**Lecture 2 – Data types and Opeartors**

The varibles in C#, are categorized into the following types:

* Value types
* Reference types
* Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class**System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int**type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof**method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine:

using System;

namespace DataTypeApplication

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Size of int: {0}", sizeof(int));

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result:

Size of int: 4

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of**built-in** reference types are: **object**, **dynamic,** and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

object obj;

obj = 100; // this is boxing

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is:

dynamic <variable\_name> = value;

For example,

dynamic d = 20;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

String str = "Tutorials Point";

A @quoted string literal looks as follows:

@"Tutorials Point";

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is:

type\* identifier;

For example,

char\* cptr;

int\* iptr;

The varibles in C#, are categorized into the following types:

* Value types
* Reference types
* Pointer types

Value Type

Value type variables can be assigned a value directly. They are derived from the class**System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int**type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof**method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine:

using System;

namespace DataTypeApplication

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Size of int: {0}", sizeof(int));

Console.ReadLine();

}

}

}

When the above code is compiled and executed, it produces the following result:

Size of int: 4

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of**built-in** reference types are: **object**, **dynamic,** and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

object obj;

obj = 100; // this is boxing

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is:

dynamic <variable\_name> = value;

For example,

dynamic d = 20;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

String str = "Tutorials Point";

A @quoted string literal looks as follows:

@"Tutorials Point";

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is:

type\* identifier;

For example,

char\* cptr;

int\* iptr;

**Operators and Operands**

An operation is an action performed on one or more values either to modify the value held by one or both of the variables, or to produce a new value by combining existing values. Therefore, an operation is performed using at least one symbol and at least one value. The symbol used in an operation is called an operator. A value involved in an operation is called an operand.

A unary operator is an operator that performs its operation on only one operand. An operator is referred to as binary if it operates on two operands.

|  |
| --- |
| **Curly Brackets { }** |

Curly brackets are probably the most used and the most tolerant operators of C#. Fundamentally, curly brackets are used to create a section of code. As such they are required to delimit the bodies of namespaces, classes, structures, exceptions, etc. They are also optionally used in conditional statements. Curly brackets are also used to create variable scope.

|  |
| --- |
| **Parentheses ( )** |

Parentheses are used to isolate a group of items that must be considered as belonging to one entity. For example, parentheses are used to differentiate a method such as **Main** from a regular variable. Here is an example:

class Exercise

{

static void Main()

{

}

}

Parentheses can also be used to isolate an operation or an expression with regard to another operation or expression.

|  |
| --- |
| **Semi-Colon;** |

The semi-colon is used to indicate the end of an expression or a declaration. Here is an example:

int number;

As we will learn, there are other uses of the semi-colon.

|  |
| --- |
| **The Comma ,** |

The comma is used to separate variables used in a group. For example, a comma can be used to delimit the names of variables that are declared with the same data type. Here is an example:

class Exercise

{

static void Main()

{

string firstName, lastName, fullName;

}

}

The comma can also be used to separate the member of an enumeration or the arguments of a method. We will review all of them when the time comes.

|  |
| --- |
| **The Assignment =** |

When you declare a variable, a memory space is reserved for it. That memory space may be empty until you fill it with a value. To "put" a value in the memory space allocated to a variable, you can use the assignment operator represented as =. Based on this, the assignment operation gives a value to a variable. Its syntax is:

*VariableName* = *Value*

The *VariableName* factor must be a valid variable name. It cannot be a value such as a numeric value or a (double-quoted) string. Here is an example that assigns a numeric value to a variable:

class Exercise

{

static void Main()

{

decimal salary;

// Using the assignment operator

salary = 12.55M;

}

}

Once a variable has been declared and assigned a value, you can call **Write()** or **WriteLine()** to display its value.  Here is an example:

class Exercise

{

static void Main()

{

decimal salary;

// Using the assignment operator

salary = 12.55M;

System.Console.Write("Employee's Hourly Salary: ");

System.Console.WriteLine(salary);

}

}

This would produce:

Employee's Hourly Salary: $12.55

The above code declares a variable before assigning it a value. You will usually perform this assignment when you want to change the value held by a variable. Providing a starting value to a variable when the variable is declared is referred to as initializing the variable. Here is an example:

class Exercise

{

static void Main()

{

// Using the assignment operator

decimal salary = 12.55M;

System.Console.Write("Employee's Hourly Salary: ");

System.Console.WriteLine(salary);

}

}

We saw that you can declare various variables at once by using the same data type but separating their names with commas. When doing this, you can also initialize each variable by assigning it the desired value before the comma or the semi-colon. Here is an example:

class Exercise

{

static void Main()

