Chapter 3. Operators and Expressions

In This Chapter

In this chapter we will get acquainted with the **operators in C#** and **the actions they can perform** when used with the different data types. In the beginning, we will explain which operators have higher priority and we will analyze the different types of operators, according to the number of the arguments they can take and the actions they perform. In the second part, we will examine **the conversion of data types**. We will explain when and why it is needed to be done and how to work with different data types. At the end of the chapter, we will pay special attention to the **expressions** and how we should work with them. Finally, we have prepared exercises to strengthen our knowledge of the material in this chapter.

Operators

Every programming language uses **operators**, through which we can perform different actions on the data. Let's take a look at the operators in C# and see what they are for and how they are used.

What Is an Operator?

After we have learned how to declare and set a variable in the previous chapter, we will discuss how to perform various operations with them. For this purpose we will get familiar with operators.

Operators allow processing of primitive data types and objects. They take as an input one or more operands and return some value as a result. Operators in C# are special characters (such as "**+**", "**.**", "**^**", etc.) and they perform transformations on one, two or three operands. Examples of operators in C# are the signs for adding, subtracting, multiplication and division from math (**+**, **-**, *****, **/**) and the operations they perform on the integers and the real numbers.

Operators in C#

Operators in C# can be separated in several different categories:

- **Arithmetic** operators – they are used to perform simple mathematical operations.

- **Assignment** operators – allow assigning values to variables.
- **Comparison** operators – allow comparison of two literals and/or variables.
- **Logical** operators – operators that work with Boolean data types and Boolean expressions.
- **Binary** operators – used to perform operations on the binary representation of numerical data.
- **Type conversion** operators – allow conversion of data from one type to another.

Operator Categories

Below is a list of the operators, separated into categories:

Category	Operators
arithmetic	`-`, `+`, `*`, `/`, `%`, `++`, `--`
logical	`&&`, `||`, `!`, `^`
binary	`&`, `|`, `^`, `~`, `<<`, `>>`
comparison	`==`,`!=`, `>`, `<`, `>=`, `<=`
assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
string concatenation	`+`
type conversion	`(type)`, `as`, `is`, `typeof`, `sizeof`
other	`.`, `new`, `()`, `[]`, `?:`, `??`

Types of Operators by Number of Arguments

Operators can be separated into different types according to the number of arguments they could take:

Operator type	Number of arguments (operands)
unary	takes one operand
binary	takes two operands
ternary	takes three operands

All **binary operators** in C# are **left-associative**, i.e. the expressions are calculated from left to right, except for the assignment operators. All assignment operators and conditional operators **`?:`** and **`??`** are right-associative, i.e. the expressions are calculated from right to left. The unary operators are not associative.

Some of the operators in C# perform different operations on the different data types. For example the operator **`+`**. When it is used on numeric data types (**int**, **long**, **float**, etc.), the operator performs mathematical addition. However, when we use it on strings, the operator concatenates (joins together) the content of the two variables/literals and returns the new string.

Operators – Example

Here is an example of using operators:

```
int a = 7 + 9;
Console.WriteLine(a); // 16

string firstName = "John";
string lastName = "Doe";

// Do not forget the space between them
string fullName = firstName + " " + lastName;
Console.WriteLine(fullName); // John Doe
```

The example shows how, as explained above, when the operator **+** is used on numbers it returns a numerical value, and when it is used on strings it returns concatenated strings.

Operator Precedence in C#

Some operators have **precedence** (priority) over others. For example, in math multiplication has precedence over addition. The operators with a higher precedence are calculated before those with lower. The operator **()** is used to **change the precedence** and like in math, it is calculated first.

The following table illustrates the precedence of the operators in C#:

Priority	Operators
Highest priority	**(**, **)**
	++, **--** (as postfix), **new**, **(type)**, **typeof**, **sizeof**
	++, **--** (as prefix), **+**, **-** (unary), **!**, **~**
	*****, **/**, **%**
	+ (string concatenation)
	+, **-**
…	**<<**, **>>**
	<, **>**, **<=**, **>=**, **is**, **as**
	==, **!=**
	&, **^**, **|**

Lowest priority	**&&**
	||
	?:, ??
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

The operators located **upper in the table have higher precedence** than those below them, and respectively they have an advantage in the calculation of an expression. To change the precedence of an operator we can use brackets.

When we write expressions that are more complex or have many operators, it is recommended to **use brackets** to avoid difficulties in reading and understanding the code. For example:

```
// Ambiguous
x + y / 100

// Unambiguous, recommended
x + (y / 100)
```

Arithmetical Operators

The arithmetical operators in C# **+**, **-**, ***** are the same like the ones in math. They perform addition, subtraction and multiplication on numerical values and the result is also a numerical value.

The **division operator /** has different effect on integer and real numbers. When we divide an integer by an integer (like **int**, **long** and **sbyte**) the returned value is an integer (no rounding, the fractional part is cut). Such division is called an **integer division**. Example of integer division: 7 **/** 3 = 2.

