25", 1.25.GetType()); //double
```

```
        Console.WriteLine(format, "1.25F", 1.25F.GetType()); //float
        Console.WriteLine(format, "1.25M", 1.25M.GetType()); //decimal
        Console.WriteLine();
        Console.WriteLine(format, "10F", 10F.GetType()); //floart
        Console.WriteLine(format, "10D", 10D.GetType()); //double
        Console.WriteLine(format, "10M", 10M.GetType()); //decimal
        Console.WriteLine();
        Console.WriteLine(format, "-10D", (-10D).GetType()); //double
        //Console.WriteLine(format, "-10D", -10D.GetType()); //【追加】エラー
    }
}
```

p.61 暗黙の型指定

・初期化に用いる初期値により、変数の型が確定できる場合、型を明示せずに「var」とすることができる
　例：var i = 1; //変数iはint型になる
・このvarはvarキーワードまたはvar型と呼ばれ、暗黙の型指定の機能を持つ
・初期値が必要なので「var x;」のような記述は不可だが、式やメソッドの呼び出し結果を初期値とする初期化では、利用可能
　例：var s = "SCORE:" + 100; //変数sはstring型になる

p.62 var01.cs 補足

・このソースでは、Mainメソッドの戻り値型をvoidではなくintにしているが、この場合は特に意味はない
・Mainメソッドの戻り値型をintにすると、正常終了した旨を「return 0;」で返すことができるが、この場合は特に意味はない
・よって、ここまでのサンプルソースと同様に「void Main()」にして、「return 0;」は割愛すると良い
・p.63 dynamic01.cs も同様

アレンジ演習：**p.62 var01.cs**

・「myText + no + myc」をvar型変数sの初期値にして、その型を表示する処理を追加しよう

作成例

```
//アレンジ演習：var01.cs
using System;
class var01
{
    public static void Main()
    {
        var mytext = "猫でもわかるC#プログラミング 第"; //string型
        var no = 3; //int型
        var myc = '版'; //char型
        Console.WriteLine(mytext + no + myc);
        Console.WriteLine("mytextの型は{0}で、noの型は{1}で、mycの型は{2}です",
            mytext.GetType(), no.GetType(), myc.GetType());
        //【以下追加】
        var s = mytext + no + myc; //連結されてstring型になる
        Console.WriteLine("sの型は{0}", s.GetType());
    }
}
```

p.62 dynamic型

・C#バージョン4.0以降において、型が未確定の変数を定義可能になった。
・この場合の型がdynamic型で、var型の上位概念に当たる。
・dynamic型の変数は代入によって型が決まるので、便利だが実行効率はvar型より低くなる懸念がある
・定義済のdynamic型変数に、異なる型の値を代入すると型を変更できる（varとは異なりエラーにならない）
・ただし、内部的に、Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo.Createを用いるので、プロジェクトに下記の参照指定が必要
　⇒ Microsoft.CSharp
・手順
　①ソリューションエクスプローラで「参照」を右クリックし「参照の追加」
　②「Microsoft.CSharp」のチェックをオンにして「OK」
・なお、var型と同様に、初期化によって型を決めることもできる

アレンジ演習：**p.62 dynamic01.cs**

・変数zに文字列を代入できること、代入後の型情報が変化することを確認する処理を追加しよう
・また、var型の変数wを実数で初期化し、変数wには文字列を代入できないことを確認する処理を追加しよう（確認後にコメントアウト）

作成例

```
//アレンジ演習：p.63 dynamic01.cs
using System;
class Dynamic01 {
  public static void Main() {
    dynamic x = 10, y = "abc", z;
    z = 1.25;
    Console.WriteLine("x ---- {0}", x.GetType());
    Console.WriteLine("y ---- {0}", y.GetType());
    Console.WriteLine("z ---- {0}", z.GetType());
    //【以下追加】
    z = "ABC"; //double型だったがstring型に変わり、文字列を代入できる
    Console.WriteLine("z ---- {0}", z.GetType()); //型が変わっている
    var w = 1.25; //double型になる
    //w = "ABC"; //エラー（double型なので文字列は代入できない）
  }
}
```

p.64 スコープ

・スコープは「視野（見える範囲）」⇒「有効範囲」の意味
・基本的に、変数などが定義されたブロックが、そのスコープになる
・ブロックが入れ子構造になっている場合（ブロックの中にさらにブロックがある場合）、内側のブロック内部もスコープになる
・なお、C/C++などでは、スコープが異なれば同名の変数を利用可能だが、C#では制限があり、後述するfor文などの場合を除いてエラーになる

p.64 scope01.cs 補足

・このソース内で行っているように、{} を用いることで、プログラマが自由にブロックを記述できる

・ブロックを記述することで、変数などのスコープを制限でき、ブロック外で誤って利用されることを防止できる
・ただし、必要性のないブロックの記述はプログラムの可読性を損なう上に、C#ではスコープが異なっても同名の変数は利用できないので、利用には注意

アレンジ演習：**p.64 scope01.cs**

・「ここではcは見えません」「ここではb,cは見えません」とコメントしてある位置で、実際に記述してエラーになることを確認しよう（確認後コメントアウト）
・また、「ここではcは見えません」とコメントしてある位置で「int c = 5;」とすると、追加した文ではなく、上の「int c = 1;」がエラーになることを確認しよう（確認後コメントアウト）

作成例

