عنوان مضمون

Visual Programming-I

توسط : صفری

خزان1397

C# data type and type conversion
C# variable, constant and literals

VARIABLES

variables are concerned with the storage of data.

- C# syntax for declaring variables merely specifies the type and variable name:

<type> < var-name >;

double d;

- where *type* is the data type of the variable and *var-name* is its name.

<data_type> <variable_list>;

int i, j, k;

char c, ch;

float f, salary;

Simple Types

- Simple types include types such as numbers and Boolean (true or false) values that make up the fundamental building blocks for your applications.

TYPE	ALIAS FOR	ALLOWED VALUES
sbyte	System.SByte	Integer between –128 and 127
byte	System.Byte	Integer between 0 and 255
short	System.Int16	Integer between –32768 and 32767
ushort	System.UInt16	Integer between 0 and 65535
int	System.Int32	Integer between –2147483648 and 2147483647
uint	System.UInt32	Integer between 0 and 4294967295
long	System.Int64	Integer between –9223372036854775808 and 9223372036854775807
ulong	System.UInt64	Integer between 0 and 18446744073709551615

Simple Types

TYPE	ALIAS FOR	MIN M	MAX M	MIN E	MAX E	APPROX MIN VALUE	APPROX MAX VALUE
`float`	`System.Single`	0	2^{24}	−149	104	1.5×10^{-45}	3.4×10^{38}
`double`	`System.Double`	0	2^{53}	−1075	970	5.0×10^{-324}	1.7×10^{308}
`decimal`	`System.Decimal`	0	2^{96}	−28	0	1.0×10^{-28}	7.9×10^{28}

TYPE	ALIAS FOR	ALLOWED VALUES
`char`	`System.Char`	Single Unicode character, stored as an integer between 0 and 65535
`bool`	`System.Boolean`	Boolean value, `true` or `false`
`string`	`System.String`	A sequence of characters

Variable Naming

- The basic variable naming rules are as follows:

➢ The first character of a variable name must be either a letter, an underscore character(_), or the *at* symbol (@).

➢ Subsequent characters may be letters, underscore characters, or numbers.

➢certain keywords that have a specialized meaning to the C# compiler, such as the using and namespace keywords shown earlier. If you use one of these by mistake, the compiler complains

➢Remember that C# is case sensitive

Variable Naming

- For example, the following variable names are fi ne:

  myBigVar

  VAR1

  _test

- These are not, however:

  99BottlesOfBeer

  namespace

  It's-All-Over

Variable Declaration and Assignment

- To recap, recall that you declare variables simply using their type and name:

```
int age;
int xSize, ySize;
```

- You then assign values to variables using the = assignment operator:

```
age = 25;
int xSize = 4, ySize = 5;
```

TRY IT OUT

- Create a new console application
- Add the following code to Program.cs:

```
static void Main(string[] args)
{    int myInteger;
     string myString;
     myInteger = 17;
     myString = "\"myInteger\" is";
     Console.WriteLine("{0} {1}.", myString, myInteger);
     Console.ReadKey();
}
```

- Execute the code.

TYPE CONVERSION

- However, in general, the different types of variables use varying schemes to represent data. This implies that even if it were possible to place the sequence of bits from one variable into a variable of a different type.
- Type conversion is converting one type of data to another type. It is also known as Type Casting.
- For example, you might want to assign an **int** value to a **float** variable, as shown here:

```
int i;
float f;
i =10;
F= i;
```

TYPE CONVERSION

- Type conversion takes two forms:

➤ **Implicit conversion** –(**Automatic Conversions**) These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.

➤ **Explicit conversion** —(**Casting Incompatible Types**)These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

Implicit type conversion –(Automatic Conversions)

When one type of data is assigned to another type of variable, an *implicit* type conversion will take place automatically if:

- **The two types are compatible.**
- **The destination type has a range that is greater than the source type.**

When these two conditions are met, a *widening conversion* takes place. For example, the **int(32 bits)** type is always large enough to hold all valid **byte(8 bits)** values, and both **int** and **byte** are compatible integer types, so an implicit conversion can be applied.

For example, the following program is perfectly valid since **long** to **double** is a widening conversion that is automatically performed.

```
class LtoD {
        static void Main() {
             long L;
            double D;
            L=100123285L;
            D=L;
            Console.WriteLine("L and D: " + L + " " + D);
         }
}
```

Although there is an implicit conversion from long to double, there is no implicit conversion from double to long since this is not a widening conversion. Thus, the following version of the preceding program is invalid:

```
using System;
class LtoD {
        static void Main() {
             long L;
            double D;
            D=100123285.0;
            L=D; // Illegal!!!
            Console.WriteLine("L and D: " + L + " " + D);
        }
}
```

Implicit type conversion –(Automatic Conversions)

TYPE	CAN SAFELY BE CONVERTED TO
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Explicit type conversion – (Casting Incompatible Types)

- For example, the following modification to the code from the last section attempts to convert a short value into a byte:

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

- If you attempt to compile the preceding code, you will receive the following error:

  Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists (are you missing a cast?)

Explicit type conversion – (Casting Incompatible Types)

Although the implicit type conversions are helpful, they will not fulfill all programming needs because they apply only to widening conversions between compatible types. For all other cases you must employ a cast. A *cast* is an instruction to the compiler to convert the outcome of an expression into a specified type. Thus, it requests an explicit type conversion. A cast has this general form:

(*target-type*) *expression*

(int) (x / y)

```
using System;
namespace TypeConversionApplication
{
  class ExplicitConversion
  {
    static void Main(string[] args)
    {
      double d = 5673.74;
      int i;

      // cast double to int.
      i = (int)d;
      Console.WriteLine(i);
      Console.ReadKey();
    }
  }
}
```

Explicit Conversions Using the Convert Commands

- You can specify many such explicit conversions in this way:

COMMAND	RESULT
`Convert.ToBoolean(val)`	`val` converted to `bool`
`Convert.ToByte(val)`	`val` converted to `byte`
`Convert.ToChar(val)`	`val` converted to `char`
`Convert.ToDecimal(val)`	`val` converted to `decimal`
`Convert.ToDouble(val)`	`val` converted to `double`
`Convert.ToInt16(val)`	`val` converted to `short`
`Convert.ToInt32(val)`	`val` converted to `int`
`Convert.ToInt64(val)`	`val` converted to `long`
`Convert.ToSByte(val)`	`val` converted to `sbyte`
`Convert.ToSingle(val)`	`val` converted to `float`
`Convert.ToString(val)`	`val` converted to `string`
`Convert.ToUInt16(val)`	`val` converted to `ushort`
`Convert.ToUInt32(val)`	`val` converted to `uint`
`Convert.ToUInt64(val)`	`val` converted to `ulong`

Explicit Conversions Using the Convert Commands

Accepting Values from User:

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

```
int num;

num = Convert.ToInt32(Console.ReadLine());
```

Constants and Literals

- The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

  Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.

Defining Constants

Constants are defined using the const keyword. Syntax for defining a constant is −

const <data_type> <constant_name> = value;

String Literals

- you saw a few of the escape sequences you can use in string literals.

ESCAPE SEQUENCE	CHARACTER PRODUCED	UNICODE (HEX) VALUE OF CHARACTER
\'	Single quotation mark	0x0027
\"	Double quotation mark	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert (causes a beep)	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B