### Expressions

In this chapter you take a detailed look at expressions in C#. Expressions are at the core of all coding that you will do, so we will take quite some time to work through the different kinds of expressions allowed in C#.

Any valid sequence of operators and operands is called an expression. Expressions have a specific order for evaluating of the operands and operators. Also, different expressions will have different meanings.

##### Classifications of Expressions

There are a number of different classifications of expressions. Each expression falls into one category:

* + **Value.** Every value will have an associated type.
  + **Variable.** Every variable will have an associated type, the declared type of the variable.
  + **Namespace.** Expressions with the namespace classification can only appear on the left- hand side for a member access.
  + **Type.** Expressions with the type classification can only appear on the left-hand side for a member access.
  + **Method group.** These are overloaded methods that result from member lookup.
  + **Anonymous method.** These are expressions used on a delegate creation expression or implicitly converted to a compatible delegate type.
  + **Property access.** Every property access used has an associated type, which will be the type of the property.
  + **Event access.** Every event access used has an associated type, which will be the type of the event.
  + **Indexer access.** Every indexer access used has an associated type, which will be the ele- ment type of the indexer.

*When an expression is an invocation of any method with a return type of* void*, the expression is classi- fied as having no classification at all — a nothing.*

Results of an Expression

The result of an expression cannot be any of the following:

* Anonymous method
* Event access
* Method group
* Namespace
* Type

Instead, these categories are merely intermediates used in specific contexts.

##### Expression Values

Most expressions invariably end up with a value. Since this is the case, if the expression denotes a namespace, a nothing, or a type, a compiler error is generated.

If an expression denotes a variable, indexer, or property access, the value will be implicitly and automat- ically substituted. Let’s take a look at the rules that encompass this:

* **Variables.** Values of variables will be the value stored in the variable.
* **Indexers.** This value is obtained by invoking the get-accessor of the indexer. If no get- accessor exists, a compiler-time error results.
* **Property access.** This value is obtained by invoking the get-accessor of the property access. If no get-accessor exists, a compiler-time error results.

##### Expressions and Operators

All expressions are made up of operands and operators. Operands are the inputs to the operators, and the operators are used to indicate what operations should be applied to the operands. The following table provides an example.

|  |  |  |
| --- | --- | --- |
| **operand** | **operator** | **operand** |
| 2 | + | 5 |

The commonest types of operators are mathematical operators such as +, -, \*, and /. The commonest types of operands in C# include variables, constants, and expressions.

Three Kinds of Operator

There are three kinds of operators:

* + **Unary.** Takes one operand and uses either prefix (-x) or postfix (x++) notation
  + **Binary.** Takes two operands and all use infix notation (that is, they go between the operands)

x + y y - z

* + **Ternary.** There is only one ternary operator, ?:. This takes three operands, and it uses infix notation.

(z ? x : y)

This is a handy shorthand way of saying:

condition ? value if true : value if false

In expressions, the order of evaluation is controlled by both the precedence and associativity of the oper- ators (discussed in more detail in the following section).

Operands are processed left to right:

4 + 4 + 3

4 + 4 = 8 + 3 = 11

This order can be overridden using parentheses:

4 + (4 \* 3)

(4 \* 3) = 12 + 4 = 16

Note that this expression:

4 + 4 \* 3

Is the same as:

4 + (4 \* 3)

Operator Precedence and Associativity

Expressions that contain more than one operator rely on operator precedence to control the order in which the operators are evaluated.

Here is a table that lists the operator precedence for all operators, from high to low:

**Category Operators**

Primary

Unary

Multiplicative

Additive Shift

Relational and type-testing

Equality

Logical AND Logical XOR Logical OR Conditional AND Conditional OR Null Coalesing

x.y f(x)

a[x] x++ x-- new

typeof checked unchecked

+

-

!

~

++x (T)x

\*

/

%

+

-

<<

>>

<

>

<=

>=

is as

==

!=

&

^

| &&

||

??

**Category Operators**

Assignment ?:

\*=

/=

%=

+=

-=

<<=

>>=

&=

^=

|=

When operands are between two operators and these two operators have the same precedence value, associativity is used to control the order of processing.

