INTRODUCTION TO C#

Overview of C#

C#' (pronounced as 'C sharp') is a new computer-programming language developed by Microsoft Corporation, USA. C# is a fully *object-oriented* language like Java and is the first *Component-oriented* Language.

C# is designed for building robust, reliable and durable components to handle real-world applications. Major highlights of C# are:

- It is a brand new language derived from the C / C++ family
- It simplifies and modernizes C++
- It is the only component-oriented language available today
- It is the only language designed for the .NET Framework
- It is a concise, lean and modern language
- It combines the best features of many commonly used languages: the productivity of Visual Basic, the power of C++ and the elegance of Java
- It is intrinsically object-oriented and web-enabled
- It has a lean and consistent syntax
- It embodies today's concern for simplicity, productivity and robustness
- It will become the language of choice for .NET programming
- Major parts of .NET Framework are actually coded in C#

Characteristics of C#:

They are:

- Simple
- Consistent
- Modern
- Object-oriented
- Type-safe
- Versionable
- Compatible
- Interoperable and
- Flexible

1.4.1 Simple

C# simplifies C++ by eliminating irksome operators such as ->,:: and pointers. C# treats integer and Boolean data types as two entirely different types. This means that the use of = in place of = = in **if** statements will be caught by the compiler.

1.4.2 Consistent

C# supports an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

1.4.3 Modern

C# is called a modern language due to a number of features it supports. It supports

- Automatic garbage collection
- Rich intrinsic model for error handling
- Decimal data type for financial applications
- Modern approach to debugging and
- Robust security model

1.4.4 Object-Oriented

C# is truly object-oriented. It supports all the three tenets of object-oriented systems, namely,

- Encapsulation
- Inheritance
- Polymorphism

1.4.5 Type-safe

Type-safety promotes robust programs.

1.4.6 Versionable

Making new versions of software modules work with the existing applications is known as *versioning*.

C# provides support for versioning with the help of **new** and **override** keywords.

1.4.7 Compatible

C# enforces the .NET common language specifications and therefore allows inter-operation with other .NET languages.

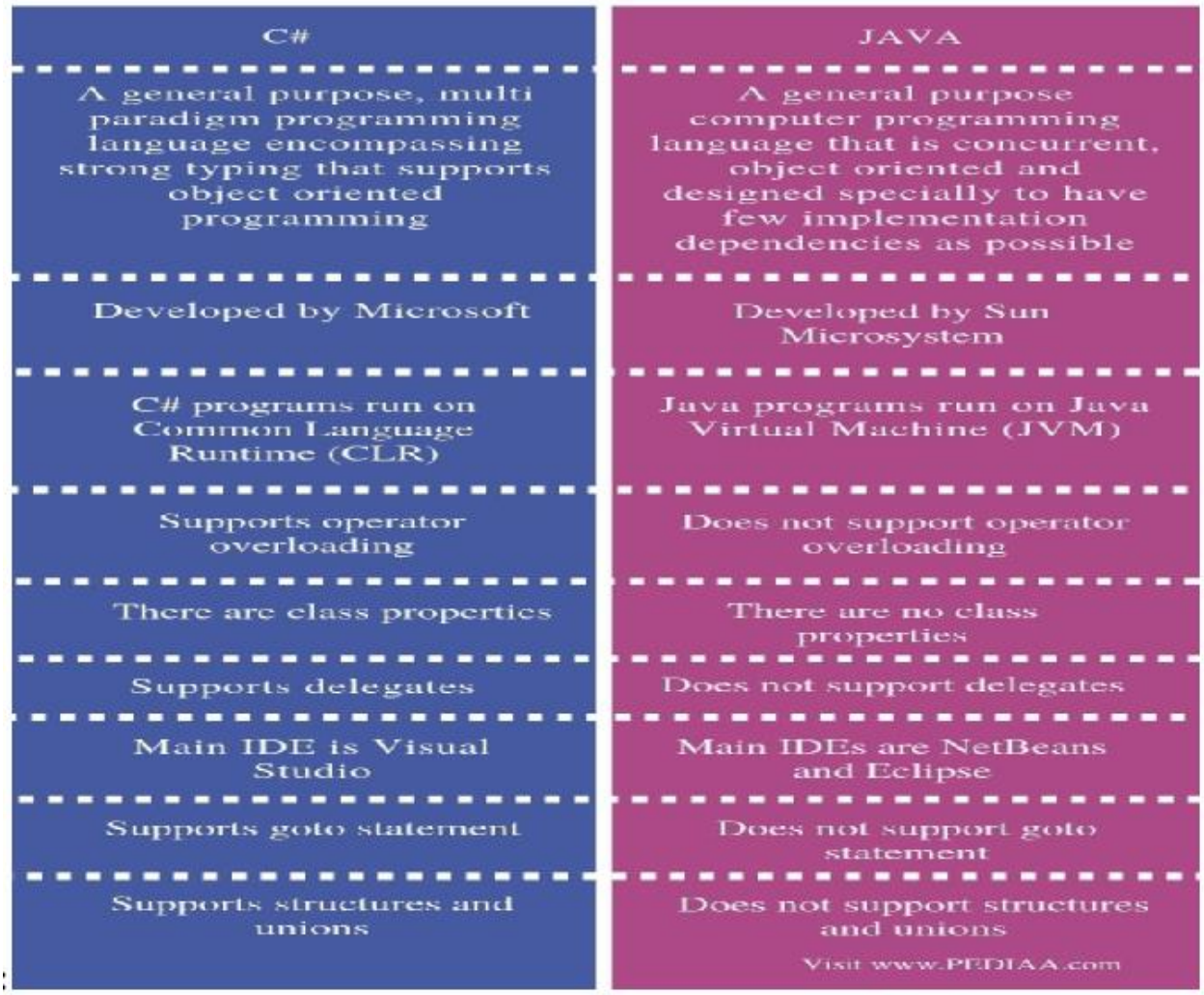
1.4.8 Flexible

Although C# does not support pointers, we may declare certain classes and methods as 'unsafe' and then use pointers to manipulate them. However, these codes will not be type-safe.

1.4.9 Inter-operability

C# provides support for using COM objects, no matter what language was used to author them. C# also supports a special feature that enables a program to call out any native API.

Differences between JAVA and C#:



Applications of C#

- Console applications
- Windows applications
- Developing Windows controls
- Developing ASP.NET projects
- Creating Web controls
- Providing Web services
- Developing .NET component library

Understanding .NET: The C# Environment:

The origins of .NET Technology:

The following are the 3 phases of origin:

- OLE technology
- COM technology
- .NET technology

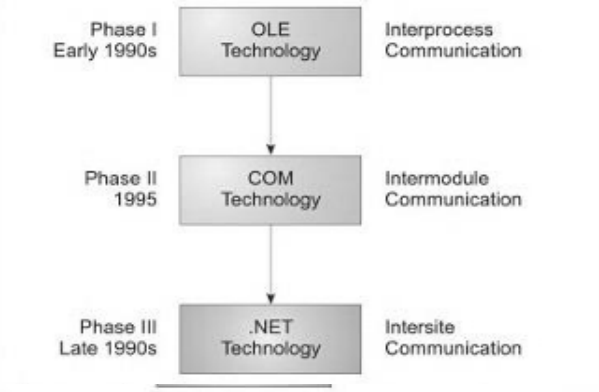


Fig. 2.2 *Three generations of component model*

OLE Technology

OLE (Object Linking and Embedding) technology was developed by Microsoft in the early 1990s to enable easy interprocess communications. OLE provided support to achieve the following:

- To embed documents from one application into another application
- To enable one application to manipulate objects located in another application

COM Technology

the component-based model for developing software programs. In the component-based approach, a program is broken into a number of independent components where each one offers a particular service. Each component can be developed and tested independently and then integrated it into the main system. This technology is known as the *Component Object Model* (COM) and the software built using COM is referred to as *componentware*.

.NET Technology

.NET technology is a third-generation component model. This provides a new level of inter-operability compared to COM technology. COM provides a standard binary mechanism for inter-module communication.

This line contains a number of keywords: **public**, **static** and **void.** This is very similar to the **main** of C++ and JavD. In contrast to Java and C++, **Main** has a capital, not lowercase M. The meaning and purpose of these keywords are give below:

public	The keyword **public** is an access modifier that tells the C# compiler that the **Main** method is accessible by anyone
static	The keyword **static** declares that the **Main** method is a global one and can be called without creating an instance of the class. The compiler stores the address of the method as the entry point and uses this information to begin execution before any objects are created.
void	The keyword **void** is a type modifier that states that the **Main** method does not return any value (but simply prints some text to the screen).