```
//アレンジ演習：p.64 scope01.cs
using System;
class scope01
{
    public static void Main()
    {
        int a = 10;
        Console.WriteLine("a = {0}", a); //ここではaが有効
        {
            int b = 5;
            Console.WriteLine("a = {0}, b = {1}", a, b); //ここではa,bが有効
            {
                int c = 1;
                Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c); //ここではa,b,cが有効
            }
            //ここではcは見えません
            Console.WriteLine("a = {0}, b = {1}", a, b);  //ここではa,bが有効
            //c = 5; //【追加】この文がエラーになる
            //int c = 5; //【追加】この文ではなく上の「int c = 1;」がエラーになる
        }
        //ここでは、b,cは見えません
        Console.WriteLine("a = {0}", a);  //ここではaが有効
        //b = 5; //【追加】この文がエラーになる
        //c = 5; //【追加】この文がエラーになる
    }
}
```

p.65 型変換

・整数型と実数型において、扱える範囲が広いものを「大きな型」、その反対を「小さな型」と呼ぶ。
・大きな型に小さな型の値を代入すると、一部の場合を除いて、自動的に型変換が行われて代入が成功する
・これを暗黙の型変換という
　例：int i = 10; double d = i; //iの値がint型からdouble型に暗黙の型変換されて代入

・例外としては、decimal型からfloat型などがある
　例：decimal m = 10; float f = m; //decimal型からfloat型への暗黙の型変換は不可なのでエラー
※ 暗黙の型変換が可能かどうかは p.95 表4.2 参照

p.65 型変換：型キャスト

・プログラマの責任において強制的に型変換を行うことが可能で、これを型キャストという
・例えば、格納されている値の範囲から見て、小さい型に代入可能な場合などに行う
・型キャストの書式：(型)変数や式
・なお、float型からdecimal型への暗黙の型変換は不可だが、型キャストなら可能
　例：decimal m = 10; float f = **(float)**m; //decimal型からfloat型への型キャストは可能
・型キャストができなかった場合、文法エラーまたは実行時エラーになるので注意
　例：char c = 'a'; bool x = (bool)c; //char型からbool型への型キャストはできないので文法エラー

p.66 cast01.cs 補足

・このプログラムは実行しても何も表示されない

アレンジ演習：**p.66 cast01.cs** 補足

・変数a、bの値を表示する処理を追加しよう
・また、上記の型変換と型キャストの例を追記して、エラーの有無を確認しよう
　※エラーになったらコメントアウトすること

作成例

```
//アレンジ演習：p.66 cast01.cs
using System;
class cast01
{
    public static void Main()
    {
        long a = 2;
        byte b;
        b = (byte)a; //long型からbyte型にキャストすれば代入可能
        //【以下追加】
        Console.WriteLine("long a = {0}, byte b = {1}", a, b);
        int i = 10; double d = i; //iの値がint型からdouble型に暗黙の型変換されて代入
        Console.WriteLine("int i = {0}, double d = {1}", i, d);
        //decimal m = 10; float f = m; //decimal型からfloat型への暗黙の型変換は不可なのでエラー
        decimal m = 10; float f = (float)m; //decimal型からfloat型への型キャストは可能
        Console.WriteLine("decimal m = {0}, float f = {1}", m, f);
        //char c = 'a'; bool x = (bool)c; //char型からbool型への型キャストはできないので文法エラー
    }
}
```

p.66 列挙型

・列挙は識別子（列挙子）をグループ化して名前をつけたもので、プログラマが定義した型として用いることができる
・内部的には、連続する整数値に名前をつけたものになる
・定義書式：enum 列挙名 { 列挙子①, 列挙子②, … }
・例えば、じゃんけんを行うゲームで、整数値0でグー、1でチョキ、2でパーを現わすなら、
　enum janken {goo, choki, paa};
　と定義すると良い
・列挙子は「列挙名.列挙子」で利用できる。例：janken.goo、janken.choki、janken.paa
・こうすることで、プログラムの可読性が上がるのがメリット
・なお、C#が提供する列挙もあり、ゲームで最も用いるのが、Keys列挙体

https://learn.microsoft.com/ja-jp/dotnet/api/system.windows.forms.keys
これを用いると、13番であるEnterキーは「Keys.Enter」で、65番であるAキーは「Keys.A」で扱える
・月名を列挙にする場合など、先頭の列挙子の整数値を指定することもでき、それ以降は+1した整数値になる
enum mons { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Org, Sep, Oct, Nov, Dec };
・なお、Console.WriteLine/Writeで列挙子を表示すると、列挙子そのものが表示される
・列挙子が持つ整数値を表示したり、計算に用いるには、int型にキャストすると良い

提出：アレンジ演習：**enum01.cs**

・列挙子 MyMonth.Aprと、MyMonth.Mayをint型にキャストせずに表示してみよう

次回予告：p.69「オブジェクト型とボックス化」から