These are the rules of associativity:

* + Apart from assignment and null coalescing operators, all other binary operators are left associa- tive. That means that operations are carried out left to right.
  + Assignment, null coalescing, and the single ternary operator (the conditional operator) are right associative. This means that operations are carried out right to left.

Operator Overloading

All unary and binary operators have a predefined set of implementations available by default (that is, the + operator can carry out addition, the – subtraction, and so on) in any expression they are used in. To augment these predefined implementations, user-defined implementations can be introduced by includ- ing operator declarations in classes and structs.

User-defined operator implementations always take precedence over predefined operator implementa- tions. Only when there is no applicable user-defined operator implementation are predefined operator implementations used.

Overloadable unary operators are:

❑ +

* + -
  + !
  + ~

❑ ++

* + --
  + true
  + false

Overloadable binary operators are:

❑ +

* -
* \*
* /
* %
* &
* |
* ^
* <<
* >>

❑ ==

* !=
* >
* <
* >=
* <=

It is important to note that when any binary operator is overloaded, the associated assignment operator, if it exists, is implicitly overloaded.

In expressions, operators are referenced using operator notation, but in declarations, operators are refer- enced using functional notation. The following table shows the relationship between operator and func- tional notations for unary and binary operators.

**Operator notation Functional notation**

op x

Here op denotes any overloadable unary prefix operator.

x op

Here op denotes the unary postfix ++ and – operators.

x op y

Here op denotes any overloadable binary operator.

operator op(x)

operator op(x)

operator op(x,y)

User-defined operator declarations require one or more of the parameters to be of the class or struct type that contains the operator declaration.

User-defined operator declarations cannot modify any of the following aspects of an operator:

* Associativity
* Precedence
* Syntax

Unary Operator Overload Resolution

An operation that takes on the form op x or x op (where op is an overloadable unary operator and x is an expression of type X) is processed using the follow rules:

* The set of candidate user-defined operators provided by X for the operation operator op(x) is determined using the following rules:

Given a type T and an operation operator op(A), where op is an overloadable operator and A is an argument list, the set of candidate user-defined operators provided by T for operator op(A) is determined as follows:

* + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If the set of candidate user-defined operators is not empty, these are then set as the candidate operators for the operation. Otherwise, the predefined unary operator op implementations become the candidate operators for the operation. If type X is not an enum type, any predefined unary operator with a parameter type that is an enum type is not considered.
* The following overload resolution rules are applied to the set of candidate operators to select the most appropriate operator with regard to the argument list (x). This operator becomes the result of the overload resolution process. Given a type T and an operation operator op(A), where op is an overloadable operator and A is an argument list, the set of candidate user- defined operators provided by T for operator op(A) is determined as follows:
  + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If overload resolution fails to select a best operator, a compiler error is generated.

Binary Operator Overload Resolution

An operation of the form x op y, where op is an overloadable binary operator, x is an expression of type X, and y is an expression of type Y, will be processed according to the following rules:

* The set of candidate user-defined operators provided by X and Y for the operation operator op(x, y) are determined. The set consists of the union of the candidate operators provided by X and the candidate operators provided by Y, each determined using the rules which follow:
  + Determine the type T0 that results from removing the trailing ? modifiers, if any, from T.
  + For all operator op declarations in T0, if at least one operator is applicable with respect to the argument list A, the set of candidate operators consists of all applicable operator op declarations in T0. The lifted forms of the operators declared in T0 are considered also to be declared by T0.
  + Alternatively, if T0 is an object, the set of candidate operators is empty.
  + Alternatively, the set of candidate operators provided by T0 is the set of candidate opera- tors provided by the direct base class of T0.
* If the set of candidate user-defined operators is not empty, this is set as the candidate operators for the operation. If it is empty, the predefined binary operator op implementations become the set of candidate operators for the operation.
* The overload resolution rules (listed above) are applied to the set of candidate operators to select the best operator with respect to the argument list (x, y), and this operator becomes the result of the overload resolution process.
* If overload resolution fails to select a best operator, a compiler error is generated.

Lifted Operators

Lifted operators allow predefined and user-defined operators that operate on non-nullable value types to be used with nullable forms of those types. Lifted operators are formed from predefined and user-defined operators. These operators, however, do have to meet certain requirements, discussed as follows.