3.2.4 The Output Line

The only executable statement in the program is

```
System.Console.WriteLine("C# is sharper than C++.");
```

Namespaces:

Namespaces are the way C# segregates the .NET library classes into reasonable groupings.

C# supports a feature known as **using** directive that can be used to import the namespace **System** into the program. Once a namespace is imported, we can use the elements of that namespace without using the namespace as prefix. This is illustrated in Program 3.2.

Program 3.2 | **A PROGRAM USING NAMESPACES AND COMMENTS**

```
/*
This program uses namespaces and comment lines
*/
using System; // System is a namespace
class SampleTwo
{
    // Main method begins
    public static void Main( )
    {
        Console.WriteLine("Hello!");
    }
    //  Main method ends
}
```

Note that the first statement in the program is

```
using System;
```

This tells the compiler to look in the **System** library for unresolved class names. Note that we have not used the **System** prefix to the **Console** class in the output line.

Adding Comments:

C# permits two types of comments, namely,

- Single-line comments
- Multiline comments

Single-line comments begin with a double backslash (//) symbol and terminate at the end of the line. We can use // on a line of its own or after a code statement. This can also be used to comment out an entire line or part of a line of source code. Everything after the // on a line is considered a comment.

If we want to use multiple lines for a comment, we must use the second type known as multi-line comment. This comment starts with the /* characters and terminates with */ as shown in the beginning of the program. These comments can also occur within a line. For example:

```
using /* namespace */ System; //o.k
```

Command Line Arguments:

Command line arguments are parameters supplied to the **Main** method at the time of invoking it for execution. Program 3.6 accepts a name from the command line and writes it to the console.

Program 3.6 | COMMAND LINE ARGUMENTS

```
/*
This program uses command line arguments as input
*/
using System;

class SampleSix
{
    public static void Main (string [ ] args)
    {

        Console.Write ("welcome to");
        Console.Write (" "+args[0]);
        Console.WriteLine (" "+args[1]);
    }
}
```

Main is declared with a parameter **args**. The parameter **args** is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array **args** as its elements. We can access the array elements by using a subscript like **args[0], args[1]** and so on. For example, consider the command line

SampleSix **C sharp**

This command line contains two arguments which are assigned to the array **args** as follows:

C ⟶ args [0]
sharp ⟶ args [1]

Main with a Class:

Program 3.8 | A PROGRAM WITH TWO CLASSES

```
class TestClass              // class definition
{
    public void fun( )
    {
        System.Console.WriteLine("C# is modern");
    }
}

class SampleSeven
{
    public static void Main( )
    {

        TestClass test = new TestClass( );  // creating test object
        test.fun( );  // calling fun ( ) function
    }
}
```

This program has two class declarations, one for the **TestClass** and another for the **Main** method. **TestClass** contains only one method to print a string "C# is modern". The **Main** method in **SampleSeven** class creates an object of **TestClass** and uses it to invoke the method **fun ( )** contained in **TestClass** as follows:

```
TestClass test = new TestClass ( );
test.fun ( );
```

The object **test** is used to invoke the method **fun ( )** of **TestClass** with the help of the dot operator. Execution of Program 3.8 will produce the following output:

```
C# is modern
```

Using Mathematical Functions:

Program 3.10 | USING MATHEMATICAL FUNCTIONS

```
using System;
class SampleNine
{
    public static void Main( )
    {

        double x = 5.0; //Declaration and initialization
        double y;       // Simple declaration
        y = Math.Sqrt(x);
        Console.WriteLine ( " y = " + y) ;
    }
}
```

This program contains more executable statements. The statement

```
double x = 5.0;
```

declares a variable **x** and initializes it to the value 5.0 and the statement

```
double y;
```

merely declares a variable **y**. Note that both of them have been declared as **double** type variables. **double** is a data type used to represent a floating-point number. Data types are discussed in the next chapter.

The statement

```
y = Math.Sqrt(x);
```

invokes the method **Sqrt()** of **Math** class which is a part of **System** namespace. The program produces the following output:

```
y = 2.23606
```

Compile Time Errors:

A program contains 2 types of errors:

- Syntax errors
- Logic errors

While syntax errors will be caught by the compiler, logic errors should be eliminated by testing the program logic carefully.

Program Structure:

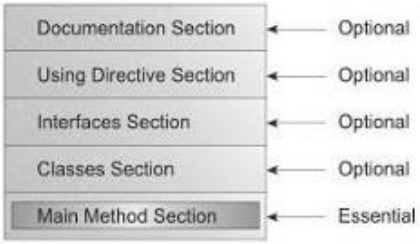


Fig. 3.3 *C# program structure*

An executable C# program may contain a number coding blocks as shown in Fig. 3.3. The documentation section consists of a set of comments giving the name of the program, the author, date and other details, which the programmer (or other users) may like to use at a later stage. Comments must explain the

- Why and what of classes, and the
- How of algorithms

This would greatly help maintaining the program.

The **using** directive section will include all those namespaces that contain classes required by the application. **using** directives tell the compiler to look in the namespace specified for these unresolved classes.

An interface is similar to a class but contains only abstract members. Interfaces are used when we want to implement the concept of multiple inheritance in a program. Interface

A C# program may contain multiple class definitions. Classes are the primary and essential elements of a C# program. These classes are used to map the objects of real-world problems. The number of classes depends on the complexity of the problem.

Since every C# application program requires a **Main** method as its starting point, the class containing the **Main** is the essential part of the program. A simple C# program may contain only this part. The **Main** method creates objects of various classes and establishes communications between them. On reaching the end of **Main**, the program terminates and the control passes back to the operating system.

Literals, Variables and DataTypes:

In simple terms, a C# program is a collection of tokens, comments and white spaces. C# includes the following five types of tokens:

- Keywords
- Identifiers
- Literals
- Operators
- Punctuators

Table 4.1 *C# keywords*

abstract	event	namespace	static
as	explicit	new	string
base	extern	null	struct
bool	false	object	switch
break	finally	operator	this
byte	fixed	out	throw
case	float	override	true
catch	for	params	try
char	foreach	private	typeof

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, labels, namespaces, interfaces, etc. C# identifiers enforce the following rules:

- They can have alphabets, digits and underscore characters
- They must not begin with a digit
- Upper case and lower case letters are distinct
- Keywords in stand-alone mode cannot be used as identifiers

C# permits the use of keywords as identifiers when they are prefixed with the '@' character

Operators are symbols used in expressions to describe operations involving one or more operands.

Punctuators are symbols used for grouping and separating code. They define the shape and function of a program. Punctuators (also known as *separators*) in C# include:

- Parentheses ( )
- Braces { }
- Brackets [ ]
- Semicolon ;
- Colon :
- Comma ,
- Period .

Statements in C# are like sentences in natural languages. A statement is an executable combination of tokens ending with a semicolon. C# implements several types of statements. They include:

- Empty statements
- Labeled statements
- Declaration statements
- Expression statements
- Selection statements
- Interaction statements
- Jump statements
- The **try** statements
- The **checked** statements
- The **unchecked** statements
- The **lock** statements
- The **using** statements

Program 4.1 | **IDENTIFIER EXAMPLE**

```
using System;
class IdentifierExample
{
    public static void Main()
    {
       // Using if keyword as an identifier by prefixing @
       int @if;
       Console.WriteLine("—Demonstrating use of if keyword as an identifier by prefixing @—\n");
       for(@if = 0; @if < 10; @if++)
          Console.WriteLine("The value of @if is: {0}",@if);
    }
}
```

Literals:

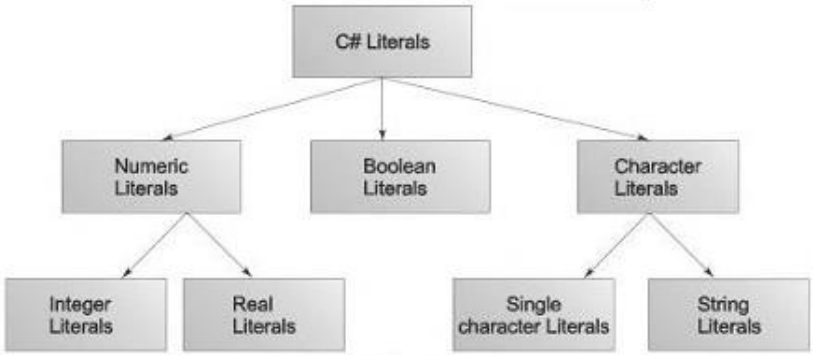


Fig. 4.1 *C# literals*

4.2.1 Integer Literals

An *integer* literal refers to a sequence of digits. There are two types of integers, namely, *decimal* integers and *hexadecimal* integers.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer literals are:

123 -321 0 654321

4.2.2 Real Literals

Integer literals are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on.