{

// Initializing various variables when declaring them with the same data type

double value1 = 224.58, value2 = 1548.26;

System.Console.Write("Value 1 = ");

System.Console.WriteLine(value1);

System.Console.Write("Value 2 = ");

System.Console.WriteLine(value2);

System.Console.WriteLine();

}

}

This would produce:

Value 1 = 224.58

Value 2 = 1548.26

|  |
| --- |
| **Single-Quote '** |

The single quote is used to include one character to initialize, or assign a symbol to, a variable declared as **char**. A single-quote is usually combined with another and a character is included between them. Here is an example:

class Exercise

{

static void Main()

{

char gender;

string firstName, lastName, fullName;

gender = 'M';

}

}

You can include only one character between single-quotes except if the combination of symbols can be evaluated to one character. This is the case for escape sequences. Here is an example:

class Exercise

{

static void Main()

{

System.Console.WriteLine('\n');

}

}

|  |
| --- |
| **Double-Quotes "** |

The double-quote " is used to delimit a string. Like the single-quote, the double-quote is usually combined with another. Between the combination of double-quotes, you can include an empty space, a character, a word, or a group of words, making it a string. Here is an example:

class Exercise

{

static void Main()

{

System.Console.WriteLine("The Wonderful World of C#!!!");

}

}

A double-quoted string can also be declared and then assigned to a variable.

|  |
| --- |
| **Square Brackets [ ]** |

Square brackets are mostly used to control the dimension or index of an [**array**](http://www.functionx.com/csharp4/Lesson33.htm). We will learn how to use them when we study arrays.

|  |
| --- |
| **The Positive Operator +** |

Algebra uses a type of ruler to classify numbers. This ruler has a middle position of zero. The numbers on the left side of the 0 are referred to as negative while the numbers on the right side of the rulers are considered positive:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -∞ |  | -6 | -5 | -4 | -3 | -2 | -1 |  | 1 | 2 | 3 | 4 | 5 | 6 |  | +∞ |
| 0 | | | | | | | | | | | | | | | | |
| -∞ |  | -6 | -5 | -4 | -3 | -2 | -1 |  | 1 | 2 | 3 | 4 | 5 | 6 |  | +∞ |

A value on the right side of 0 is considered positive. To express that a number is positive, you can write a + sign on its left. Examples are +4, +228, +90335. In this case the + symbol is called a unary operator because it acts on only one operand. The positive unary operator, when used, must be positioned on the left side of its operand, never on the right side.

As a mathematical convention, when a value is positive, you don't need to express it with the + operator. Just writing the number without any symbol signifies that the number is positive. Therefore, the numbers +4, +228, and +90335 can be, and are better, expressed as 4, 228, 90335. Because the value doesn't display a sign, it is referred as **unsigned**.

To express a variable as positive or **unsigned**, you can just type it. here is an example:

class Exercise

{

static void Main()

{

// Displaying an unsigned number

System.Console.Write("Number = ");

System.Console.WriteLine(+802);

}

}

This would produce:

Number = 802

|  |
| --- |
| **The Negative Operator -** |

As you can see on the above ruler, in order to express any number on the left side of 0, it must be appended with a sign, namely the - symbol. Examples are -12, -448, -32706. A value accompanied by - is referred to as negative. The - sign must be typed on the left side of the number it is used to negate. Remember that if a number does not have a sign, it is considered positive. Therefore, whenever a number is negative, it MUST have a - sign. In the same way, if you want to change a value from positive to negative, you can just add a - sign to its left.  
  
Here is an example that uses two variables. One has a positive value while the other has a negative value:

class Exercise

{

static void Main()

{

// Displaying an unsigned number

System.Console.Write("First Number ");

System.Console.WriteLine(+802);

// Displaying a negative number

System.Console.Write("Second Number ");

System.Console.WriteLine(-802);

}

}

This would produce:

First Number 802

Second Number -802

|  |
| --- |
| **Unary Operators: The sizeof Operator** |

We have already seen that when declaring a variable, the compiler reserves a portion of space in the computer memory to hold that variable. We also illustrated how much space some data types need to store their variable. C# provides the unary **sizeof** operator that allows you to find out how much space a data type.