Integer division by 0 is not allowed and causes a runtime exception **DivideByZeroException**. The remainder of integer division of integers can be obtained by the operator **%**. For example, 7 **%** 3 = 1, and –10 **%** 2 = 0.

When dividing two real numbers or two numbers, one of which is real (e.g. **float**, **double**, etc.), a **real division** is done (not integer), and the result is a real number with a whole and a fractional part. For example: 5.0 / 2 = 2.5. In the division of real numbers it is **allowed to divide by 0.0** and respectively the result is **+∞** (**Infinity**), **-∞** (**-Infinity**) or **NaN** (invalid value).

The operator for **increasing by one** (increment) **++** adds one unit to the value of the variable, respectively the operator **--** (**decrement**) subtracts one unit from the value. When we use the operators **++** and **--** as a **prefix** (when we place them immediately before the variable), the new value is calculated first and then the result is returned. When we use the same operators as **post-fix** (meaning when we place them immediately after the variable) the original value of the operand is returned first, then the addition or subtraction is performed.

Arithmetical Operators – Example

Here are some examples of arithmetic operators and their effect:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
Console.WriteLine(squareSide); // 4.25
Console.WriteLine(squareArea); // 18.0625

int a = 5;
int b = 4;
Console.WriteLine(a + b);      // 9
Console.WriteLine(a + (b++));  // 9
Console.WriteLine(a + b);      // 10
Console.WriteLine(a + (++b));  // 11
Console.WriteLine(a + b);      // 11
Console.WriteLine(14 / a);     // 2
Console.WriteLine(14 % a);     // 4

int one = 1;
int zero = 0;
// Console.WriteLine(one / zero); // DivideByZeroException

double dMinusOne = -1.0;
double dZero = 0.0;
Console.WriteLine(dMinusOne / zero); // -Infinity
Console.WriteLine(one / dZero); // Infinity
```

Logical Operators

Logical (Boolean) operators take Boolean values and return a Boolean result (true or **false**). The basic Boolean operators are "**AND**" (&&), "**OR**" (||), "**exclusive OR**" (^) and **logical negation** (!).

The following table contains the logical operators in C# and the operations that they perform:

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true

false	**false**	**true**	**false**	**false**	**false**

The table and the following example show that the logical "AND" (**&&**) returns true only when both variables contain truth. Logical "OR" (**||**) returns true when at least one of the operands is true. The logical negation operator (**!**) changes the value of the argument. For example, if the operand has a value **true** and a negation operator is applied, the new value will be **false**. The negation operator is a unary operator and it is placed before the argument. Exclusive "OR" (**^**) returns **true** if only one of the two operands has the value **true**. If the two operands have different values, exclusive "OR" will return the result **true**, if they have the same values it will return **false**.

Logical Operators – Example

The following example illustrates the usage of the logical operators and their actions:

```
bool a = true;
bool b = false;
Console.WriteLine(a && b);              // False
Console.WriteLine(a || b);              // True
Console.WriteLine(!b);                  // True
Console.WriteLine(b || true);           // True
Console.WriteLine((5 > 7) ^ (a == b));  // False
```

Laws of De Morgan

Logical operations fall under the **laws of De Morgan** from the mathematical logic:

```
!(a && b) == (!a || !b)
!(a || b) == (!a && !b)
```

The first law states that the negation of the conjunction (logical AND) of two propositions is equal to the disjunction (logical OR) of their negations.

The second law states that the negation of the disjunction of both statements is equivalent to the conjunction of their negations.

Operator for Concatenation of Strings

The operator **+** is used to **join strings** (**string**). It concatenates (joins) two or more strings and returns the result as a new string. If at least one of the arguments in the expression is of type **string**, and there are other operands of type different from **string**, they will be automatically converted to type **string**, which allows successful string concatenation.

It is fantastic how .NET runtime handles such operation incompatibilities for us on the fly, saving us some coding time and allowing us to concentrate on the main objectives of our programming task! However, it is a good practice to not miss to cast the variables on which we wish to apply an operation; we should instead have them converted to the appropriate type for each operation, so that we are in full control of the end result and prevent implicit type casts. We will provide more detailed information on casting operations further down in the section "Type Conversion" of this chapter.

Operator for Concatenation of Strings – Example

Here is an example, which shows concatenations of two strings and a string with a number:

```
string csharp = "C#";
string dotnet = ".NET";
string csharpDotNet = csharp + dotnet;
Console.WriteLine(csharpDotNet); // C#.NET
string csharpDotNet4 = csharpDotNet + " " + 5;
Console.WriteLine(csharpDotNet4); // C#.NET 5
```

In the example we initialize two variables of type **string** and assign them values. On the third and fourth row we concatenate both strings and pass the results to the method **Console.WriteLine()** to print it on the console. On the next line we join the resulting string with a space and the number 5. We assign the returned value to the variable **csharpDotNet5**, which will automatically be converted to type **string**. On the last row we print the result.

Concatenation (joining, gluing) of strings is a slow operation and should be used carefully. It is recommended to use the StringBuilder class for iterative (repetitive) operations on strings.

In the chapter "Strings" we will explain in detail why the **StringBuilder class** must be used for join operations on strings performed in a loop.

Bitwise Operators

A bitwise operator is an operator that acts on the binary representation of numeric types. In computers all the data and particularly numerical data is represented as a series of ones and zeros. The **binary numeral system** is used for this purpose. For example, number **55** in the binary numeral system is represented as **00110111**.

Binary representation of data is convenient because zero and one in electronics can be implemented by Boolean circuits, in which zero is represented as "no electricity" or for example with a voltage of -5V and the one is presented as "have electricity" or say with voltage +5V.

We will examine in depth the **binary numeral system** in the chapter "Numeral Systems", but just for now we can consider that the numbers in computers are represented as ones and zeros, and bitwise operators are used to analyze and change those ones to zeros and vice versa.

Bitwise operators are very similar to the logical ones. In fact, we can imagine that the logical and bitwise operators perform the same thing but using different data types. Logical operators work with the values **true** and **false** (Boolean values), while bitwise operators work with numerical values and are applied bitwise over their binary representation, i.e., they work with the bits of the number (the digits **0** and **1** of which it consists). Just like the logical operators in C#, there are bitwise operators "AND" (**&**), bitwise "OR" (**|**), bitwise negation (**~**) and excluding "OR" (**^**).

Bitwise Operators and Their Performance

The bitwise operators' performance on binary digits **0** and **1** is shown in the following table:

x	y	~x	x & y	x | y	x ^ y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

As we see bitwise and logical operators are very much alike. The difference in the writing of "AND" and "OR" is that the logical operators are written with double ampersand (**&&**) and double vertical bar (**||**), and the bitwise – with a single ampersand or vertical bar (**&** and **|**). Bitwise and logical operators for exclusive "OR" are the same "**^**". For logical negation we use "**!**", while for bitwise negation (inversion) the "**~**" operator is used.

In programming there are two bitwise operators that have no analogue in logical operators. These are the **bit shift left** (**<<**) and **bit shift right** (**>>**). Used on numerical values, they move all the bits of the value to the left or right. The bits that fall outside the number are lost and replaced with 0.

The **bit shifting operators** are used in the following way: on the left side of the operator we place the variable (operand) with which we want to use the operator, on the right side we put a numerical value, indicating how many bits we want to offset. For example, **3 << 2** means that we want to move the bits of the number three, twice to the left. The number 3 presented in bits looks like this: "**0000 0011**". When you move twice left, the binary value will look like this: "**0000 1100**", and this sequence of bits is the number 12. If we look at the example we can see that actually we have multiplied the number by 4. Bit shifting itself can be represented as multiplication (bitwise shifting left) or division (bitwise shifting right) by a power of 2. This occurrence is due to the nature of the binary numeral system. Example of moving to the right is **6 >> 2**, which means to move the binary number "**0000 0110**" with two positions to the right. This means that we will lose two right-most digits and feed them with zeros on the left. The end result will be "**0000 0001**" which is 1.

Bitwise Operators – Example

Here is an example of using bitwise operators. The binary representation of the numbers and the results of the bitwise operators are shown in the comments (green text):

```
byte a = 3;                 // 0000 0011 = 3
byte b = 5;                 // 0000 0101 = 5

Console.WriteLine(a | b);   // 0000 0111 = 7
Console.WriteLine(a & b);   // 0000 0001 = 1
Console.WriteLine(a ^ b);   // 0000 0110 = 6
Console.WriteLine(~a & b);  // 0000 0100 = 4
Console.WriteLine(a << 1);  // 0000 0110 = 6
Console.WriteLine(a << 2);  // 0000 1100 = 12
Console.WriteLine(a >> 1);  // 0000 0001 = 1
```

In the example we first create and initialize the values of two variables **a** and **b**. Then we print on the console the results of some bitwise operations on the two variables. The first operation that we apply is "OR". The example shows that for all positions where there was 1 in the binary representation of the variables **a** and **b**, there is also 1 in the result. The second operation is "AND". The result of the operation contains 1 only in the right-most bit, because the only place where **a** and **b** have 1 at the same time is their right-most bit. Exclusive "OR" returns ones only in positions where **a** and **b** have different values in their binary bits. Finally, the logical negation and bitwise shifting: left and right, are illustrated.

Comparison Operators

Comparison operators in C# are used to compare two or more operands. C# supports the following comparison operators:

- greater than (**>**)
- less than (**<**)
- greater than or equal to (**>=**)
- less than or equal to (**<=**)
- equality (**==**)
- difference (**!=**)

All comparison operators in C# are **binary** (take two operands) and the returned result is a Boolean value (**true** or **false**). Comparison operators have lower priority than arithmetical operators but higher than the assignment operators.

Comparison Operators – Example

The following example demonstrates the usage of comparison operators in C#:

```
int x = 10, y = 5;
Console.WriteLine("x > y : " + (x > y));    // True
Console.WriteLine("x < y : " + (x < y));    // False
Console.WriteLine("x >= y : " + (x >= y));  // True
Console.WriteLine("x <= y : " + (x <= y));  // False
Console.WriteLine("x == y : " + (x == y));  // False
Console.WriteLine("x != y : " + (x != y));  // True
```

In the example, first we create two variables **x** and **y** and we assign them the values 10 and 5. On the next line we print on the console using the method **Console.WriteLine(…)** the result from comparing the two variables **x** and **y** using the operator **>**. The returned value is **true** because **x** has a greater value than **y**. Similarly, in the next rows the results from the other 5 comparison operators, used to compare the variables **x** and **y**, are printed.

Assignment Operators

The operator for **assigning value to a variable** is "**=**" (the character for mathematical equation). The syntax used for assigning value is as it follows:

```
operand1 = literal, expression or operand2;
```

Assignment Operators – Example

Here is an example to show the usage of the assignment operator:

```
int x = 6;
string helloString = "Hello string.";
int y = x;
```

In the example we assign value 6 to the variable x. On the second line we assign a text literal to the variable **helloString**, and on the third line we copy the value of the variable **x** to the variable **y**.

Cascade Assignment

The assignment operator can be used in **cascade** (more than once in the same expression). In this case assignments are carried out consecutively from right to left. Here's an example:

```
int x, y, z;
x = y = z = 25;
```

On the first line in the example we initialize three variables and on the second line we assign them the value 25.

The assignment operator in C# is "=", while the comparison operator is "==". The exchange of the two operators is a common error when we are writing code. Be careful not to confuse the comparison operator and the assignment operator as they look very similar.

Compound Assignment Operators

Except the assignment operator there are also **compound assignment operators**. They help to reduce the volume of the code by typing two operations together with an operator: operation and assignment. Compound operators have the following syntax:

```
operand1 operator = operand2;
```

The upper expression is like the following:

```
operand1 = operand1 operator operand2;
```

Here is an example of a compound operator for assignment:

```
int x = 2;
int y = 4;

x *= y; // Same as x = x * y;
Console.WriteLine(x); // 8
```

The most commonly used compound assignment operators are **+=** (adds value of **operand2** to **operand1**), **-=** (subtracts the value of the right operand from the value of the left one).Other compound assignment operators are ***=**, **/=** and **%=**.

The following example gives a good idea of how the compound assignment operators work:

```
int x = 6;
int y = 4;

Console.WriteLine(y *= 2);  // 8
int z = y = 3;              // y=3 and z=3

Console.WriteLine(z);       // 3
Console.WriteLine(x |= 1);  // 7
Console.WriteLine(x += 3);  // 10
Console.WriteLine(x /= 2);  // 5
```

In the example, first we create the variables **x** and **y** and assign them values 6 and 4. On the next line we print on the console **y**, after we have assigned it a new value using the operator ***=** and the literal 2.The result of the operation is 8. Further in the example we apply the other compound assignment operators and print the result on the console.

Conditional Operator ?:

The **conditional operator ?:** uses the Boolean value of an expression to determine which of two other expressions must be calculated and returned as a result. The operator works on three operands and that is why it is called ternary operator. The character "**?**" is placed between the first and second operand, and "**:**" is placed between the second and third operand. The first operand (or expression) must be **Boolean**, and the next two operands must be **of the same type**, such as numbers or strings.

The operator **?:** has the following syntax:

```
operand1 ? operand2 : operand3
```

It works like this: if **operand1** is set to **true**, the operator returns as a result **operand2**. Otherwise (if **operand1** is set to **false**), the operator returns as a result **operand3**.

During the execution, the value of the first argument is calculated. If it has value **true**, then the second (middle) argument is calculated and it is returned as a result. However, if the calculated result of the first argument is **false**, then the third (last) argument is calculated and it is returned as a result.

Conditional Operator "?:" – Example

The following example shows the usage of the operator "**?:**":

```
int a = 6;
int b = 4;
Console.WriteLine(a > b ? "a>b" : "b<=a"); // a>b
```

```
int num = a == b ? 1 : -1; // num will have value -1
```

Other Operators

So far we have examined arithmetic, logical and bitwise operators, the operator for concatenating strings, also the conditional operator **?:**. Besides them in C # there are several other operators worth mentioning.

The "." Operator

The **access operator** "**.**" (dot) is used to access the member fields or methods of a class or object. Example of usage of point operator:

```
Console.WriteLine(DateTime.Now); // Prints the date + time
```

Square Brackets [] Operator

Square brackets **[]** are used to **access elements of an array by index**, they are the so-called **indexer**. Indexers are also used for accessing characters in a string. Example:

```
int[] arr = { 1, 2, 3 };
Console.WriteLine(arr[0]); // 1
string str = "Hello";
Console.WriteLine(str[1]); // e
```

Brackets () Operator

Brackets **()** are used to **override the priority of execution** of expressions and operators. We have already seen how the brackets work.

Type Conversion Operator

The operator for type conversion **(type)** is used to **convert** a variable from one type to another. We will examine it in details in the section "Type Conversion".

Operator "as"

The operator **as** also is used for **type conversion** but invalid conversion returns null, not an exception.

Operator "new"

The **new** operator is used to **create and initialize new objects**. We will examine it in details in the chapter "Creating and Using Objects".

Operator "is"

The **is** operator is used to check whether an object is compatible with a given type (**check object's type**).

Operator "??"

The operator **??** is similar to the conditional operator **?:**. The difference is that it is placed between two operands and returns the left operand only if its value is not null, otherwise it returns the right operand. Example:

```
int? a = 5;
Console.WriteLine(a ?? -1); // 5
string name = null;
Console.WriteLine(name ?? "(no name)"); // (no name)
```

Other Operators – Examples

Here is an example that shows the operators we just explained:

```
int a = 6;
int b = 3;

Console.WriteLine(a + b / 2);                          // 7
Console.WriteLine((a + b) / 2);                        // 4

string s = "Beer";
Console.WriteLine(s is string);                        // True

string notNullString = s;
string nullString = null;
Console.WriteLine(nullString ?? "Unspecified");  // Unspecified
Console.WriteLine(notNullString ?? "Specified"); // Beer
```

Type Conversion and Casting

Generally, operators work over arguments with the same data type. However, C# has a wide variety of data types from which we can choose the most appropriate for a particular purpose. To perform an operation on variables of two different data types we need to convert both to the same data type. **Type conversion** (**typecasting**) can be **explicit** and **implicit**.

All expressions in C# have a type. This type can derive from the expression structure and the types, variables and literals used in it. It is possible to write an expression which type is inappropriate for the current context. In some cases this will lead to a compilation error, but in other cases the context can get a type that is similar or related to the type of the expression. In this case the program performs a **hidden type conversion**.

Specific conversion from type **S** to type **T** allows the expression of type **S** to be treated as an expression of type **T** during the execution of the program. In some cases this will require a validation of the transformation. Here are some examples:

- Conversion of type **object** to type **string** will require verification at runtime to ensure that the value is really an instance of type **string**.
- Conversion from **string** to **object** does not require any verification. The type **string** is an inheritor of the type **object** and can be converted to its base class without a risk of an error or data loss. We shall examine inheritance in details in the chapter "Object-Oriented Programming Principles".
- Conversion of type **int** to **long** can be made without verification during the execution, because there is no risk of data loss since the set of values of type **int** is a subset of values of type **long**.
- Conversion from type **double** to **long** requires conversion of 64-bit floating-point value to 64-bit integer. Depending on the value, data loss is possible and therefore it is necessary to convert the types explicitly.

In C# not all types can be converted to all other types, but only to some of them. For convenience, we shall group some of the possible transformations in C# according to their type into three categories:

- implicit conversion;
- explicit conversion;
- conversion to or from **string**;

Implicit Type Conversion

Implicit (hidden) type conversion is possible only when there is no risk of data loss during the conversion, i.e. when converting from a lower range type to a larger range (e.g. from **int** to **long**). To make an implicit conversion it is not necessary to use any operator and therefore such transformation is called implicit. The **implicit conversion** is done automatically by the compiler when you assign a value with lower range to a variable with larger range or if the expression has several types with different ranges. In such case the conversion is executed into the type with the highest range.

Implicit Type Conversion – Examples

Here is an example of implicit type conversion:

```
int myInt = 5;
Console.WriteLine(myInt); // 5

long myLong = myInt;
Console.WriteLine(myLong); // 5
Console.WriteLine(myLong + myInt); // 10
```

In the example we create a variable **myInt** of type **int** and assign it the value 5. After that we create a variable **myLong** of type **long** and assign it the value contained in **myInt**. The value stored in **myLong** is automatically converted from type **int** to type **long**. Finally, we output the result from adding the two variables. Because the variables are from different types they are automatically converted to the type with the greater range, i.e. to type **long** and the result that is printed on the console is **long** again. Indeed, the given parameter to the method **Console.WriteLine()** is of type **long**, but inside the method it will be converted again, this time to type **string**, so it can be printed on the console. This transformation is performed by the method **Long.ToString()**.

Possible Implicit Conversions

Here are some possible implicit conversions of primitive data types in C#:

- **sbyte** → **short**, **int**, **long**, **float**, **double**, **decimal**;
- **byte** → **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**;
- **short** → **int**, **long**, **float**, **double**, **decimal**;
- **ushort** → **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**;
- **char** → **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal** (although **char** is a character type in some cases it may be regarded as a number and have a numeric type of behavior, it can even participate in numeric expressions);
- **uint** → **long**, **ulong**, **float**, **double**, **decimal**;
- **int** → **long**, **float**, **double**, **decimal**;
- **long** → **float**, **double**, **decimal**;
- **ulong** → **float**, **double**, **decimal**;
- **float** → **double**.

There is **no data loss when converting types of smaller range to types with a larger range**. The numerical value remains the same after conversion. There are a few exceptions. When you convert type **int** to type **float** (32-bit values), the difference is that **int** uses all bits for a whole number, whereas **float** has a part of bits used for representation of a fractional part. Hence, loss of precision is possible because of rounding when conversion from **int** to **float** is made. The same applies for the conversion of 64-bit **long** to 64-bit **double**.

Explicit Type Conversion

Explicit type conversion is used whenever there is a possibility of data loss. When converting floating point type to integer type there is always a loss of data coming from the elimination of the fractional part and an **explicit conversion** is obligatory (e.g. **double** to **long**). To make such a conversion it is necessary to use the operator for data conversion (**type**). There may also be data loss when converting a type with a wider range to type with a narrower one (**double** to **float** or **long** to **int**).

Explicit Type Conversion – Example

The following example illustrates the use of explicit type conversion and data loss that may occur in some cases:

```
double myDouble = 5.1d;
Console.WriteLine(myDouble); // 5.1

long myLong = (long)myDouble;
Console.WriteLine(myLong); // 5

myDouble = 5e9d; // 5 * 10^9
Console.WriteLine(myDouble); // 5000000000

int myInt = (int)myDouble;
Console.WriteLine(myInt); // -2147483648
Console.WriteLine(int.MinValue); // -2147483648
```

In the first line of the example we assign a value 5.1 to the variable **myDouble**. After we convert (explicitly) to type **long** using the operator **(long)** and print on the console the variable **myLong** we see that the variable has lost its fractional part, because **long** is an integer. Then we assign to the real double precision variable **myDouble** the value 5 billion. Finally, we convert **myDouble** to **int** by the operator **(int)** and print variable **myInt**. The result is the same like when we print **int.MinValue** because **myDouble** contains a value bigger than the range of **int**.

It is not always possible to predict what the value of a variable will be after its scope overflows! Therefore, use sufficiently large types and be careful when switching to a "smaller" type.

Data Loss during Type Conversion

We will give an example for data loss during type conversion:

```
long myLong = long.MaxValue;
```

```
int myInt = (int)myLong;

Console.WriteLine(myLong); // 9223372036854775807
Console.WriteLine(myInt); // -1
```

The type conversion operator may also be used in case of an intentional implicit conversion. This contributes to the readability of code, reducing the chance for errors and it is considered good practice by many programmers.

Here are some more examples for type conversions:

```
float heightInMeters = 1.74f; // Explicit conversion
double maxHeight = heightInMeters; // Implicit
double minHeight = (double)heightInMeters; // Explicit
float actualHeight = (float)maxHeight; // Explicit

float maxHeightFloat = maxHeight; // Compilation error!
```

In the example above at the last line we have an expression that will generate a compilation error. This is because we try implicitly to convert type **double** to **float**, which can cause data loss. C# is a strongly typed programming language and does not allow such appropriation of values.

Forcing Overflow Exceptions during Casting

Sometimes it is convenient, instead of getting the wrong result, when a type overflows during switching from larger to smaller type, to get notification of the problem. This is done by the keyword **checked** which includes a **check for overflow in integer types**:

```
double d = 5e9d; // 5 * 10^9
Console.WriteLine(d); // 5000000000
int i = checked((int)d); // System.OverflowException
Console.WriteLine(i);
```

During the execution of the code fragment above an exception (i.e. notification of an error) of type **OverflowException** is raised. More information about the exceptions and the methods to catch and handle them can be found in the chapter "Exception Handling".

Possible Explicit Conversions

The explicit conversions between numeral types in C# are possible between any couple among the following types:

sbyte, **byte**, **short**, **ushort**, **char**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**

In these conversions data can be lost, like data about the number size or information about its precision.

Notice that conversion to or from **`string`** is not possible through typecasting.

Conversion to String

If it is necessary we can convert any type of data, including the value **`null`**, to string. The conversion of strings is done automatically whenever you use the concatenation operator (**+**) and one of the arguments is not of type string. In this case the argument is **converted to a string** and the operator returns a new string representing the concatenation of the two strings.

Another way to convert different objects to type string is to call the method **`ToString()`** of the variable or the value. It is valid for all data types in .NET Framework. Even calling **`3.ToString()`** is fully valid in C# and the result will return the string **`"3"`**.

Conversion to String – Example

Let's take a look on several examples for converting different data types to string:

```
int a = 5;
int b = 7;

string sum = "Sum = " + (a + b);
Console.WriteLine(sum);

String incorrect = "Sum = " + a + b;
Console.WriteLine(incorrect);

Console.WriteLine(
  "Perimeter = " + 2 * (a + b) + ". Area = " + (a * b) + ".");
```

The result from the example is as follows:

```
Sum = 12
Sum = 57
Perimeter = 24. Area = 35.
```

From the results it is obvious, that concatenating a number to a character string returns in result the string followed by the text representation of the number. Note that the "**+**" for concatenating strings can cause **unpleasant effects** on the addition of numbers, because it has equal priority with the operator "**+**" for mathematical addition. Unless the priorities of the operations are changed by placing the brackets, they will always be executed from left to right.

More details about converting from and to **string** we will look at the chapter "Console Input and Output".

Expressions

Much of the program’s work is the calculation of expressions. **Expressions are sequences of operators, literals and variables** that are calculated to a value of some type (number, string, object or other type). Here are some examples of expressions:

```
int r = (150-20) / 2 + 5;

// Expression for calculating the surface of the circle
double surface = Math.PI * r * r;

// Expression for calculating the perimeter of the circle
double perimeter = 2 * Math.PI * r;

Console.WriteLine(r);
Console.WriteLine(surface);
Console.WriteLine(perimeter);
```

In the example three expressions are defined. The first expression calculates the radius of a circle. The second calculates the area of a circle, and the last one finds the perimeter. Here is the result from the fragment above:

```
70
15393.80400259
439.822971502571
```

Side Effects of Expressions

The calculation of the expression can have **side effects**, because the expression can contain embedded assignment operators, can cause increasing or decreasing of the value and calling methods. Here is an example of such a side effect:

```
int a = 5;
int b = ++a;

Console.WriteLine(a); // 6
Console.WriteLine(b); // 6
```

Expressions, Data Types and Operator Priorities

When writing expressions, the data types and the behavior of the used operators should be considered. Ignoring this can lead to unexpected results. Here are some simple examples:

```
// First example
double d = 1 / 2;
Console.WriteLine(d); // 0, not 0.5

// Second example
double half = (double)1 / 2;
Console.WriteLine(half); // 0.5
```

In the first example, an expression divides two integers (written this way, 1 and two are integers) and assigns the result to a variable of type **double**. The result may be unexpected for some people, but that is because they are ignoring the fact that in this case the operator "**/**" works over integers and the result is an integer obtained by cutting the fractional part.

The second example shows that if we want to do division with fractions in the result, it is necessary to convert to **float** or **double** at least one of the operands. In this scenario the division is no longer integer and the result is correct.

Division by Zero

Another interesting example is **division by 0**. Most programmers think that division by 0 is an invalid operation and causes an error at runtime (exception) but this is actually true only for integer division by 0. Here is an example, which shows that fractional division by **0** is **Infinity** or **NaN**:

```
int num = 1;
double denum = 0; // The value is 0.0 (real number)
int zeroInt = (int) denum; // The value is 0 (integer number)
Console.WriteLine(num / denum); // Infinity
Console.WriteLine(denum / denum); // NaN
Console.WriteLine(zeroInt / zeroInt); // DivideByZeroException
```

Using Brackets to Make the Code Clear

When working with expressions it is important to **use brackets** whenever there is the slightest doubt about the priorities of the operations. Here is an example that shows how useful the brackets are:

```
double incorrect = (double)((1 + 2) / 4);
Console.WriteLine(incorrect); // 0
```