Unary Operators

The unary operators are:

❑ +

❑ ++

* -
* --
* !
* ~

An operator exists in a lifted form if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single ? modifier to the operand and result types (for example, !?).

The lifted operator produces a null value when the operand is null.

Equality Operators

The equality operators are:

❑ ==

* + !=

For equality operators, a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is bool.

The lifted forms are created by adding a single ? modifier to each operand type.

Relational Operators

The relational operators are:

* + <
  + >
  + <=
  + >=

The lifted form of a relational operator exists if the operand types are both non-nullable value types and if the result type is bool.

The lifted form is constructed by adding a single ? modifier to each operand type. The lifted operator produces the value false if one or both operands are null.

##### Member Lookup

A member lookup happens when the meaning of a name in the context of the type must be determined. A member lookup can happen as part of evaluating a simple-name or a member-access in an expression.

Member lookup takes into account not only the name of a member but also the number of type parame- ters the member has. It also looks at whether the member is accessible. For the purposes of member lookup, both generic methods and nested generic types have the number of type parameters that are indicated in their respective declarations and all other members will have zero type parameters.

A member lookup of a name N with K type parameters in a type T is processed in the following way:

* + The set of accessible members named N is worked out:
    - If T is a type parameter, then the set is the union of the sets of accessible members named N in each of the types specified as a primary constraint or secondary constraint for T, com- bined with the set of accessible members named N in object.
    - Alternatively, the set consists of all accessible members named N in T (which includes inherited members and the accessible members named N in object). If T is a constructed type, the set of members is obtained by substituting type arguments. Members that include an override modifier are excluded from the set.
* If the set of accessible members is empty, the member lookup does not produce a match, and no further steps are made.
* If K is zero, all nested types whose declaration included type parameters are removed. If K is not zero, all members with a different number of type parameters are removed.
* The members hidden by other members are also removed from the set. For every member S.M in the set, where S is the type in which the member M is declared, the following set of rules is applied:
  + If M is a constant, enumeration member, event, field, property, or type declaration, all members declared in a base type of S will be removed from the set.
  + If M is a method, all nonmethod members declared in a base type of S are removed.
* The interface members hidden by class members are next removed from the set. For every mem- ber S.M in the set, where S is the type in which the member M is declared, the following rules are applied if S is a class declaration other than object:
  + If M is a constant, event, enumeration member, field, property, or type declaration, all members declared in the interface declaration will be removed from the set.
  + If M is a method, all nonmethod members declared in an interface declaration are removed.
* Finally, the result of the lookup is determined:
  + If the set is made up of a single member that is not a method, this member will become the result of the lookup.
  + If the set contains nothing but methods, the group of methods is the result of the lookup.
  + Otherwise, the lookup is ambiguous, and a compiler error is generated.

Base Types

For member lookups, a type T will have the following base types:

* If T is object, T has no base type.
* If T is an enum type, the base types of T are the class types System.Enum, System.ValueType, and object.
* If T is a struct type, the base types of T are the class types System.ValueType and object.
* If T is a class type, the base types of T are the base classes of T, including the class type object.
* If T is an interface type, the base types of T are the base interfaces of T and the class type object.
* If T is an array type, the base types of T are the class types System.Array and object.
* If T is a delegate type, the base types of T are the class types System.Delegate and object.
* If T is a nullable type, the base types of T are the class types System.ValueType and object.

##### Function Members

Function members contain executable statements, are always members of types, and cannot be members of namespaces.

C# defines the following categories of function members:

* + Methods
  + Properties
  + Events
  + Indexers
  + User-defined operators
  + Instance constructors
  + Static constructors
  + Finalizers

Following are tables that summarize the processing that takes place in constructs involving each of the six categories of function members that can be explicitly invoked.

*Note that* e*,* x*,* y*, and* value *indicate expressions classified as variables or values,* T *indicates an expression classified as a type,* F *is the simple name of a method, and* P *is the simple name of a property.*

**Example Description**

**Method Invocation**

The method is invoked with the instance expression e and the argument list (x, y).

F(x,y)

Overload resolution is used to select the best method F in the con- taining class or struct.

The method is invoked with the argument list (x, y).

If the method is not static, the instance expression is this.

T.F(x,y)

Overload resolution is used to select the best method F in the class or struct T.