To find out how much memory space a data type uses, type it between the parentheses of the **sizeof**operator. Here is an example:

class Exercise

{

static void Main()

{

double period = 155.50;

int size = sizeof(double);

System.Console.Write("The value ");

System.Console.Write(period);

System.Console.Write(" uses ");

System.Console.Write(size);

System.Console.WriteLine(" bytes\n");

}

}

This would produce:

The value 155.5 uses 8 bytes

Here are a few more example:

class Exercise

{

static void Main()

{

// The sizeof operator used to get the memory size used by

// a variable declared with a certain a data type

System.Console.WriteLine("The sizeof Operator");

System.Console.WriteLine("==========================");

System.Console.WriteLine("Data Type Memory Size");

System.Console.WriteLine("--------------------------");

System.Console.Write("char ");

System.Console.Write(sizeof(char));

System.Console.WriteLine(" Bytes");

System.Console.Write("bool ");

System.Console.Write(sizeof(bool));

System.Console.WriteLine(" Bytes");

System.Console.Write("int ");

System.Console.Write(sizeof(int));

System.Console.WriteLine(" Bytes");

System.Console.Write("uint ");

System.Console.Write(sizeof(uint));

System.Console.WriteLine(" Bytes");

System.Console.Write("short ");

System.Console.Write(sizeof(short));

System.Console.WriteLine(" Bytes");

System.Console.Write("ushort ");

System.Console.Write(sizeof(ushort));

System.Console.WriteLine(" Bytes");

System.Console.Write("byte ");

System.Console.Write(sizeof(byte));

System.Console.WriteLine(" Bytes");

System.Console.Write("sbyte ");

System.Console.Write(sizeof(sbyte));

System.Console.WriteLine(" Bytes");

System.Console.Write("float ");

System.Console.Write(sizeof(float));

System.Console.WriteLine(" Bytes");

System.Console.Write("double ");

System.Console.Write(sizeof(double));

System.Console.WriteLine(" Bytes");

System.Console.Write("decimal ");

System.Console.Write(sizeof(decimal));

System.Console.WriteLine(" Bytes");

System.Console.Write("long ");

System.Console.Write(sizeof(long));

System.Console.WriteLine(" Bytes");

System.Console.Write("ulong ");

System.Console.Write(sizeof(ulong));

System.Console.WriteLine(" Bytes");

System.Console.WriteLine("===========================");

System.Console.WriteLine();

}

}

This would produce:

The sizeof Operator

==========================

Data Type Memory Size

--------------------------

char 2 Bytes

bool 1 Bytes

int 4 Bytes

uint 4 Bytes

short 2 Bytes

ushort 2 Bytes

byte 1 Bytes

sbyte 1 Bytes

float 4 Bytes

double 8 Bytes

decimal 16 Bytes

long 8 Bytes

ulong 8 Bytes

===========================

|  |
| --- |
| **The Addition Operations** |

|  |
| --- |
| **Introduction** |

The addition is an operation used to add things of the same nature one to another, as many as necessary. Sometimes, the items are added one group to another. The concept is still the same, except that this last example is faster. The addition is performed in mathematics using the + sign. The same sign is used in C#.

To get the addition of two values, you add the first one to the other. After the addition of two values has been performed, you get a new value. This means that if you add value1 to value2, you would write value1 + value2. The result is another value we could call Value3. You can also add more than two values, like a + b + c.

With numbers, the order you use to add two or more values doesn't matter. This means that value1 + value2 is the same as value2 + value1. In the same way a + b + c is the same as a + c + b the same as b + a + c and the same as c + b + a

Here is an example that adds two numbers:

class Exercise

{

static void Main()

{

System.Console.Write("244 + 835 = ");

System.Console.WriteLine(244 + 835);

}

}

Here is the result:

244 + 835 = 1079

You can also add some values already declared and initialized in your program. You can also get the values from the user.

In C#, you can also add string variables to add a new string. The operation is performed as if you were using numbers. For example, "Pie" + "Chart" would produce "PieChart". You can also add add as many strings as possible by including the + operator between them. Here is an example:

class Exercise

{

static void Main()

{

var firstName = "Alexander";

var lastName = "Kallack";

var fullName = firstName + " " + lastName;

System.Console.Write("Full Name: ");

System.Console.WriteLine(fullName);

}

}

This would produce:

Full Name: Alexander Kallack

Press any key to continue . . .

|  |
| --- |
| **Incrementing a Variable** |

We are used to counting numbers such as 1, 2, 3, 4, etc. In reality, when counting such numbers, we are simply adding 1 to a number in order to get the next number in the range. The simplest technique of incrementing a value consists of adding 1 to it. After adding 1, the value or the variable is (permanently) modified and the variable would hold the new value. This is illustrated in the following example:

// This program studies value incrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of incrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value = value + 1;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of incrementing a value

Value = 12

Value = 13

C# provides a special operator that takes care of this operation. The operator is called the increment operator and is represented by ++. Instead of writing Value = Value + 1, you can write Value++ and you would get the same result. The above program can be re-written as follows:

// This program studies value incrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of incrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value++;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