```
double correct = ((double)(1 + 2)) / 4;
Console.WriteLine(correct); // 0.75

Console.WriteLine("2 + 3 = " + 2 + 3); // 2 + 3 = 23
Console.WriteLine("2 + 3 = " + (2 + 3)); // 2 + 3 = 5
```

Exercises

1. Write an expression that checks whether an integer is **odd or even**.
2. Write a Boolean expression that checks whether a given integer is **divisible by both 5 and 7**, without a remainder.
3. Write an expression that looks for a given integer if its **third digit** (right to left) is 7.
4. Write an expression that checks whether the **third bit** in a given integer is 1 or 0.
5. Write an expression that calculates the **area of a trapezoid** by given sides **a**, **b** and height **h**.
6. Write a program that prints on the console the **perimeter and the area of a rectangle** by given side and height entered by the user.
7. The gravitational field of the Moon is approximately 17% of that on the Earth. Write a program that calculates the **weight of a man on the moon** by a given weight on the Earth.
8. Write an expression that checks for a given point {x, y} if it is **within the circle** K({0, 0}, R=5). Explanation: the point {0, 0} is the center of the circle and 5 is the radius.
9. Write an expression that checks for given point {x, y} if it is **within the circle** K({0, 0}, R=5) and **out of the rectangle** [{-1, 1}, {5, 5}]. Clarification: for the rectangle the lower left and the upper right corners are given.
10. Write a program that takes as input a **four-digit number** in format **abcd** (e.g. 2011) and performs the following actions:
    - Calculates the sum of the digits (in our example 2+0+1+1 = 4).
    - Prints on the console the number in reversed order: **dcba** (in our example 1102).
    - Puts the last digit in the first position: **dabc** (in our example 1201).
    - Exchanges the second and the third digits: **acbd** (in our example 2101).

11. We are given a number **n** and a position **p**. Write a sequence of operations that prints the value of **the bit on the position p** in the number (0 or 1). Example: **n**=35, **p**=5 -> 1. Another example: n=35, **p**=6 -> 0.
12. Write a Boolean expression that checks if the bit on position **p** in the integer **v** has the value 1. Example v=5, **p**=1 -> **false**.
13. We are given the number **n**, the value **v** (**v** = 0 or 1) and the position **p**. write a sequence of operations that changes the value of **n**, so the bit on the position **p** has the value of **v**. Example: n=35, p=5, v=0 -> n=3. Another example: n=35, p=2, v=1 -> n=39.
14. Write a program that checks if a given number **n** (1 < **n** < 100) is a **prime number** (i.e. it is divisible without remainder only to itself and 1).
15. * Write a program that **exchanges the values of the bits** on positions 3, 4 and 5 with bits on positions 24, 25 and 26 of a given 32-bit unsigned integer.
16. * Write a program that **exchanges bits** {p, p+1, …, p+k-1} with bits {q, q+1, …, q+k-1} of a given 32-bit unsigned integer.

Solutions and Guidelines

1. Take the **remainder of dividing the number by 2** and check if it is **0** or **1** (respectively the number is odd or even). **Use % operator** to calculate the remainder of integer division.
2. Use a logical "AND" (**&&** operator) and the remainder operation **%** in division. You can also solve the problem by only one test: the division of 35 (think why).
3. Divide the number by 100 and save it in a new variable, which then divide by 10 and take the remainder. The remainder of the division by 10 is the third digit of the original number. Check if it is equal to 7.
4. Use **bitwise "AND"** on the current number and the number that has 1 only in the third bit (i.e. number 8, if bits start counting from 0). If the returned result is different from 0 the third bit is 1:

```
int num = 25;
bool bit3 = (num & 8) != 0;
```

5. The formula for **trapezoid surface** is: **S = (a + b) * h / 2**.
6. Search the Internet for **how to read integers** from the console and use the formula for rectangle area calculation. If you have difficulties see instructions on the next problem.
7. Use the following code to **read the number from the console**:

```
Console.Write("Enter number: ");
int number = Convert.ToInt32(Console.ReadLine());
```

Then **multiply by 0.17** and print it.

8. Use the **Pythagorean Theorem a^2 + b^2 = c^2**. The point is inside the circle when **(x*x) + (y*y) ≤ 5*5**.

9. Use the code from the previous task and **add a check for the rectangle**. A point is inside a rectangle with walls parallel to the axes, when in the same time it is right of the left wall, left of the right wall, down from the top wall and above the bottom wall.

10. To get the individual **digits of the number** you can divide by 10 and take the remainder of the division by 10:

```
int a = num % 10;
int b = (num / 10) % 10;
int c = (num / 100) % 10;
int d = (num / 1000) % 10;
```

11. Use **bitwise operations**:

```
int n = 35; // 00100011
int p = 6;
int i = 1; // 00000001
int mask = i << p; // Move the 1-st bit left by p positions

// If i & mask are positive then the p-th bit of n is 1
Console.WriteLine((n & mask) != 0 ? 1 : 0);
```

12. The task is similar to the previous one.

13. Use bitwise operations by analogy with the previous two problems. You can reset the bit at position **p** in the number **n** as follows:

```
n = n & (~(1 << p));
```

You can set bits in the **unit** at position **p** in the number **n** as follows:

```
n = n | (1 << p);
```

Think how you can combine the above two hints.

14. Read about **loops** in the Internet or in the chapter "Loops". Use a loop and check the number for divisibility by all integers from 1 to the square root of the number. Since **n < 100**, you can find in advance all prime numbers from 1 to 100 and checks the input over them. The **prime**

numbers in the range [1…100] are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 and 97.

15. Use 3 times a combination of **getting and setting a bit at a given position**. The first exchange is given below:

```
int bit3 = (num >> 3) & 1;
int bit24 = (num >> 24) & 1;
num = num & (~(1 << 24)) | (bit3 << 24);
num = num & (~(1 << 3)) | (bit24 << 3);
```

16. Extend the solution of the previous problem to perform a **sequence of bit exchanges in a loop**. Read about loops in the chapter "Loops".