0.0083 -0.75 435.36

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part, which is an integer.

4.2.3 Boolean Literals

There are two Boolean literal values:

- true
- false

4.2.4 Single Character Literals

A single-character literal (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character in the examples above constants are:

'5' 'X' ';' ' '

4.2.5 String Literals

A string literal is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces. Examples are:

"Hello C#" "2001" "WELL DONE" "?...!" "5+3" "X"

4.2.6 Backslash Character Literal

C# supports some special backslash character constants that are used in output methods. For example, the symbol '\n' stands for a new-line character. A list of such backslash character literals is given in Table 4.2.

Table 4.2 *Backslash character literals*

CONSTANT	MEANING
'\a'	alert
'\b'	back space
'\f'	form feed
'\n'	new-line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\\'	backslash
'\0'	null

Variables:

A *variable* is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. Every variable has a type that determines what values can be stored in the variable.

A variable name can be chosen by the programmer in a meaningful way so as to reflect what it represents in the program. Some examples of variable names are:

- average
- height
- total_height
- classStrength

As mentioned earlier, variable names may consist of alphabets, digits and the underscore (_), subject to the following conditions:

1. They must not begin with a digit.
2. Uppercase and lowercase are distinct. This means that the variable **Total** is not the same as **total** or **TOTAL**.
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of any length.

DataTypes:

Every variable in C# is associated with a data type. Data types specify the size and type of values that can be stored. C# is a language rich in its *data types*. The variety available allows the programmer to select the type appropriate to the needs of the application.

The types in C# are primarily divided into two categories:

- Value types
- Reference types

Value types (which are of fixed length) are stored on the *stack*, and when a value of a variable is assigned to another variable, the value is actually copied. This means that two identical copies of the value are available in memory. Reference types (which are of variable length) are stored on the *heap*, and when an assignment between two reference variables occurs, only the reference is copied; the actual value remains in the same memory location. This means that there are two references to a single value.

A third category of types called *pointers* is available for use only in *unsafe code*. Value types and reference types are further classified as *predefined* and *user-defined* types as shown in Fig. 4.2.

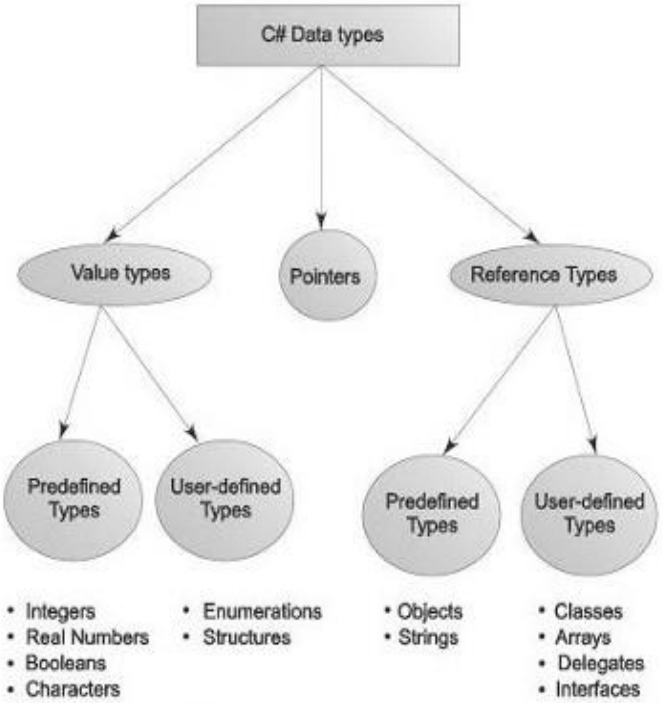


Fig. 4.2 *Taxonomy C# data types*

Value Types:

The value types of C# can be grouped into two categories (as shown in Fig. 4.2), namely,

- User-defined types (or complex types) and
- Predefined types (or simple types)

We can define our own complex types known as *user-defined* value types which include **struct** types and enumerations. They are discussed in Chapter 11.

Predefined value types which are also known as *simple types* (or primitive types) are further subdivided into:

- Numeric types,
- Boolean types, and
- Character types.

Numeric types include integral types, floating-point types and decimal types. These are shown diagrammatically in Fig. 4.3.

Note: C# 2.0 added a new type called *nullable* type. This type variable can hold an undefined value. Any value type variable can be defined as a nullable type.

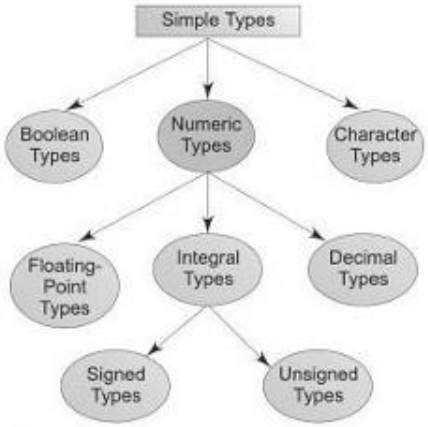


Fig. 4.3 *Categories of simple type data*

Reference Types:

As with value types, the reference types can also be divided into two groups:

- User-defined (or complex) types
- Predefined (or simple) types

User-defined reference types refer to those types which we define using predefined types. They include:

- Classes
- Interfaces
- Delegates
- Arrays

These complex types will be discussed in later chapters when we take up these topics individually.

Predefined reference types include two data types:

- Object type
- String type

The **object** type is the ultimate base type of all other intrinsic and user-defined types in C#. We can use an **object** reference to bind to an object of any particular type. We shall see later how we can use the **object** type to convert a value type on the stack to an **object** type to be placed on the heap.

C# provides its own string type for creating and manipulating strings. String literals discussed earlier can be stored in string objects as values. We can perform a number of operations such as copying, comparing and concatenation on these string objects.

Scope of Variables:

The scope of a variable is the region of code within which the variable can be accessed. This depends on the type of the variable and place of its declaration. C# defines several categories of variables. They include:

- Static variables
- Instance variables
- Array elements
- Value parameters
- Reference parameters
- Output parameters
- Local variables

Boxing and Unboxing:

4.12.1 Boxing

Any type, value or reference can be assigned to an object without an explicit conversion. When the compiler finds a value type where it needs a reference type, it creates an object 'box' into which it places the value of the value type. The following code illustrates this:

```
int m = 100;
object om = m; // creates a box to hold m
```

When executed, this code creates a temporary reference_type 'box' for the object on heap. We can also use a C-style cast for boxing.

```
int m = 100;
object om = (object)m; //C-style casting
```

Note that the boxing operation creates a copy of the value of the **m** integer to the object **om**. Now both the variables **m** and **om** exist but the value of **om** resides on the heap. This means that the values are independent of each other. Consider the following code:

```
int m = 10:
object om = m;
m = 20;
Console.WriteLine(m);  // m = 20
Console.WriteLine(om); //om = 10
```

When a code changes the value of **m**, the value of **om** is not affected.

4.12.2 Unboxing

Unboxing is the process of converting the object type back to the value type. Remember that we can only unbox a variable that has previously been boxed. In contrast to boxing, unboxing is an explicit operation using C-style casting.

```
int m = 10;
object om = m;    //box m
int n = (int)om;    //unbox om back to an int
```

When performing unboxing, C# checks that the value type we request is actually stored in the object under conversion. Only if it is, the value is unboxed.

When unboxing a value, we have to ensure that the value type is large enough to hold the value of the object. Otherwise, the operation may result in a runtime error. For example, the code

```
int m = 500;
object om = m;
byte n = (byte)om;
```

will produce a runtime error.

Operators and Expessions:

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions.