The ++ is a unary operator because it operates on only one variable. It is used to modify the value of the variable by adding 1 to it. Every time the Value++ is executed, the compiler takes the previous value of the variable, adds 1 to it, and the variable holds the incremented value:

// This program studies value incrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of incrementing a value");

value++;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value++;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

Value++;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of incrementing a value

Value = 13

Value = 14

Value = 15

|  |
| --- |
| **Pre and Post-Increment** |

When using the ++ operator, the position of the operator with regard to the variable it is modifying can be significant. To increment the value of the variable before re-using it, you should position the operator on the left of the variable:

// This program studies value incrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of incrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

System.Console.Write("Value = ");

System.Console.WriteLine(++value);

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of incrementing a value

Value = 12

Value = 13

Value = 13

When writing ++Value, the value of the variable is incremented before being called. On the other hand, if you want to first use a variable, then increment it, in other words, if you want to increment the variable after calling it, position the increment operator on the right side of the variable:

// This program studies value incrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of incrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

System.Console.Write("Value = ");

System.Console.WriteLine(value++);

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of incrementing a value

Value = 12

Value = 12

Value = 13

|  |
| --- |
| **Compound Addition** |

It is not unusual to add a constant value to a variable. All you have to do is to declare another variable that would hold the new value. Here is an example:

// This program studies value incrementing and decrementing

public class Exercise

{

static void Main()

{

double value = 12.75;

double newValue;

System.Console.WriteLine("Techniques of incrementing and decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

newValue = value + 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(newValue);

}

}

This would produce:

Techniques of incrementing and decrementing a value

Value = 12.75

Value = 15.17

The above technique requires that you use an extra variable in your application. The advantage is that each value can hold its own value although the value of the second variable depends on whatever would happen to the original or source variable. Sometimes in your program you will not need to keep the original value of the source variable. You may want to permanently modify the value that a variable is holding. In this case, you can perform the addition operation directly on the variable by adding the desired value to the variable. This operation modifies whatever value a variable is holding and does not need an additional variable.

// This program studies value incrementing and decrementing

public class Exercise

{

static void Main()

{

var value = 12.75;

System.Console.WriteLine("Techniques of incrementing and decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value += 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This program produces the same result as the previous.

|  |
| --- |
| **The Multiplication Operations** |

|  |
| --- |
| **Introduction** |

The multiplication allows adding one value to itself a certain number of times, set by a second value. As an example, instead of adding a value to itself in this manner: A + A + A + A, since the variable a is repeated over and over again, you could simply find out how many times A is added to itself, then multiply a by that number which, is this case, is 4. This would mean adding a to itself 4 times, and you would get the same result.

Just like the addition, the multiplication is associative: a \* b \* c = c \* b \* a. When it comes to programming syntax, the rules we learned with the addition operation also apply to the multiplication.

Here is an example:

class Exercise

{

static void Main()

{

// Initializing various variables when declaring them with the same data type

double value1 = 224.58, value2 = 1548.26;

var result = value1 \* value2;

System.Console.Write(value1);

System.Console.Write(" \* ");

System.Console.Write(value2);

System.Console.Write(" = ");

System.Console.WriteLine(result);

System.Console.WriteLine();

}

}

|  |
| --- |
| **Compound Multiplication** |

We saw that you could add or subtract a value to a variable and assign the result to the same variable. You can perform the same operation with the multiplication. Here is an example:

class Exercise

{

static void Main()

{

double value = 12.75;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value = Value \* 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Value = 12.75

Value = 30.855

Press any key to continue . . .

To make this operation easy, the C# language supports the compound multiplication assignment operator represented as \*=. To use it, use the \*= operator and assign the desired value to the variable. Here is an example:

class Exercise

{

static void Main()

{

double value = 12.75;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value \*= 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

|  |
| --- |
| **The Subtraction Operations** |

|  |
| --- |
| **Introduction** |

The subtraction operation is used to take out or subtract a value from another value. It is essentially the opposite of the addition. The subtraction is performed with the - sign. Here is an example:

class Exercise

{

static void Main()

{

// Values used in this program

double value1 = 224.58, value2 = 1548.26;

var result = value1 - value2;

System.Console.Write(value1);

System.Console.Write(" - ");

System.Console.Write(value2);

System.Console.Write(" = ");

System.Console.WriteLine(result);

}

}

This would produce:

224.58 - 1548.26 = -1323.68

Press any key to continue . . .

