変数

データの保存

プログラムは常に何らかのデータ（情報）を扱い、処理の目的に使用します
データはリテラルだけではその場限りなので、保存する必要があります

現在はHDDが安くなり、誰もが大容量の記録ドライブを手に入れられるようになりましたが
保存といってもデータファイルとしてHDDに記録するのではなく
プログラムはより高速でCPUの近くにある主記憶装置（メインメモリ）に一時的に保存します

メモリのどこに保存するかは、コンパイラがコンパイル時に全て計算してくれます
私たちは、コンパイラにどのアドレスにアクセスしたいかを指示するだけでよいのです
この保存場所を表す識別子を**変数**と呼びます

変数の命名規則は、クラスの命名規則と同じです
変数を使用する場合は、まず**宣言**する必要があり
変数には、なんでも格納できるわけではなく保存するデータ型を指定する必要があるのです

type variable1 [, variable2 , ...]

type には、宣言する変数の型を示す C# のキーワードを指定します
variable1 や variable2 には、それぞれ変数の名前を宣言します
このようにカンマ , で区切ることで同じ型の変数を同じに複数宣言できます

変数の型は、次のようなキーワードが用意されています
保存する必要があるデータを定め、それに合わせて型を宣言します
宣言は C 言語と異なり、ブロック内の自由な位置で宣言できます

型宣言子	説明
object	.NET Framework の System.Object クラスに基く全ての変数の基底オブジェクト型
bool	ブーリアン値 (true / false)
byte	符号なし8bit整数
sbyte	符号付8bit整数
char	16bit Unicode
short	符号付16bit整数
ushort	符号なし16bit整数
int	符号付32bit整数
uint	符号なし32bit整数
long	符号付64bit整数
ulong	符号なし64bit整数
float	32bit浮動小数点
double	64bit浮動小数点
decimal	128bit型。高精度で金融計算に適している
string	Unicode キャラクタの文字列への参照

object と bool 型はまだ説明していないが、これは後ほど紹介する（今は必要ない）
宣言した変数には、その後何らかのデータを**代入**することができます
代入は主に、**代入演算子 =** を用います

variable = expression

variable には変数名を、exporession には式（次章で紹介）を指定します
数学と違い、プログラムは左辺に右辺の値が渡されます

```
class Test {
 static void Main() {
  int i;
  double d;
  string str;

  i = 1024;
  d = 0.015;
  str = "Kitty on your lap";

  System.Console.WriteLine(i);
  System.Console.WriteLine(d);
  System.Console.WriteLine(str);
 }
}
```

整数型変数 i、浮動小数点型変数 d、文字列型変数 str を宣言し
その後、それぞれの変数に適した型の値を変数に代入しています
こうすれば、いつでも変数を呼び出して最後に代入した値を取得できます

変数は何度も更新することができます
なんからの値が代入されている変数は、更新すると前の値は消えてしまいます

```
class Test {
```

```
 static void Main() {
  string str;

  str = "Kitty on your lap";
  System.Console.WriteLine(str);
  str = "Tokyo mew mew";
  System.Console.WriteLine(str);
 }
}
```

最初に呼び出した WriteLine() は "Kitty on your lap" を出力し
その後 str 変数に新しい値を代入してから再び WriteLine() で出力しています
この時はすでに "Tokyo mew mew" という文字列になっているのでこれが表示されます

注意しなければならないのですが、メソッド内の変数を宣言した時
変数は使用される前に**必ず初期化**していなければなりません
初期化とは、変数の宣言後に最初に値を代入することをさしますが
これが行われていなければコンパイルすることができないのでエラーになります

```
class Test {
 static void Main() {
  string str;
  System.Console.WriteLine(str);
 }
}
```

このプログラムは、変数の宣言後に何も初期化することなく出力しようとしています
しかし変数は初期化されていないので、なんの値が入っているのかわかりません
そのため、このプログラムはコンパイルするとエラーが出るためコンパイルできません

これを防ぐ最良の手段は変数の宣言後に初期化するということですが
複雑なプログラムではフロー制御によって初期化が複数に分岐することがあるのです
これでは、ある分岐点では初期化されるが別のパターンで初期化されないことがあります
このような理由から、変数を**宣言と同時に初期化**することが一般的です

```
class Test {
 static void Main() {
  string str = "Kitty on your lap";
  System.Console.WriteLine(str);
 }
}
```

