APPENDIX B

OPERATOR PRECEDENCE

Operator precedence specifies the order in which operators act on operands. When more than one operator can operate on an operand (because the operators are positioned next to the operand, one on either side), this operand is first processed by the operator of the higher precedence.

Table B.1 shows the precedence of the majority of C#'s operators. The operators positioned closer to the top of the table have higher precedence than operators placed in a lower position. Operators in the same category have the same precedence.

Note

L-R is short for Left-Right associativity. R-L is short for Right-Left associativity.

`Expr` is short for expression

TABLE B.1 Operator Precedence

CATEGORY	OPERATOR	ASSOCIATIVITY	MEANING
Grouping	`(<Expression>)`	L-R	Parentheses for grouping
Primary	`<Object_name>.<Member_name>`		Member access
	`<Method_name>(<Arguments>)`	L-R	Method call
	`<Array_name>[<Index>]`		Array access
	`<Variable>++`	R-L	Postfix increment operator
	`<Variable>--`	R-L	Postfix decrement operator
	`new <Class_name> (<Arguments>)`		Class instantiation
	`typeof(<Type>)`		Type evaluation
	`sizeof(<Type>)`		struct size determination
Unary	`+ <Expression>`	R-L	Unary positive
	`- <Expression>`	R-L	Unary negative
	`! <Boolean_expression>`	R-L	Logical negation (not)
	`++ <Variable>`	R-L	Prefix increment operator
	`-- <Variable>`	R-L	Prefix decrement operator
	`(<Type>) <Expr>`	R-L	Type cast

TABLE B.1 continued

CATEGORY	OPERATOR	ASSOCIATIVITY	MEANING
Multiplicative	`<Expr1> * <Expr2>`	L-R	Multiply
	`<Expr1> / <Expr2>`	L-R	Divide
	`<Expr1> % <Expr2>`	L-R	Modulus
Additive	`<Expr1> + <Expr2>`	L-R	Add
	`<Expr1> - <Expr2>`	L-R	Subtract
Relational	`<Expr1> < <Expr2>`	L-R	Less than
	`<Expr1> <= <Expr2>`	L-R	Less than or equal to
	`<Expr1> > <Expr2>`	L-R	Greater than
	`<Expr1> >= <Expr2>`	L-R	Greater than or equal to
	`<Object> is <Type>`		Type compatibility
Equality	`<Expr1> == <Expr2>`	L-R	Equal to
	`<Expr1> != <Expr2>`	L-R	Not equal to
Logical Bitwise	`<Bool_expr1> & <Bool_expr2>`	L-R	Bitwise And
	`<Bool_expr1> ^ <Bool_expr2>`	L-R	Bitwise exclusive Or
	`<Bool_expr1> | <Bool_expr2>`	L-R	Bitwise Or
Logical Boolean	`<Bool_expr1> && <Bool_expr2>`	L-R	Logical And
	`<Bool_expr1> || <Bool_expr2>`	L-R	Logical Or
	`<Bool_expr> ? <Expr1> : <Expr2>`	R-L	Conditional
Assignment	`<Variable> = <Expr>`	R-L	Simple assignment
	`<Variable> *= <Expr>`	R-L	Multiply and assign
	`<Variable> /= <Expr>`	R-L	Divide and assign
	`<Variable> %= <Expr>`	R-L	Find modulus and assign
	`<Variable> += <Expr>`	R-L	Add and assign
	`<Variable> -= <Expr>`	R-L	Subtract and assign