Unlike the addition, the subtraction is not associative. In other words, a - b - c is not the same as c - b - a. Consider the following program that illustrates this:

class Exercise

{

static void Main()

{

// This tests whether the addition is associative

System.Console.WriteLine(" =+= Addition =+=");

System.Console.Write("128 + 42 + 5 = ");

System.Console.WriteLine(128 + 42 + 5);

System.Console.Write(" 5 + 42 + 128 = ");

System.Console.WriteLine(5 + 42 + 128);

System.Console.WriteLine();

// This tests whether the subtraction is associative

System.Console.WriteLine(" =-= Subtraction =-=");

System.Console.Write("128 - 42 - 5 = ");

System.Console.WriteLine(128 - 42 - 5);

System.Console.Write(" 5 - 42 - 128 = ");

System.Console.WriteLine(5 - 42 - 128);

System.Console.WriteLine();

}

}

This would produce:

=+= Addition =+=

128 + 42 + 5 = 175

5 + 42 + 128 = 175

=-= Subtraction =-=

128 - 42 - 5 = 81

5 - 42 - 128 = -165

Notice that both operations of the addition convey the same result. In the subtraction section, the numbers follow the same order but produce different results.

|  |
| --- |
| **Decrementing a Variable** |

When counting numbers backward, such as 8, 7, 6, 5, etc, we are in fact subtracting 1 from a value in order to get the lesser value. This operation is referred to as decrementing a value. This operation works as if a value is decremented by 1, as in Value = Value ГѓЖ’Г†вЂ™ГѓвЂљГ‚ВўГѓЖ’Г‚ВўГѓВўГўвЂљВ¬Г…ВЎГѓвЂљГ‚В¬ГѓЖ’Г‚ВўГѓВўГўв‚¬ЕЎГ‚В¬ГѓвЂ¦Гўв‚¬Е“ 1:

// This program studies value decrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value = value - 1;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of decrementing a value

Value = 12

Value = 11

As done to increment, C# provides a quicker way of subtracting 1 from a value. This is done using the decrement operator, that is --. To use the decrement operator, type ГѓЖ’Г†вЂ™ГѓвЂљГ‚ВўГѓЖ’Г‚ВўГѓВўГўвЂљВ¬Г…ВЎГѓвЂљГ‚В¬ГѓЖ’Г‚ВўГѓВўГўв‚¬ЕЎГ‚В¬ГѓвЂ¦Гўв‚¬Е“- on the left or the right side of the variable when this operation is desired. Using the decrement operator, the above program could be written:

// This program studies value decrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value--;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

|  |
| --- |
| **Pre-Decrementing a Value** |

Once again, the position of the operator can be important. If you want to decrement the variable before calling it, position the decrement operator on the left side of the operand. This is illustrated in the following program:

// This program studies value decrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

System.Console.Write("Value = ");

System.Console.WriteLine(--value);

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of decrementing a value

Value = 12

Value = 11

Value = 11

If you plan to decrement a variable only after it has been accessed, position the operator on the right side of the variable. Here is an example:

// This program studies value decrementing

public class Exercise

{

static void Main()

{

var value = 12;

System.Console.WriteLine("Techniques of decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

System.Console.Write("Value = ");

System.Console.WriteLine(Value--);

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of decrementing a value

Value = 12

Value = 12

Value = 11

|  |
| --- |
| **Compound Subtraction** |

You may want to subtract a constant value from a variable. To decrement a value from a variable, use the subtraction and apply the same technique. This is done with the -= operator. Here is an example:

// This program studies value incrementing and decrementing

public class Exercise

{

static void Main()

{

var value = 12.75;

System.Console.WriteLine("Techniques of incrementing and decrementing a value");

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value -= 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Techniques of incrementing and decrementing a value

Value = 12.75

Value = 10.33

|  |
| --- |
| **The Division Operation** |

|  |
| --- |
| **Introduction** |

Dividing an item means cutting it in pieces or fractions of a set value. For example, when you cut an apple in the middle, you are dividing it in 2 pieces. If you cut each one of the resulting pieces, you will get 4 pieces or fractions. This is considered that you have divided the apple in 4 parts. Therefore, the division is used to get the fraction of one number in terms of another. The division is performed with the forward slash /. Here is an example:

class Exercise

{

static void Main()

{

// Initializing various variables when declaring them with the same data type

double value1 = 224.58, value2 = 1548.26;

var result = value1 / value2;

System.Console.Write(value1);

System.Console.Write(" / ");

System.Console.Write(value2);

System.Console.Write(" = ");

System.Console.WriteLine(result);

System.Console.WriteLine();

}

}

This would produce:

224.58 / 1548.26 = 0.145053156446592

Press any key to continue . . .

When performing the division, be aware of its many rules. Never divide by zero (0). Make sure that you know the relationship(s) between the numbers involved in the operation.

|  |
| --- |
| **Compound Division** |

Remember that you can add, subtract, or multiply a value to a variable and assign the result to the variable itself. You can also perform this operation using the division. Here is an example:

class Exercise

{

static void Main()

{

double value = 12.75;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value = value / 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

This would produce:

Value = 12.75

Value = 5.26859504132231

Press any key to continue . . .