C# supports a rich set of operators. C# operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators.
7. Bitwise operators
8. Special operators

Table 5.1 *Arithmetic operators*

OPERATOR	MEANING
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Table 5.2 *Relational operators*

OPERATOR	MEANING
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Table 5.3 *Relational expressions*

EXPRESSION	VALUE
4.5 <= 10	true
4.5 < –10	false
–35 >= 0	false
10 < 7+5	true
5.0 != 5	false
a + b == c+d	true*

Program 5.2 IMPLEMENTATION OF RELATIONAL OPERATORS

```
using System;

class RelationalOperators
{
    public static void Main( )
    {
        float a = 15.0F, b = 20.75F, c = 15.0F;
        Console.WriteLine(" a = " + a);
        Console.WriteLine(" b = " + b);
        Console.WriteLine(" c = " + c);
        Console.WriteLine(" a < b is " + (a<b));
        Console.WriteLine(" a > b is " + (a>b));
        Console.WriteLine(" a == c is " + (a==c));
        Console.WriteLine(" a <= c is " + (a<=c));
        Console.WriteLine(" a >= b is " + (a>=b));
        Console.WriteLine(" b != c is " + (b!=c));
        Console.WriteLine(" b == a+c is " + (b==a+c));
    }
}
```

The output of Program 5.2 would be:

```
a = 15
b = 20.75
c = 15
a < b is true
a > b is false
a == c is true
a <= c is true
a >= b is false
b != c is true
b == a+c is false
```

Table 5.4 *Logical operators*

OPERATOR	MEANING
&&	logical AND
||	logical OR
!	logical NOT
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical exclusive OR

Table 5.6 *Shorthand assignment operators*

STATEMENT WITH SIMPLE ASSIGNMENT OPERATOR	STATEMENT WITH SHORTHAND OPERATOR
a = a+1	a += 1
a = a–1	a –= 1
a = a*(n+1)	a *= n+1
a = a/(n+1)	a /= n+1
a = a%b	a %= b

Increment and decrement operators:

C# has two very useful operators not generally found in many other languages. These are the increment and decrement operators:

++ and --

Program 5.3 | INCREMENT OPERATOR ILLUSTRATED

```
class IncrementOperator
{
      public static void Main()
      {
           int m = 10, n = 20;
           System.Console.WriteLine(" m = " + m);
           System.Console.WriteLine(" n = " + n);
           System.Console.WriteLine(" ++m = " + ++m);
           System.Console.WriteLine(" n++ = " + n++);
           System.Console.WriteLine(" m = " + m);
           System.Console.WriteLine(" n = " + n);
      }
}
```

The output of Program 5.3 is as follows:

```
m = 10
n = 20
++m = 11
n++ = 20
m = 11
n = 21
```

Conditional Operator:

The character pair **? :** is a *ternary operator* available in C#. This operator is used to construct conditional expressions of the form

exp1 ? *exp2* : *exp3*

where *exp1*, *exp2* and *exp3* are expressions.

Table 5.7 *Bitwise operators*

OPERATOR	MEANING
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical XOR
~	one's complement
<<	shift left
>>	shift right

Special operators:

C# supports the following special operators.

is	(relational operator)
as	(relational operator)
typeof	(type operator)
sizeof	(size operator)
new	(object creator)
.(dot)	(member-access operator)
checked	(overflow checking)
unchecked	(prevention of overflow checking)

Arithmetic Expressions:

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. We have used a number of simple expressions in the examples discussed so far. C# can handle any complex mathematical expressions. Some of the examples of C# expressions are shown in Table 5.8. Remember that C# does not have an operator for exponentiation.

Table 5.8 *Expressions*

ALGEBRAIC EXPRESSION	*C# EXPRESSION*
axb–c	a*b–c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2+2x+1$	3*x*x+2*x+1
$\frac{x}{y}+c$	x/y+c

Evaluation of Expressions:

Expressions are evaluated using an assignment statement of the form

variable = expression;

variable is any valid C# variable name. When the statement is encountered, the *expression* is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a*b-c;
y = b/c*a;
z = a-b/c+d;
```

Precedence of arithmetic operators:

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C#:

High priority * / %

Low priority + -

Type Conversions:

We often encounter situations where there is a need to convert a data of one type to another before it is used in arithmetic operations or to store a value of one type into a variable of another type. For example, consider the code below:

```
byte b1 = 50;
byte b2 = 60;
byte b3 = b1 + b2;
```

This code attempts to add two byte values and to store the result into a third byte variable. But this will not work. The compiler will give an error message:

"cannot implicitly convert type **int** to type **byte**."

In C#, type conversions take place in two ways (as shown in Fig. 5.1)

- Implicit conversions
- Explicit conversions

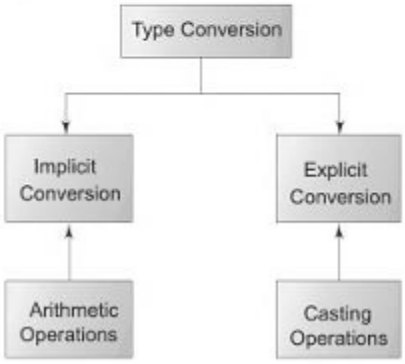


Fig. 5.1 *Type conversions in C#*

5.13.1 Implicit Conversions

Implicit conversions are those that will always succeed. That is, the conversion can always be performed without any loss of data. For numeric types, this implies that the *destination* type can fully represent the range of the *source* type. For example, a **short** can be converted implicitly to an **int**, because the **short** range is a subset of the **int** range. Therefore,

```
short b = 75;
int a = b, // implicit conversion.
```

For example, implicit conversions are possible in the following cases:

- From **byte** to **decimal**
- From **uint** to **double**
- From **ushort** to **long**

5.13.2 Explicit Conversions

There are many conversions that cannot be implicitly made between types. If we attempt such conversions, the compiler will give an error message. For example, the following conversions cannot be made implicitly:

- **int** to **short**
- **int** to **uint**
- **uint** to **int**
- **float** to **int**
- **decimal** to any numeric type
- any numeric type to **char**

Table 5.9 *Use of casts*

EXAMPLE	ACTION
x = (**int**) 7.5	7.5 is converted to integer by truncation.
a =(**int**)21.3/(**int**) 4.5	Evaluated as 21/4 and the result would be 5.
b = (**double**) sum/n	Division is done in floating-point mode.
y = (**int**) (a+b)	The result of a + b is converted to integer.
z = (**int**) a+b	**a** is converted to integer and then added to b.
p = cost ((**double**) x)	Converts x to double before using it as parameter.

Program 5.6 | **THE USE OF CASTING OPERATION**

```
using System;
class Casting
{
    public static void Main ( )
    {
        float sum;
        int i ;
        sum = 0.0F;
        for (i = 1; i <= 10 ; i++)
        {
            sum = sum + 1/ (float)i;
            Console.Write (" i = " + i );
            Console.WriteLine (" Sum = " + sum) ;
        }
    }
}
```

The output of Program 5.6:

```
i = 1       Sum = 1
i = 2       Sum = 1.5
i = 3       Sum = 1.833333
i = 4       Sum = 2.083333
i = 5       Sum = 2.283334
i = 6       Sum = 2.45
i = 7       Sum = 2.592857
i = 8       Sum = 2.717857
i = 9       Sum = 2.828969
i = 10      Sum = 2.928968
```

Control Structures:

C# language possesses such decision-making capabilities and supports the following statements known as *control* or *decision-making* statements.

1. **if** statement
2. **switch** statement
3. Conditional operator statement

Decision making with 'if' statement:

The **if** statement may be implemented in different forms depending on the complexity of the conditions to be tested.

1. Simple **if** statement
2. **if..else** statement
3. Nested **if..else** statement
4. **else if** ladder

1.Simple 'if' statement:

The general form of a simple **if** statement is

```
if(boolean-expression)
{
        statement-block;
}
statement-x;
```

2.The 'if....else' statement:

The **if....else** statement is an extension of the simple **if** statement. The general form is

```
if(boolean_expression)
{
    True-block statement(s)
}
else
{

    False-block statement(s)
}
statement-x
```

Program 6.2 counts the even and odd numbers in a list of numbers using the **if....else** statement.

Program 6.2 | EXPERIMENTING WITH IF....ELSE STATEMENT

```
using System;
class IfElseTest
{
  public static void Main( )
{
  int[] number = { 50, 65, 56, 71, 81 };
  int even = 0, odd = 0;

  for (int i = 0; i < number.Length; i++)
  {
    if ((number[i] % 2) == 0)  // use of modulus operator
    {
      even += 1;  // counting EVEN numbers
    }
    else
    {
      odd += 1;  // counting ODD numbers

       }
  }

  Console.WriteLine("Even Numbers : " + even);
  Console.WriteLine("Odd Numbers : " + odd);
 }
}
```

Output of Program 6.2 would be:

```
Even Numbers    : 2
Odd Numbers     : 3
```

3.Nesting of 'if...else' statement:

When a series of decisions are involved, we may have to use more than one **if....else** statement in *nested* form as follows:

```
if (test condition1)
{
    if (test condition2)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
}
else
{
        statement-3;
}
statement-x;
```

4.The 'else...if' ladder:

There is another way of putting **ifs** together when multipath decisions are involved. A multipath decision is a chain of **ifs** in which the statement associated with each **else** is an **if**. It takes the following general form:

```
if (condition 1)
    statement-1;
else if (condition 2)
    statement-2;
    else if (condition 3)
        statement-3;
    else if (condition n)
        statement-n;
    else
        default-statement;
statement-x;
```

Program 6.5 | **DEMONSTRATION OF ELSE IF LADDER**

```
using System;
class ElseIfLadder
{
  public static void Main( )
  {
    int[ ] rollNumber = { 111, 222, 333, 444 };
    int[ ] marks = { 81, 75, 43, 58 };
    for (int i = 0; i < rollNumber.Length; i++)
    {
      if (marks[i] > 79)
        Console.WriteLine(rollNumber[i] + " Honours");
      else if (marks[i] > 59)
          Console.WriteLine(rollNumber[i] + " I Division");
        else if (marks[i] > 49)
            Console.WriteLine(rollNumber[i] + " II Division");
          else
            Console.WriteLine(rollNumber[i] + " FAIL");
    }
  }
}
```

Program 6.5 produces the following output:

```
111 Honours
222 I Division
333 FAIL
444 II Division
```

The Switch Statement

C# has a built-in multiway decision statement known as a **switch**. The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statements associated with that **case** is executed. The general form of the **switch** statement is as shown below:

```
switch(expression)
{
    case value-1:
        block-1
        break;

    case value-2:
        block-2
        break;
    ......
    ......
    default:
        default-block
        break;
}
statement-x;
```

The **switch** statement is executed in the following order:

1. The **expression** is evaluated first.
2. The value of the expression is successively compared against the values, *value-1, value-2, ...* If a **case** is found whose value matches the value of the *expression*, then the block of statements that follows the case are executed.
3. The **break** statement at the end of each block signals the end of a particular case and causes an exit from the **switch** statement, transferring the control to the *statement-x* following the **switch.**
4. The **default** is an optional case. When present, it will be executed if the value of the expression does not match any of the case values. If not present, no action takes place when all matches fail and the control goes to the *statement-x.*

Program 6.6 illustrates the use of **switch** for a menu-driven interactive program.

Program 6.6 | **TESTING THE SWITCH STATEMENT**

```
using System;
class CityGuide
{
    public static void Main( )
    {
        Console.WriteLine("Select your choice");
        Console.WriteLine("London");
        Console.WriteLine("Bombay");
        Console.WriteLine("Paris");
        Console.WriteLine("Type your choice");
        String name = Console.ReadLine ( );

        switch (name)
        {
            case "Bombay":
                Console.WriteLine("Bombay:Guide 5");
                break;
            case "London":
                Console.WriteLine("London:Guide 10");
                break;
            case "Paris":
                Console.WriteLine("Paris:Guide 15");
                break;
            default:
                Console.WriteLine ("Invalid choice");
                break;
        }
    }
}
```

Output of Program 6.6:

```
Select your choice
London
Bombay
Paris
Type your choice
London
London : Guide 10
```

Fallthrough in Switch statement:

In the absence of the break statement in a case block, if the control moves to the next case block without any problem, it is known as 'fallthrough'. Fallthrough is permitted in C, C++ and Java. But C# does not permit automatic fallthrough, if the case block contains executable code.

The '?:' operator:

C# has an unusual operator, useful for making two-way decisions; it is a combination of ? and :, and takes three operands. This operator is popularly known *as the conditional operator*. The general form of use of the *conditional operator* is as follows:

conditional expression ? expression1 : expression2

The *conditional expression* is evaluated first. If the result is true, *expression 1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned. For example, the segment

```
if (x < 0)
    flag = 0;
else
    flag = 1;
```

can be written as

```
flag = (x<0) ? 0 : 1;
```

The 'foreach' statement:

The **foreach** statement is similar to the **for** statement but implemented differently. It enables us to iterate the elements in arrays and collection classes such as **List** and **HashTable**. The general form of the **foreach** statement is:

```
foreach (type variable in expression)
{
    Body of the loop
}
```

The *type* and *variable* declare the *iteration* variable. During execution, the iteration variable represents the array element (or collection element in case of collections) for which an iteration is currently being performed. **in** is a keyword.

The *expression* must be an *array* or *collection* type and an explicit conversion must exist from the element type of the collection to the type of the iteration variable. *Example:*

```
public static void Main (string [ ] args)
{
    foreach ( string s in args)
    {
        Console.WriteLine(s);
    }
}
```

This program segment displays the command line arguments. The same may be achieved using the **for** statement as follows:

```
public static void Main (string[ ] args)
{
    for ( int i = 0; i < args.Length ; i++ )
    {
        Console.WriteLine(args [i]);
    }
}
```

Program 7.6 illustrates the use of **foreach** statement for printing the contents of a numerical array.

Program 7.6 | **PRINTING ARRAY VALUES USING FOREACH STATEMENT**

```
using System;
class ForeachTest
{
        public static void Main ( )
        {
             int[ ] arrayInt = { 11, 22, 33, 44 };
             foreach ( int m in arrayInt)
             {
                  Console.Write(" " + m);
             }
                  Console.WriteLine( );
        }
}
```

Program 7.6 will display the following output:

```
11   22   33   44
```

Methods in C#:

Objects encapsulate data, and code to manipulate that data. The code designed to work on the data is known as *methods* in C#. Methods give objects their behavioral characteristics. They are used not only to access and process data contained in the object but also to provide responses to any messages received from other objects.

Declaring Methods:

Methods are declared inside the body of a class, normally after the declaration of data fields. The general form of a method declaration is

```
modifiers type methodname (formal-parameter-list)
{
     method _ body
}
```

Method declaration has five parts:

- Name of the method (*methodname*)
- Type of value the method returns *(type)*
- List of parameters (*formal-parameter-list*)
- Body of the method
- Method modifiers (*modifier*)

The *methodname* is a valid C# identifier. The *type* specifies the type of value the method will return. This can be a simple data type such as **int** as well as any class type. If the method does not return anything, we specify a return type of **void.** Note that we cannot omit the return type altogether.

The *formal-parameter-list* is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The parameters are separated by commas. In the case where no input data are required, the declaration must still include an empty set of parantheses( ) after the method name. Examples are:

```
int Fun1 (int m, float x, float y) //three parameters
void Display ( ) //no parameters
```

Invoking Methods:

The invoking is done using the dot operator as shown below:

```
objectname.methodname( actual-parameter-list );
```

Here, *objectname* is the name of the object on which we are calling the method *methodname*. The *actual-parameter-list* is a comma separated list of 'actual values' (or expressions) that must match in type, order and number with the formal parameter list of the *methodname* declared in the class.

Program 8.1 | **DEFINING AND INVOKING A METHOD**

```
using System;
class Method   //   class containing the method
{
     // Define the Cube method
     int Cube ( int x )
     {
          return ( x * x * x );
     }
}
// Client class to invoke the cube method
class MethodTest
{
          public static void Main( )
          {
              // Create object for invoking cube
              Method M = new Method ( );
              // Invoke the cube method
              int y = M.Cube (5);   //Method call
              // Write the result
              Console.WriteLine( y );
         }
}
```

This program will display an output of 125.

Nesting of Methods:

a method can be called using only its name by another method of the

same class. This is known as *nesting of methods*.

Program 8.3 illustrates nesting of methods inside a class. The class **Nesting** defines two methods, namely **Largest ( )** and **Max ( ).** The method **Largest ( )** calls the method **Max ( )** to determine the largest of the two numbers and then displays the result.

Program 8.3 | NESTING OF METHODS

```
using System;
class Nesting
{
     void Largest ( int m, int n )
     {
          int large = Max ( m , n );   //Nesting
          Console.WriteLine( large );
     }
     int  Max (int a, int b)
     {
          int x = ( a > b ) ? a : b ;
          return ( x );
     }
}
class NestTesting
{
     public static void Main( )
     {
          Nesting next = new Nesting ( );
          next.Largest ( 100, 200 ) ;   //Method call
     }
}
```

Method Parameters:

The invocation involves not only passing the values into the method but also getting back the results from the method. For managing the process of passing values and getting back the results, C# employs four kinds of parameters.

- Value parameters
- Reference parameters
- Output parameters
- Parameter arrays

Pass by Value:

By default, method parameters are *passed by value*. That is, a parameter declared with no modifier is passed by value and is called a *value parameter*. When a method is invoked, the values of actual parameters are assigned to the corresponding formal parameters. The values of the value parameters can be changed within the method. The value of the actual parameter that is passed by value to a method is not changed by any changes made to the corresponding formal parameter within the body of the method.

Program 8.4 | ILLUSTRATION OF PASSING BY VALUE