最初に初期化するべき値が決定されているのならば、このように初期化するべきでしょう
行数も短くなり、その分コードも可読性が増します

サフィックス

変数に値や文字列を代入し、データを保存することができました
ところが、リテラルでは整数型は一つしかありませんが変数には様々な型がありました

ある整数型を変数に代入する時、通常は変数の方に合わせて数値は整形されます
byte に数値を代入すれば、その数値は8ビットに整形されて代入されます
ただしこの場合、数値は変数に代入できるだけのサイズである必要があります

```
class Test {
 static void Main() {
  byte var1 = 255;
  short var2 = 32767;
  uint var3 = 4294967295;

  System.Console.WriteLine(var1);
  System.Console.WriteLine(var2);
  System.Console.WriteLine(var3);
 }
}
```

このように、数値リテラルは暗黙の変換で各種の変数型に変換されます
しかし、リテラルに明確なサイズを指定したい場合は**サフィックス**を付加します
サフィックスはそのリテラルの型を明確に示すことができるもので、リテラルの末尾に指定します

サフィックス	リテラル
無し	int、uint、long、ulong のうち、表せる型
U , u	uint、ulong のうち、表せる型
L , l	long、ulong のうち、表せる型
UL , Ul , uL LU , Lu , lU , lu	ulong 型

小文字の l は 1 と読み違える可能性があるため、基本的に L を使うことが推奨されます
サフィックスは、型を明示することで暗黙の変換を防ぎます
これは、数値の型が厳密に問われる処理などでは重要になることがあります

```
class Test {
 static void Main() {
  ulong var = 255UL;
  System.Console.WriteLine(var);
 }
```

```
 }
```

このプログラムは数値リテラルのサフィックスを明示しています
変数の型を int などに変更してコンパイルするとエラーを確認できます

実数リテラルにも同様にサフィックスを指定することができます
これも、整数型リテラルと考え方は同じです

サフィックス	リテラル
無し	double型
F , f	float型
D , d	double型
M , m	decimal型

実数型は整数型と異なり**変換は行われない**ことに注意してください
整数型はサフィックスが無くても表現できる範囲であれば格納できましたが
実数型はサフィックスが無い場合は double 型であると解釈されます
たとえば、実数リテラルを float に代入するには、F サフィックスを指定する必要があります

```
class Test {
 static void Main() {
  float var1 = 0.02F;
  double var2 = 12.5D;
  decimal var3 = 234.63M;

  System.Console.WriteLine(var1);
  System.Console.WriteLine(var2);
  System.Console.WriteLine(var3);
 }
}
```

このプログラムはサフィックスを指定しているため実数リテラルを各型に代入できますが
サフィックスを取り除くと double 以外はエラーが出ることが確認できます

負の単項演算子

変数の型では、int と uint というような同じサイズの型がそれぞれ用意されています
int は uint と同じサイズですが**符号付**という点で異なっています

符号とは、数値型で**負数**を表現するための手段です
コンピュータは負数を最上位ビットで判断し、最上位ビットが1のときは
以下のビットの2の補数の負の値であると解釈されます（詳しくはコンピュータ科学の分野となる）
そのため、負数を表現する時は最上位ビットが符号表現のために必要になり
結果として負数を扱う変数はビットで表現できる値の半分の数値が表現できる最大値となる
このような理由から負数を使わないならば符号なしの変数を使ったほうが合理的なのです

負数はプログラムで表現する時は**負の単項演算子 -** を数値の前に指定します
さらに、変数の前に - を指定して変数が格納する値の**符号を反転**できます

```
class Test {
 static void Main() {
  int var = 100;
  System.Console.WriteLine(-10);
  System.Console.WriteLine(-var);

  var = -1000;
  System.Console.WriteLine(-var);
 }
}
```

このプログラムの結果は次のようになります

-10
-100
1000

最初の WriteLine(-10) は数値リテラル 10 の負数を表しています
次の WriteLine(-var) は変数 var に格納されている 100 という値を負数にしています

注目は、最後の結果です
var に -1000 という数値を代入し、この値の負数を -var で表現しています
結果 -1000 の負数は 1000 というように符号が逆になって表示されます

では、正確には負の単項演算子は符号を逆にするものなのでしょうか？
正しい考え方としては**2の補数を求める**とするべきでしょう
（2の補数とは、1の補数に 1 加算したもので、1の補数とはビットを反転したものである）

つまり、本来 -1000 という値は機械語で 1000 の2の補数として表現されます
1000は2進数で 0011 1110 1000 であり、この2の補数は 1100 0001 1000 です
この値が var に格納されていて WriteLine(-var) で 1100 0001 1000 の2の補数が求められ
この結果 0011 1110 1000 となるので10進数の 1000 という値が表示されたのです

では、逆に + 演算子を使えば負数を整数に返られるのかと考えますが

これは何の意味もありません
変数の絶対値を求めたい場合は、変数が 0 以下であるかどうかを調べ
0 以下であればその2の補数を求めることで絶対値を求めます

前のページへ　戻る　次のページへ