A compiler error is generated if the method is not static.

e. F(x,y)

The method is invoked with the argument list (x, y).

Overload resolution is used to select the best method F in the class, struct, or interface given by the type of e.

A compiler error is generated if the method is static.

**Example Description**

**Property Access**

P The get accessor of the property P in the containing class or struct is invoked.

A compiler error is generated if P is write-only.

P=value

If P is not static, the instance expression is this.

The set accessor of the property P in the containing class or struct is invoked with the argument list (value).

A compiler error is generated if P is read-only.

If P is not static, the instance expression is this.

T.P The get accessor of the property P in the class or struct T is invoked.

T.P=value

A compiler error is generated if P is not static or if P is write-only.

The set accessor of the property P in the class or struct T is invoked with the argument list (value).

A compile-time error occurs if P is not static or if P is read-only.

e.P The get accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e.

e.P=value

**Event Access**

A compiler error is generated if P is static or if P is write-only.

The set accessor of the property P in the class, struct, or interface given by the type of e is invoked with the instance expression e and the argument list (value).

A compiler error is generated if P is static or if P is read-only.

E +=value

The add accessor of the event E in the containing class or struct is invoked.

If E is not static, the instance expression is this.

E -= value

The remove accessor of the event E in the containing class or struct is invoked.

If E is not static, the instance expression is this.

**Example Description**

T.E+=value

The add accessor of the event E in the class or struct T is invoked.

T.E-=value

A compiler error is generated if E is not static.

The get accessor of the event E in the class or struct T is invoked.

e.E+=value

A compiler error is generated if E is not static.

The add accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e.

e.E-=value

A compiler error is generated if E is static.

The remove accessor of the event E in the class, struct, or interface given by the type of e is invoked with the instance expression e.

**Indexer Access**

A compile-time error occurs if E is static.

e[x, y]

Overload resolution is used to select the most appropriate indexer in the class, struct, or interface given by the type of e.

The get accessor of the indexer is invoked with the instance expression e and the argument list (x, y).

e[x, y]=value

A compiler error is generated if the indexer is set to write-only.

Overload resolution is used to select the most appropriate indexer in the class, struct, or interface given by the type of e.

The set accessor of the indexer is invoked with the instance expression e and the argument list (x, y, value).

A compiler error is generated if the indexer is read-only.

**Operator Invocation**

-x

x+y

Overload resolution is used to select the best unary operator in the class or struct given by the type of x.

Overload resolution is used to select the best binary operator in the classes or structs given by the types of x and y.

**Instance Constructor Invocation**

New T(x,y) Overload resolution is used to select the most appropriate instance constructor in the class or struct T.

**Argument Lists**

Every function member invocation will include an argument list. This list provides the values or variable references used by the parameters of the function member.

The syntax used for specifying the argument list will depend on the function member category. The following are rules for determining the argument list:

* For all the following, arguments are specified as an argument list (detailed later):
  + Delegates
  + Instance constructors
  + Methods
* For all properties, the argument list is empty when invoking the get accessor.
* For events, the argument list will be made up of the expression that appears as the right operand of the += or -= operator.
* For all indexers, the argument list is made up of the expressions specified between the square brackets ([ and ]) in the indexer access.
* For any user-defined operators, the argument list will be made up of the single operand of the unary operator or the two operands of the binary operator.

The arguments of the following are always passed as value parameters:

* Events
* Properties
* User-defined operators

Arguments of indexers are passed as value parameters or parameter arrays. Here is the structure of an argument list:

argument-list: argument

argument-list , argument

argument:

expression

ref variable-reference out variable-reference

An argument list is made up of one or more arguments. These arguments are separated by commas. Each argument can take one of the following forms:

124

* An expression used to indicate that the argument is passed as a value parameter
* The keyword ref followed by a variable-reference, which indicates that the argument is passed as a reference parameter
* The keyword out followed by a variable-reference, used to indicate that the argument is passed as an output parameter

Overload Resolution

Overload resolution is a mechanism used by the C# compiler that allows it to select the most appropriate function member to invoke given an argument list and a set of candidate function members.