```
using System;
class PassByValue
{
          static void Change (int m)
          {
               m = m+10;   //value of m is changed
          }
          public static void Main( )
          {
               int x = 100;
               Change (x);
               Console.WriteLine( "x =" + x );
          }
}
```

This program will produce the output

```
X = 100
```

Pass by reference:

We have just seen that pass by value is the default behaviour of methods in C#. We can, however, force the value parameters to be passed by reference. To do this, we use the **ref** keyword. A parameter declared with the **ref** modifier is a reference parameter. *Example:*

```
void Modify ( ref int x )
```

Here, **x** is declared as a reference parameter.

Program 8.5 | SWAPPING VALUES USING REF PARAMETERS

```
using System;
class PassByRef
{
        static void Swap ( ref int x, ref int y )
        {
             int temp = x;
             x = y;
             y = temp;
        }
        public static void Main( )
        {
             int m = 100;
             int n = 200;
             Console.WriteLine("Before Swapping:");
             Console.WriteLine("m = " + m);
             Console.WriteLine("n = " + n);
             Swap( ref m , ref n );
             Console.WriteLine("After Swpaping:");
             Console.WriteLine("m = " + m);
             Console.WriteLine("n = " + n);
        }
}
```

The Output Parameters:

As pointed out earlier, *output* parameters are used to pass results back to the calling method. This is achieved by declaring the parameters with an **out** keyword. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, it becomes an alias to the parameter in the calling method. When a formal parameter is declared as **out**, the corresponding actual parameter in the calling method must also be declared as **out.** For example,

```
void Output ( out int x )
{
   x = 100;
}
. . . .
int m; //m is uninitialized
Output ( out m ); //value of m is set
```

Program 8.6 | DEFINING AN OUT PARAMETER

```
using System;
class Output
{
     static void Square ( int x, out int y )
     {
          y = x * x;
     }
     public static void Main( )
     {
          int m;   //need not be initialized
          Square ( 10, out m );
          Console.WriteLine("m = " + m);
     }
}
```

Variable argument list:

In C#, we can define methods that can handle variable number of arguments using what are known as *parameter arrays*. Parameter arrays are declared using the keyword ***params***. *Example:*

```
void Function1 ( Params int [ ] x )
{
    ....
    ....
}
```

Program 8.7 | CONCEPT OF VARIABLE ARGUMENTS

```
using System;
class Params
{
    static void Parray ( params int [ ] arr )
    {
        Console.Write("Array elements are:");
        foreach (int i in arr)
            Console.Write(" " + i);
        Console.WriteLine( );
    }
    public static void Main( )
    {
        int [ ] x = { 11, 22, 33 } ;
        Parray ( x) ;       // call 1
        Parray ( ) ;              // call 2
        Parray ( 100, 200 ) ;      // call 3
    }
}
```

The output of Program 8.7 would be:

```
Array elements are : 11 22 33
Array elements are :
Array elements are : 100 200
```

Methods Overloading:

C# allows us to create more than one method with the same name, but with the different parameter lists and different definitions. This is called *method overloading*. Method overloading is used when methods are required to perform similar tasks but using different input parameters.

Overloaded methods must differ in number and/or type of parameters they take. This enables the compiler to decide which one of the definitions to execute depending on the type and number of arguments in the method call. Note that the method's return type does not play any role in the overload resolution.

The method selection involves the following steps:

1. The compiler tries to find an exact match in which the types of actual parameters are the same and uses that method.
2. If the exact match is not found, then the compiler tries to use the implicit conversions to the actual arguments and then uses the method whose match is unique. If the conversion creates multiple matches, then the compiler will generate an error message.

Program 8.9 | METHOD OVERLOADING

```
using System;
class Overloading
{
        public static void Main( )
        {
            Console.WriteLine(volume (10));
            Console.WriteLine(volume(2.5 F, 8));
            Console.WriteLine(volume(100L,75,15));
        }
        static int volume ( int x )                          // cube
        {
            return ( x * x * x ) ;
        }
        static double volume ( float r , int h )             // cylinder
        {
            return ( 3.14519 * r * r * h ) ;
        }
            static long volume ( long l , int b , int h )    // box
        {
            return ( l * b * h ) ;
        }
}
```

ARRAYS

An array is a group of contiguous or related data items that share a common name.

One-dimensional array:

A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted* variable or a *one-dimensional* array.

The subscript can begin with number 0. That is

x[0]

is allowed. For example, if we want to represent a set of five numbers, say (35,40,20,57,19), by an array variable **number**, then we may create the variable **number** as follows

int [ ] number = **new** int[5];

and the computer reserves five storage locations as shown below:

This would cause the array **number** to store the values shown as follows:

number[0]	35
number[1]	40
number[2]	20
number[3]	57
number[4]	19

Creating an Array:

Like other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

1. Declaring the array
2. Creating memory locations
3. Putting values into the memory locations.

Declaring the array:

Arrays in C# are declared as follows:

type[ ] arrayname;

Examples:

```
int[ ] counter;          //declare int array reference

float[ ]  marks;         //declare float array reference
int[ ]    x,y;           //declare two int array reference
```

Remember, we do not enter the size of the arrays in the declaration.

Creation of Arrays:

After declaring an array, we need to create it in the memory. C# allows us to create arrays using **new** operator only, as shown below:

arrayname = new *type*[size];

Examples:

```
number  = new int[5];          //create a 5 element int array
average = new float[10];       // create a 10 element float array
```

It is also possible to combine the two steps, declaration and creation, into one as shown below:

```
Int[ ] number = new int[5]; //declare and create 5 element int array
```

Initialization of Arrays:

The final step is to put values into the array created. This process is known as *initialization*. This is done using the array subscripts as shown below.

arrayname[subscript] = value ;

Example:

```
number[0] = 35;
number[1] = 40;
.....
.....
number[4] = 19;
```

Note that C# creates arrays starting with a subscript of 0 and ends with a value one less than the *size* specified.

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

type [ ] arrayname = {list of values};

Example:

```
int[ ] number = {35, 40, 20, 57, 19};
```

The preceding line is equivalent to:

```
int [ ] number = new int [5] { 35, 40, 20, 57, 19 };
```

This combines all the three steps, namely declaration, creation and initialization.

It is possible to assign an array object to another. For instance,

```
int[ ] a = {1,2,3};
int[ ] b;
b = a;
```

are valid in C#. Both the arrays will have the same values.

Array Length:

In C#, all arrays are class-based and store the allocated size in a variable named **Length.** We can access the length of the array **a** using **a.Length**. *Example:*

```
int aSize = a.Length;
```

Two-dimensional Arrays:

For creating two-dimensional arrays, we must follow the same steps as that of simple arrays. We may create a two-dimensional array like this:

```
int[,] myArray;
myArray = new int[3,4];
```

or

```
int[,] myArray = new int[3,4];
```

This creates a table that can store twelve integer values, four across and three down.

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int[,] table ={{0,0,0},{1,1,1}};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int[,] table = {
                    {0,0,0},
                    {1,1,1}
               };
```

Program 9.2 | APPLICATION OF TWO-DIMENSIONAL ARRAYS

```
using System;
class MulTable
{
        static int ROWS = 20;
        static int COLUMNS = 20;

    public static void Main( )
    {
        int[,] product = new int[ROWS,COLUMNS];
```

```
        Console.WriteLine("MULTIPLICATION TABLE");
        Console.WriteLine("  ");
        int i,j;
        for (i=10; i<ROWS; i++)
        {
            for (j=10; j<COLUMNS; j++)
            {
                product[i,j] = i*j;
                Console.Write("   " +product[i,j]);
            }
            Console.WriteLine("   ");
        }
    }
}
```

Variable size Arrays:

C# treats multidimensional arrays as 'arrays of arrays'. It is possible to declare a two-dimensional array as follows:

```
int[ ][ ] x= new int[3][];        //three rows array
x[0]      = new int[2];           //first row has two elements
x[1]      = new int[4];           //second row has four elements
x[2]      = new int[3];           //third row has three elements
```

The System.Array Class:

In C#, every array we create is automatically derived from the **System.Array** class.