To perform this operation faster, the C# language provides the /= operator. Here is an example of using it:

class Exercise

{

static void Main()

{

double value = 12.75;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

value /= 2.42;

System.Console.Write("Value = ");

System.Console.WriteLine(value);

}

}

|  |
| --- |
| **Introduction** |

The division program gives a result of a number with decimal values if you provide an odd number (like 147), which is fine in some circumstances. Sometimes you will want to get the value remaining after a division renders a natural result. Imagine you have 26 kids at a football (soccer) stadium and they are about to start. You know that you need 11 kids for each team to start. If the game starts with the right amount of players, how many will seat and wait?

The remainder operation is performed with the percent sign (%) which is gotten from pressing Shift + 5.

Here is an example:

class Exercise

{

static void Main()

{

var players = 18;

var remainder = players % 11;

// When the game starts, how many players will wait?.

System.Console.Write("Out of ");

System.Console.Write(players);

System.Console.Write(" players, ");

System.Console.Write(remainder);

System.Console.WriteLine(" players will have to wait when the game starts.\n");

}

}

This would produce:

Out of 18 players, 7 players will have to wait when the game starts.

Press any key to continue . . .

|  |
| --- |
| **The Compound Remainder** |

As seen with the other arithmetic operators, you can find the remainder of a variable and assign the result to the variable itself. Here is an example:

class Exercise

{

static void Main()

{

var players = 18;

// When the game starts, how many players will wait?.

System.Console.Write("Out of ");

System.Console.Write(players);

System.Console.Write(" players, ");

players = players % 11;

System.Console.Write(players);

System.Console.WriteLine(" players will have to wait when the game starts.\n");

}

}

To support a faster version of this operation, the C# language provides the compound remainder operator represented as %=. To use it, assign its value to the variable. Here is an example:

class Exercise

{

static void Main()

{

var Players = 18;

// When the game starts, how many players will wait?.

System.Console.Write("Out of ");

System.Console.Write(players);

System.Console.Write(" players, ");

players %= 11;

System.Console.Write(players);

System.Console.WriteLine(" players will have to wait when the game starts.\n");

}

}

|  |
| --- |
| **Bit Operations** |

|  |
| --- |
| **Introduction** |

From our introduction to variables, you may remember that the computer stores its data in memory using small locations that each contains a bit of information. Because a bit can be represented only either as 1 or 0, we can say that each bit contains 1 or 0. Bit manipulation consists of changing the value (1 or 0, or 0 or 1) in a bit. As we will see in the next few operations, it is not just about changing a value. It can involve reversing a value or kind of "moving" a bit from its current position to the next position.

The operations on bits are performed on 1s and 0s only. This means that any number is decimal or hexadecimal format involved in a bit operation must be converted to binary first.

You will almost never perform some of the operations we are going to review. You will hardly perform some other operations. There is only one operation you will perform on a regular basis. The**OR** operation is very regular in Microsoft Windows (Win32) programming so much that we were obliged to include this whole section in the lesson, as opposed to mentioning only **OR**.

|  |
| --- |
| **"Reversing" a Bit** |

Remember that, at any time, a box (or chunk) in memory contains either 1 or 0:

Bit reversal consists of changing the value of a bit. If the box contains 1, it would become 0. If it contains 0, it would become 1. To support this operation, the C# language provides the bitwise negation operator represented with the ~ symbol.

As an example, consider the number 286. The decimal number 286 converted to binary is 100011110. You can reverse each bit as follows:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 286 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| Not 286 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

To use the bitwise negation operator, type ~ on the left side of the value. Here is an example:

class Exercise

{

static void Main()

{

var number1 = 286;

System.Console.Write("286 = ");

System.Console.WriteLine(number1);

System.Console.Write("Not 286 = ");

System.Console.WriteLine(~number1);

}

}

This would produce:

286 = 286

Not 286 = -287

Press any key to continue . . .

|  |
| --- |
| **Bitwise Conjunction** |

Bitwise conjunction consists of adding the content of a bit to the content of another bit. To support the bitwise conjunction operation, the C# language provides the & operator.