Overload resolution selects the function member to invoke in the following way:

* + Invocation of a method named in an invocation expression
  + Invocation of an instance constructor named in an object-creation expression
  + Invocation of an indexer accessor through an element access
  + Invocation of a predefined or user-defined operator referenced in an expression

##### Primary Expressions

Primary expressions are made up of the simplest types of expression that can be found in C#:

primary-expression:

array-creation-expression

primary-no-array-creation-expression

primary-no-array-creation-expression: literal

simple-name parenthesized-expression member-access

invocation-expression element-access

this-access base-access

post-increment-expression post-decrement-expression object-creation-expression delegate-creation-expression typeof-expression

checked-expression unchecked-expression default-value-expression anonymous-method-expression

Literals

A primary expression made up of a literal will be classified as a value:

literal::

boolean-literal integer-literal real-literal character-literal string-literal null-literal

Simple Names

A simple name is made up of an identifier.

This identifier can be followed by a type argument list:

simple-name:

identifier type-argument-listopt

Parenthesized Expressions

A parenthesized expression is simply enclosed by parentheses:

parenthesized-expression: ( expression )

There’s very little to a parenthesized expression — the expression inside the parentheses is evaluated. The expression cannot denote a namespace or a type; otherwise, an error will be generated.

Member Access

A member access consists of either:

* A primary expression
* A predefined type
* Or a qualified-alias-member These will be followed by
* A “.” token
* An identifier
* And finally, optionally followed by a type argument list The following shows the syntax of the code that will be used:

member-access:

primary-expression . identifier type-argument-listopt predefined-type . identifier type-argument-listopt qualified-alias-member . identifier type-argument-listopt

predefined-type: one of bool

byte char decimal double float int

long object sbyte short string uint ulong ushort

A member access can take on either of the following forms:

* + E.I
  + E.I<A1, ..., AK>

E is a primary expression, predefined type, or qualified-alias-member; I is a single identifier, and <A1,

..., AK> is an optional type argument list.

Invocation Expressions

Invocation lists are used to invoke methods:

invocation-expression:

primary-expression ( argument-listopt )

The primary expression of an invocation expression is either a method group or a value of a delegate type.

If the primary expression is a method group, the invocation expression is a method invocation. If the pri- mary expression is a value of a delegate type, the invocation expression is a delegate invocation.

In the event that the primary expression is not a method group or a value of a delegate type, a compiler error is generated.

Element Access

An element access is made up of:

* + A primary-no-array-creation-expression, followed by
  + A “[“ token, followed by
  + An expression list, followed by
  + A “]” token.

The expression list consists of one or more expressions, which are separated by commas:

element-access:

primary-no-array-creation-expression [ expression-list ]

expression-list: expression

expression-list , expression

Array Access

For any array access, the primary-no-array-creation-expression of the element access will always be a value that is an array type.

The number of expressions in the expression list has to be the same as the rank of the array type. Each expression has to be of the type:

* int
* uint
* long
* ulong
* Any type that can be implicitly converted to one or more of the preceding types The result of evaluating an array access is a variable of the element type of the array.

Indexer Access

When dealing with indexer access, the primary-no-array-creation-expression of the element access will be one of the following:

* An interface type
* A struct
* A variable
* A value of a class

This Access

A this-access is made up of the reserved word this:

this-access: this

A this-access is only allowed in a code block of one of the following:

* An instance constructor
* An instance method
* An instance accessor

Base Access

A base-access is made up of the reserved word base followed by either:

* The “.” token and an identifier and optional type argument list

Or:

* + An expression list enclosed in square brackets The syntax is as follows:

base-access:

base . identifier type-argument-listopt base [ expression-list ]

new Operator

The new operator is used to create new instances of types. The new expression can take on three forms:

* + **Object-creation expressions.** Used to create new instances of class types and value types
  + **Array-creation expressions.** Used to create new instances of array types
  + **Delegate-creation expressions.** Used to create new instances of delegate types

While the new operator creates a new instance of a type, it does not mean that memory has been allo- cated, as this is handled automatically by the .NET Framework and will only consume resources when they are required.

typeof Operator

The typeof operator is used to obtain the System.Type object for a type:

typeof-expression: typeof ( type )

typeof ( unbound-type-name ) typeof ( void )

unbound-type-name:

identifier generic-dimension-specifieropt

identifier :: identifier generic-dimension-specifieropt unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:

< commasopt >

commas:

,

commas ,

sizeof Operator

The sizeof operator is used to return the number of 8-bit bytes occupied by a variable:

sizeof-expression:

sizeof ( unmanaged-type )

For many predefined types, the sizeof operator results in a constant int value, as shown in the follow- ing table:

|  |  |
| --- | --- |
| **Expression** | **Value** |
| sizeof(bool) | 1 |
| sizeof(byte) | 1 |
| sizeof(char) | 2 |
| sizeof(decimal) | 16 |
| sizeof(double) | 8 |
| sizeof(float) | 4 |
| sizeof(int) | 4 |
| sizeof(long) | 8 |
| sizeof(sbyte) | 1 |
| sizeof(short) | 2 |
| sizeof(uint) | 4 |
| sizeof(ulong) | 8 |
| sizeof(ushort) | 2 |

checked/unchecked Operators

The checked and unchecked operators are used to set the overflow-checking for integral-type arith- metic operations and conversions:

checked-expression: checked ( expression )

unchecked-expression: unchecked ( expression )

The checked operator is used to evaluate the contained expression in a checked context. The unchecked

operator, on the other hand, evaluates the contained expression in an unchecked context.

Default Value Expression

A default value expression obtains the default value of a type. Default value expressions are usually used to type parameters to work out whether they are value types or reference types:

default-value-expression: default ( type )

The result at runtime for reference values will be null, while if it is a value type, the result will be the default value of the type.

Anonymous Methods

An anonymous-method-expression is used to define anonymous methods. They evaluate to a value referencing the method:

anonymous-method-expression:

delegate anonymous-method-signatureopt block

anonymous-method-signature:

( anonymous-method-parameter-listopt ) anonymous-method-parameter-list:

anonymous-method-parameter

anonymous-method-parameter-list , anonymous-method-parameter

anonymous-method-parameter:

parameter-modifieropt type identifier

##### Unary Expressions

The following is a list of unary expressions:

unary-expression: primary-expression

+ unary-expression

- unary-expression

! unary-expression

~ unary-expression

pre-increment-expression pre-decrement-expression cast-expression

##### Cast Expressions

A cast-expression is used to explicitly convert an expression to a given type:

cast-expression:

( type ) unary-expression

##### Arithmetic Operators

The following operators are called the arithmetic operators:

* + \*
  + /
  + %

❑ +

* + – 131

The syntax of these expressions is as follows:

multiplicative-expression: unary-expression

multiplicative-expression \* unary-expression multiplicative-expression / unary-expression multiplicative-expression % unary-expression

additive-expression: multiplicative-expression

additive-expression + multiplicative-expression additive-expression – multiplicative-expression

##### Shift Operators

The two shift operators (<< and >>) are used to perform bit-shifting operations:

shift-expression: additive-expression

shift-expression << additive-expression

shift-expression right-shift additive-expression

The << operator shifts a value left by a number of bits specified, while the >> operator shifts a value right by a number of bits specified.

##### Relational/Type Testing Operators

Six relational and type-testing operators are available in C#:

❑ ==

* !=
* <
* >
* <=
* >=

The syntax of these is as follows:

relational-expression: shift-expression

relational-expression < shift-expression relational-expression > shift-expression relational-expression <= shift-expression relational-expression >= shift-expression relational-expression is type

relational-expression as type

equality-expression: relational-expression

equality-expression == relational-expression equality-expression != relational-expression

These are all comparison operators. All predefined comparison operators return a result of the bool type. The following table lists operators, along with the outcome of the operator on operands:

**Operator Outcome**

x == y x != y x < y

x > y x <= y x >= y

If x is equal to y, the result is true.

If x is not equal to y, the result is false.

If x is equal to y, the result is false.

If x is not equal to y, then the result is true.

If x is less than y, the result is true.

If x is greater than y, the result is false.

If x is less than y, the result is false.

If x is greater than y, the result is true.

If x is less than or equal to y, the result is true.

If x is greater than or equal to y, the result is false.

If x is less than or equal to y, the result is false.

If x is greater than or equal to y, the result is true.