The System.Array Class:

In C#, every array we create is automatically derived from the **System.Array** class.

Table 9.1 *Some commonly used methods of System.Array class*

METHOD/PROPERTY	PURPOSE
Clear ( )	Sets a range of elements to empty values
CopyTo ( )	Copies elements from the source array into the destination array
GetLength ( )	Gives the number of elements in a given dimension of the array
GetValue ( )	Gets the value for a given index in the array
Length	Gives the length of an array
SetValue ( )	Sets the value for a given index in the array
Reverse ( )	Reverses the contents of a one-dimensional array
Sort ( )	Sorts the elements in a one-dimensional array

Program 9.3 illustrates the implementation of **Sort ( )** and **Reverse ( )** methods.

Program 9.3 | SORTING AND REVERSING AN ARRAY

```
using System;
class SortReverse
{
    public static void Main( )
    {
        //creating an array
        int [ ] x = { 30, 10, 80, 90, 20 };
        Console.WriteLine ("Array before sorting");
        foreach (int i in x)
            Console.Write(" " + i );
        Console.WriteLine ( );
        //Sorting and reversing the array elements
        Array.Sort(x); Array.Reverse (x);
        Console.WriteLine("Array after Sorting and Reversing");
        foreach ( int i in x )
            Console.Write(" " + i);
        Console.WriteLine ( );
        }
    }
```

ArrayList class:

An array list is very similar to an array, except that it has the ability to grow dynamically. We can create an array list by indicating the initial capacity we want. *Example:*

```
ArrayList cities = new ArrayList (30);
```

It creates **cities** with a capacity to store thirty objects. If we do not specify the size, it defaults to sixteen. That is,

```
ArrayList cities = new ArrayList ( );
```

will create a **cities** list with the capacity to store sixteen objects. We can now add elements to the list using the **Add ( )** method:

```
cities.Add ("Bombay");
cities.Add ("Anand");
```

We can also remove an element:

```
cities.RemoveAt (1);
```

This will remove the object in position1. We can modify the capacity of the list using the property **Capacity:**

```
cities.Capacity = 20;
```

We may obtain the actual number of objects present in the list using the property **Count** as follows:

```
int n = cities.Count;
```

Table 9.2 *Some important ArrayList methods and properties*

METHODS/PROPERTY	PURPOSE
Add ( )	Adds an object to a list
Clear ( )	Removes all the elements from the list
Contains ( )	Determines if an element is in the list
CopyTo ( )	Copies a list to another
Insert ( )	Inserts an element into the list
Remove ( )	Removes the first occurrence of an element
RemoveAt ( )	Removes the element at the specified place
RemoveRange ( )	Removes a range of elements
Sort ( )	Sorts the elements
Capacity	Gets or sets the number of elements in the list
Count	Gets the number of elements currently in the list.

Program 9.4 demonstrates how an array list may be created and manipulated.

```
using System;
using System.Collections;
class City
{
    public static void Main( )
    {
        ArrayList n = new ArrayList ( );
        n.Add ("Madras");
        n.Add ("Bombay");
        n.Add ("Anand");
        n.Add ("Calcutta");
        n.Add  ("Delhi");
        Console.WriteLine("Capacity = "+ n.Capacity);
        Console.WriteLine("Elements present = " + n.Count);
        n.Sort( );
        for (int i = 0, i < n.Count; i ++)
        {
            Console.WriteLine(n[i]);
        }
        Console.WriteLine( );
        n.RemoveAt(4);
        for (int i = 0 ; i < n.Count ; i ++)
        {
            Console.WriteLine(n[i]);
        }
    }
}
```

Strings

String manipulation is the most common part of many C# programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in C# is by using a character array. For example:

```
char [ ] charArray = new char [4];
charArray [0] = 'N';
charArray [1] = 'a';
charArray [2] = 'm';
charArray [3] = 'e';
```

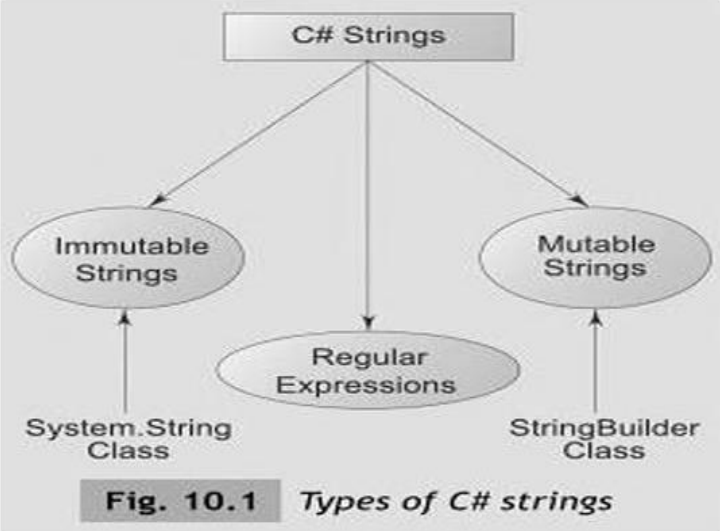


Fig. 10.1 *Types of C# strings*

Creating Strings:

C# supports a predefined reference type known as **string**. We can use **string** to declare **string** type objects. Remember, when we declare a string using **string** type, we are in fact declaring the object to be of type **System.String**, one of the built-in types provided by the .NET Framework. A string type is therefore a **System.String** type as well.

We can create immutable strings using **string** or **String** objects in a number of ways.

- Assigning string literals
- Copying from one object to another
- Concatenating two objects
- Reading from the keyboard
- Using **ToString** method

10.2.1 Assigning String Literals

The most common way to create a string is to assign a quoted string of characters known as *string literal* to a string object. For example:

```
string s1;-----//declaring a string object
s1 = "abc";--  //assigning string literal
```

Both these statements may be combined into one as follows:

```
string s1 = "abc"; //declaring and assigning
```

10.2.2 Copying Strings

We can also create new copies of existing strings. This can be accomplished in two ways:

- Using the overloaded = operator
- Using the static **Copy** method

Example:

```
string s2 = s1;                 //assigning
string s2 = string.Copy(s1);    //copying
```

Both these statements would accomplish the same thing, namely, copying the contents of **s1** into **s2.**

10.2.3 Concatenating Strings

We may also create new strings by concatenating existing strings. There are a couple of ways to accomplish this.

- Using the overloaded + operator
- Using the static **Concat** method

Examples:

```
string s3 = s1 + s2; //s1 and s2 exist already
string s3 = string.Concat(s1, s2)
```

If s1 = 'abc' and s2 = 'xyz', then both the statements will store the string 'abcxyz' in s3. Note that the contents of s2 is simply appended to the contents of s2 and the result is stored in s3.

10.2.4 Reading from the Keyboard

It is possible to read a string value interactively from the keyboard and assign it to a string object.

```
string s = Console.ReadLine( );
```

10.2.5 The ToString Method

Another way of creating a string is to call the **ToString** method on an object and assign the result to a string variable.

```
int number = 123;
string numStr = number.ToString( );
```

String Methods:

String objects are ***immutable***, meaning that we cannot modify the characters contained in them. However, since the string is an alias for the predefined **System.String** class in the Common Language Runtime (CLR), there are many built-in operations available that work with strings. All operations produce a modified version of the string rather than modifying the string on which the method is called.

Table 10.1 *String class methods*

METHOD	OPERATION
Compare ( )	Compares two strings
CompareTo ( )	Compares the current instance with another instance
ConCat ( )	Concatenates two or more strings
Copy( )	Creates a new string by copying another
CopyTo ( )	Copies a specified number of characters to an array of Unicode characters
EndsWith ( )	Determines whether a substring exists at the end of the string
Equals( )	Determines if two strings are equal

IndexOf ( )	Returns the position of the first occurrence of a substring
Insert ( )	Returns a new string with a substring inserted at a specified location
Join ( )	Joins an array of strings together
LastIndexOf ( )	Returns the position of the last occurrence of a substring
PadLeft ( )	Left-aligns the strings in a field
PadRight ( )	Right-aligns the string in a field
Remove ( )	Deletes characters from the string
Replace ( )	Replaces all instances of a character with a new character
Split ( )	Creates an array of strings by splitting the string at any occurrence of one
StartsWith ( )	Determines whether a substring exists at the beginning of the string
Substring ( )	Extracts a substring
ToLower ( )	Returns a lower-case version of the string
ToUpper ( )	Returns an upper-case version of the string
Trim ( )	Removes white space from the string
TrimEnd ( )	Removes a string of characters from the end of the string
TrimStart ( )	Removes a string of characters from the beginning of the string

Inserting Strings:

String methods are called using the string object on which we want to work. Program 10.1 illustrates the use of the **Insert ( )** method and the indexer property supported by the **System.String** class.