To perform the bit addition on two numbers, remember that they must be converted to binary first. Then:

* If a bit with value 0 is added to a bit with value 0, the result is 0

|  |  |
| --- | --- |
| Bit0 | 0 |
| Bit1 | 0 |
| Bit0 And Bit1 | 0 |

* If a bit with value 1 is added to a bit with value 0, the result is 0

|  |  |
| --- | --- |
| Bit0 | 1 |
| Bit1 | 0 |
| Bit0 And Bit1 | 0 |

* If a bit with value 0 is added to a bit with value 1, the result is 0

|  |  |
| --- | --- |
| Bit0 | 0 |
| Bit1 | 1 |
| Bit0 And Bit1 | 0 |

* If a bit with value 1 is added to a bit with value 1, the result is 1

|  |  |
| --- | --- |
| Bit0 | 1 |
| Bit1 | 1 |
| Bit0 And Bit1 | 1 |

As an example, consider the number 286 bit-added to 475. The decimal number 286 converted to binary is 100011110. The decimal number 4075 converted to binary is 111111101011. Based on the above 4 points, we can add these two numbers as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 286 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 4075 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 286 & 4075 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

Therefore, 286 & 4075 produces 100001010 which is equivalent to:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bit8 | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|  | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 286 & 4075 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
|  | 256 | 0 | 0 | 0 | 0 | 8 | 0 | 2 | 0 |

This means that 286 & 4075 = 256 + 16 + 2 = 266. To perform a bitwise conjunction, use the & operator. Here is an example:

class Exercise

{

static void Main()

{

var number1 = 286;

var number2 = 4075;

var result = number1 & number2;

System.Console.Write("286 & 4075 = ");

System.Console.WriteLine(result);

}

}

This would produce:

286 & 4075 = 266

Press any key to continue . . .

You can perform a bitwise conjunction on a variable and assign the result to the same variable. Here is an example:

class Exercise

{

static void Main()

{

var number = 286;

System.Console.WriteLine(number);

number = number & 48;

System.Console.WriteLine(number);

}

}

This would produce:

286

16

Press any key to continue . . .

Instead of performing the operation on two steps, a shorter version of this operator consists of using the &= operator. Here is an example:

class Exercise

{

static void Main()

{

var number = 286;

System.Console.WriteLine(number);

number &= 48;

System.Console.WriteLine(number);

}

}

|  |
| --- |
| **Bitwise Disjunction** |

Bitwise disjunction consists of disjoining a bit from another bit. To support this operation, the C# language provides the bitwise disjunction operator represented with |.

To perform a bitwise conjunction on two numbers, remember that they must be converted to binary first. Then:

* If a bit with value 0 is added to a bit with value 0, the result is 0

|  |  |
| --- | --- |
| Bit0 | 0 |
| Bit1 | 0 |
| Bit0 Or Bit1 | 0 |

* If a bit with value 1 is added to a bit with value 0, the result is 1

|  |  |
| --- | --- |
| Bit0 | 1 |
| Bit1 | 0 |
| Bit0 Or Bit1 | 1 |

* If a bit with value 0 is added to a bit with value 1, the result is 1

|  |  |
| --- | --- |
| Bit0 | 0 |
| Bit1 | 1 |
| Bit0 Or Bit1 | 1 |

* If a bit with value 1 is added to a bit with value 1, the result is 1

|  |  |
| --- | --- |
| Bit0 | 1 |
| Bit1 | 1 |
| Bit0 Or Bit1 | 1 |

As an example, consider the number 305 bit-disjoined to 2853. The decimal number 305 converted to binary is 100110001. The decimal number 2853 converted to binary is 101100100101. Based on the above 4 points, we can disjoin these two numbers as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 305 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 2853 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 305 | 2853 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

Therefore, 305 | 2853 produces 101100110101 which is equivalent to:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bit11 | Bit10 | Bit9 | Bit8 | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|  | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 305 Or 2853 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
|  | 2048 | 0 | 512 | 256 | 0 | 0 | 32 | 16 | 0 | 4 | 0 | 1 |

This means that 286 | 4075 = 2048 + 512 + 256 + 32 + 16 + 4 + 1 = 2869

This can also be programmatically calculated as follows:

class Exercise

{

static void Main()

{

var number1 = 305;

var number2 = 2853;

var result = number1 | number2;

System.Console.Write("305 Or 2853 = ");

System.Console.WriteLine(result);

}

}

This would produce:

305 Or 2853 = 2869

Press any key to continue . . .

You can disjoint a number of bits on a variable and assign the result to the same variable. Here is an example:

class Exercise

{

static void Main()

{

var number = 305;

System.Console.WriteLine(number);

number = number | 22;

System.Console.WriteLine(number);

}

}

This would produce:

305

311

Press any key to continue . . .