##### Logical Operators

Three logical operators are available in C#:

* + &
  + |
  + ^

The & operator computes the bitwise logical AND of the two operands. The logical AND operation com- pares 2 bits, and if they are both “1”, the result is “1”; otherwise, the result is “0”.

The | operator computes the bitwise logical OR of the two operands. The logical OR operation compares 2 bits, and if they are both “1”, the result is “1”; otherwise, the result is “0”.

The ^ operator computes the bitwise logical exclusive OR of the two operands. The logical exclusive OR (XOR) operation compares 2 bits, and if exactly one of them is “1” (that is, if they are different values), the result is “1”; otherwise (if the bits are the same), the result is “0”.

##### Conditional Logical Operators

There are two logical conditional operators in C#:

* + &&
  + || 133

The following is the syntax for these operators:

conditional-and-expression: inclusive-or-expression

conditional-and-expression && inclusive-or-expression

conditional-or-expression: conditional-and-expression

conditional-or-expression || conditional-and-expression

The simplest way to think of && and || is as conditional forms of & and |. What do we mean by that? Well, let’s look at the following operations:

x && y x || y

These are equivalent to these operations:

x & y x | y

The only difference is that y in:

x && y

is evaluated only if x is true, while for:

x || y

y is evaluated only is x is false.

##### Null Coalescing Operator

The ?? operator is called a null coalescing operator:

null-coalescing-expression: conditional-or-expression

conditional-or-expression ?? null-coalescing-expression

The ?? operator allows conditional expressions to be written that are an excellent shorthand way of replacing if statements. They take on the form:

b ? x : y

First, the condition b is evaluated. If b is true, x is evaluated and becomes the result of the operation; otherwise, y is evaluated and this becomes the result of the operation.

A conditional expression can never evaluate x and y.

##### Assignment Operators

The assignment operators are used to assign a new value to a variable, event, property, or indexer element. Eleven assignment operators are available in C# (most of these you will have come across already):

❑ =

❑ +=

* -=
* \*=
* /=
* %=
* &=
* |=
* ^=
* <<=
* >>=

The = operator is called a simple assignment operator. It is used to assign the value of the right operand to the variable, property, or indexer element given by the left operand.

The operators created by prefixing an = character with a binary operator are called the compound assignment operators. These operators carry out operations on the two operands and then assign the resulting value to the variable, property, or indexer element given by the left operand.

The += and -= operators with an event access expression as the left operand are called the event assign- ment operators.

##### Expression

An expression is either a conditional-expression or an assignment:

expression:

conditional-expression assignment

##### Constant Expressions

A constant expression can be fully and completely evaluated at the point that the code is compiled:

constant-expression: expression

A constant expression can have any one of the following types:

* bool
* byte
* char
* decimal
* double
* enumeration type
* float
* int
* long
* null type
* sbyte
* short
* string
* uint
* ulong
* ushort

For more information on these types, check out Chapter 6.

The following constructs are all allowed in constant expressions:

* Literals
* Null literals
* References to const members of class and struct types
* References to members of enumeration types
* Cast expressions (as long as the type is one of the following: bool, byte, char, decimal, double, enumeration type, float, int, long, null type, sbyte, short, string, uint, ulong, or ushort)
* The following unary operators:

❑ +

* + –
  + !
  + ~
    - The following binary operators:

❑ +

* + - * –
      * \*
      * /
      * %
      * <<
      * >>
      * &
      * |
      * ^
      * &&
      * ||

❑ ==

* + - * !=
      * <
      * >
      * <=
      * >=
    - As long as each operand is one of the following:
      * bool
      * byte
      * char
      * decimal
      * double
      * enumeration type
      * float
      * int
      * long
      * null type
      * sbyte
      * short
  + string
  + uint
  + ulong
  + ushort
* The ?: operator
* sizeof expressions

##### Boolean Expressions

All Boolean expressions will return a result of the type bool:

boolean-expression: expression

The bool type has two possible values:

* true
* false

Boolean expressions are important in a number of other C# statements where a controlling conditional statement is required. These statements are:

* Do
* For
* If
* While

Boolean expressions have to be of a type that can be implicitly converted to bool or that implements

operator true.

##### Summary

In this chapter we’ve taken a detailed look at expressions in C#. These expressions will form the back- bone of a majority of code that a programmer will create.

In Chapter 10, you look at C# statements.