A short version of this operation consists of using the |= operator. Here is an example:

class Exercise

{

static void Main()

{

var number = 305;

System.Console.WriteLine(number);

number |= 22;

System.Console.WriteLine(number);

}

}

|  |
| --- |
| **Bitwise Exclusion** |

Bitwise exclusion consists of adding two bits with the following rules. To support bitwise exclusion, the C# language provides the ^ operator:

* If both bits have the same value, the result is 0

|  |  |  |
| --- | --- | --- |
| Bit0 | 0 | 1 |
| Bit1 | 0 | 1 |
| Bit0 ^ Bit1 | 0 | 0 |

* If both bits have different values, the result is 1

|  |  |  |
| --- | --- | --- |
| Bit0 | 0 | 1 |
| Bit1 | 1 | 0 |
| Bit0 ^ Bit1 | 1 | 1 |

As an example, consider the number 618 bit-excluded from 2548. The decimal number 618 converted to binary is 1001101010. The decimal number 2548 converted to binary is 100111110100. Based on the above 2 points, we can bit-exclude these two numbers as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 618 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2548 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 618 ^ 2548 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

Therefore, 305 ^ 2853 produces 101110011110 which is equivalent to:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bit11 | Bit10 | Bit9 | Bit8 | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|  | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 618 ^ 2548 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
|  | 2048 | 0 | 512 | 256 | 128 | 0 | 0 | 16 | 8 | 4 | 2 | 0 |

This means that 286 ^ 4075 = 2048 + 512 + 256 + 128 + 16 + 8 + 4 + 2 = 2974

This can also be programmatically calculated as follows:

class Exercise

{

static void Main()

{

var number1 = 618;

var number2 = 2548;

var result = number1 ^ number2;

System.Console.WriteLine(result);

}

}

This would produce:

2974

Press any key to continue . . .

You can bitwise exclude a number of bits from a variable and assign the result to the variable itself. Here is an example:

class Exercise

{

static void Main()

{

var number = 618;

System.Console.WriteLine(number);

number = number ^ 38;

System.Console.WriteLine(number);

}

}

This would produce:

618

588

Press any key to continue . . .

As an alternative, you can use the ^= operator to get the same result. Here is an example:

class Exercise

{

static void Main()

{

var number = 618;

System.Console.WriteLine(number);

number ^= 38;

System.Console.WriteLine(number);

}

}

|  |
| --- |
| **Left-Shifting the Bits** |
|  |

Left shifting the bits consists of pushing each bit from right to left. You can do this by one ore more bits. Once again, to perform this operation, the number has to be converted to its binary equivalent. To support this operation, the C# language provides an operator represented as <<.

Imagine you have a number as 741. Its binary equivalent is 1011100101 and can be represented as:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

To perform a left shift operation on these bits, you push each from its position to the left, depending on the number of pushes you want. For example, to left-shift by 1 bit, you push each bit to the left by one position. Actually, you consider the bit at position x and the bit to its left at position y. You replace the value of Bit y by the value of Bit x. If Bit x is the most right bit, it receives a value of 0. This can be illustrated as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Original |  | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| << by 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |

As a result, we get 10111001010. The decimal result can be calculated as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Bit10 | Bit9 | Bit8 | Bit7 | Bit6 | Bit5 | Bit4 | Bit3 | Bit2 | Bit1 | Bit0 |
|  | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 741 << 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
|  | 1024 | 0 | 256 | 128 | 64 | 0 | 0 | 8 | 0 | 2 | 0 |

Consequently, 741 << 1 = 1024 + 256 + 128 + 64 + 8 + 2 = 1482. To programmatically perform this operation, use the << operator. Here is an example:

class Exercise

{

static void Main()

{

System.Console.WriteLine(741 << 1);

}

}

This would produce:

1482

Press any key to continue . . .

In the same way, you can push the bits to the left by more than one unit. Here is an example:

class Exercise

{

static void Main()

{

var number = 248 << 5;

System.Console.WriteLine(number);

}

}

This would produce:

7936

Press any key to continue . . .

You can push a few bits to the left of a variable and assign the result to the variable itself. Here is an example:

class Exercise

{

static void Main()

{

var number = 248;

System.Console.WriteLine(number);

number = number << 5;

System.Console.WriteLine(number);

}

}

This would produce:

248

7936

Press any key to continue . . .

To provide another version of this operation, the C# language is equipped with the <<= operator. Here is an example of using it:

class Exercise

{

static void Main()

{

var number = 248;

System.Console.WriteLine(number);

number <<= 5;

System.Console.WriteLine(number);

}

}

|  |
| --- |
| **Right-Shifting the Bits** |

You can shift the bits to the right. Everything is done as reviewed for the left shift but in reverse order. To support this operation, the C# language provides the >> operator.

**Questions**

1. **Two main data types in C#?**
2. **What are the conditional operators?**
3. **How to separate two statements in the code?**
4. **How to shift the bits?**
5. **What Binary operators do You know?**