Program 10.1 | USING THE INSERT ( ) METHOD

```
using System;
class StringMethod
{
        public static void Main( )
        {
            string s1 = "Lean";
        string s2 = s1.Insert (3,"r");          // s2 = Learn
        string s3 = s2.Insert (5,"er");         // s3 = Learner
            for (int i = 0; i < s3.Length; i++)
                Console.Write(s3[i]);
            Console.Writeline( );
        }
}
```

Comparing Strings:

String class supports overloaded methods and operators to compare whether two strings are equal or not. They are:

- Overloaded **Compare( )** method
- Overloaded **Equals( )** method
- Overloaded == operator

10.5.1 Compare( ) Method

There are two versions of overloaded static **Compare** method. The first one takes two strings as parameters and compares them. *Example:*

```
int n = string.Compare(s1,s2);
```

This performs a case-sensitive comparison and returns different integer values for different conditions as under:

- Zero integer, if s1 is equal to s2
- A positive integer (1), if s1 is greater than s2
- A negative integer (–1), if s1 is less than s2

For example, if s1 = "abc" and s2 = "ABC", then n will be assigned a value of –1. Remember, a lowercase letter has a smaller ASCII value than an uppercase letter.

10.5.2 Equals( ) Method

The **string** class supports an overloaded **Equals** method for testing the equality of strings. There are again two versions of **Equals** method. They are implemented as follows:

```
bool b1 = s2.Equals(s1);
bool b2 = string.Equals (s2, s1);
```

These methods return a Boolean value **true** if s1 and s2 are equal, otherwise **false**.

10.5.3 The == Operator

A simple and natural way of testing the equality of strings is by using the overloaded = = operator.

Example:

```
bool b3 = (s1 == s2); //b3 is true if they are equal
```

Finding Substrings:

It is possible to extract substrings from a given string using the overloaded **Substring** method available in **String** class. There are two version of **Substring:**

- s.Substring(n)
- s.Substring(n1, n2)

The first one extracts a substring starting from the nth position to the last character of the string contained in **s**. The second one extracts a substring from **s** beginning at n1 position and ending at n2 position. *Examples:*

```
string s1 = "NEW YORK";
string s2 = s1.Substring(5);
string s3 = s1.Substring(0,3);
string s4 = s1.Substring(5,8);
```

When executed, the string variables will contain the following substrings:

```
s2:     YORK
s3:     NEW
s4:     YORK
```

Mutable Strings:

Mutable strings that are modifiable can be created using the **StringBuilder** class. *Examples:*

```
StringBuilder str1 = new StringBuilder("abc");
StringBuilder str2 = new StringBuilder ( );
```

The string object **str1** is created with an initial size of three characters and **str2** is created as an empty string. They can grow dynamically as more characters are added to them. They can grow either unbounded or up to a configurable maximum. Mutable strings are also known as *dynamic strings*.

Table 10.2 *Some useful stringBuilder methods*

METHOD	OPERATION
Append ( )	Appends a string
AppendFormat ( )	Appends strings using a specific format
EnsureCapacity ( )	Ensures sufficient size
Insert ( )	Inserts a string at a specified position
Remove ( )	Removes the specified characters
Replace ( )	Replaces all instances of a character with a specified one

Table 10.3 *StringBuilder properties*

PROPERTY	PURPOSE
Capacity	To retrieve or set the number of characters the object can hold
Length	To retrieve or set the length
MaxCapacity	To retrieve the maximum capacity of the object
[ ]	To get or set a character at a specified position

Program 10.3 | **USING STRINGBUILDER METHODS**

```
using System.Text;   //For using StringBuilder
using System;
class StringBuilderMethod
{
     public static void Main( )
     {
          StringBuilder s = new StringBuilder          ("Object ");
          Console.WriteLine("Original string           : " + s);
          Console.WriteLine("Length                    : " + s.Length);
          //Appending a string
          s.Append("language ");
          Console.WriteLine("String now                : " + s);
          //Inserting a string
          s.Insert (7,"oriented ");
          Console.WriteLine("Modified string           : " + s);
          //Setting a character
          int n = s.Length;
          s[n-1] = '!' ;
          Console.WriteLine("Final string              : " + s);
     }
}
```

Look at the output produced by Program 10.3:

```
Original string    :    Object
Length             :    7
String now         :    Object language
Modified string    :    Object oriented language
Final string       :    Object oriented language!
```

Arrays of Strings:

We can also create and use arrays that contain strings. The statement

```
string [ ] itemArray = new string [3];
```

will create an **itemArray** of size 3 to hold three strings. We can assign the strings to the **itemArray** element by element using three different statements, or more efficiently using a **for** loop. We could also provide an array with a list of initial values in curly braces:

```
string [ ] itemArray = {"Java", "C++", "Csharp"};
```

The size of the array is determined by the number of elements in the initialization list. The size of the array, once created, cannot be changed.

If we want an array whose length is determined dynamically or an array which can be extended at run time, we have to use the **ArrayList** class to create a list.

Program 10.5 | MANIPULATING STRING ARRAYS

```
using System;
class Strings
{
      public static void Main( )
      {
            string[ ]countries = {"India", "Germany",  "America","France"};
      int n = countries.Length;
      //Sort alphabetically
      Array.Sort(countries);
      for (int i = 0; i < n; i++)
      {
            Console.WriteLine(countries[i]);
      }
      Console.WriteLine( );
      //Reverse the array elements
      Array.Reverse(countries);
      For (int i = 0; i < n; i++)
      {
            Console.WriteLine(countries[i]);
         }
      }
}
```

Regular Expressions:

Regular expressions provide a powerful tool for searching and manipulating a large text. A regular expression may be applied to a text to accomplish tasks such as:

- To locate substrings and return them
- To modify one or more substrings and return them
- To identify substrings that begin with or end with a pattern of characters
- To find all words that begin with a group of characters and end with some other characters
- To find all the occurrences of a substring pattern, and

A regular expression (also known as a *pattern string)* is a string containing two types of characters.

- Literals
- Metacharacters

Expression	*Meaning*
"\bm"	Any word beginning with m
"er\b"	Any word ending with er.
"\BX\B"	Any X in the middle of a word
"\bm\S*er\b"	Any word beginning with m and ending with er.
" |,"	Any word separated by a space or a comma

Example:

```
using System;
using System.Text;   //for StringBuilder class
using System.Text.RegularExpressions;  //for Regex class
Class RegexTest
{
      public static void Main ( )
      {
            string str;
            str = "Amar, Akbar, Antony are friends!";
            Regex reg = new Regex (" |, ");
            StringBuilder sb = new StringBuilder( );
            int count = 1;
      foreach(string sub in reg.Split(str))
      {
            sb.AppendFormat("{0}: {1}\n", count++, sub);
      }
      Console.WriteLine(sb);
}
}
```

Structures and Enumerations

C# allows us to define our own complex value types (known as user-defined value types) based on these simple data types. There are two sorts of value types we can define in C#:

- Structures
- Enumerations

Structures:

Structures (often referred to as *structs*) are similar to classes in C#. Although classes will be used to implement most objects, it is desirable to use structs where simple composite data types are required. Because they are value types stored on the stack, they have the following advantages compared to class objects stored on the heap:

- They are created much more quickly than heap-allocated types.
- They are instantly and automatically deallocated once they go out of scope.
- It is easy to copy value type variables on the stack.

11.2.1 Defining a Struct:

Structs are declared using the **struct** keyword. The simple form of a struct definition is as follows:

```
struct struct-name
{
    data member1;
    data member2;

        . . .
        . . .
}
```

Example:

```
struct Student
{
    public string Name;
    public int RollNumber;
    public double TotalMarks;
}
```

The keyword **struct** declares **Student** as a new data type that can hold three variables of different data Types. These variables are known as *members* or *fields* or *elements*. The identifier **Student** can now be used to create variables of type **Student**. *Example:*

```
Student s1 ;      //declare a student
```

11.2.2 Assigning Values to Members

Member variables can be accessed using the simple dot notation as follows:

```
s1.Name          =    "John";
s1.RollNumber    =    999 ;
s1.TotalMarks    =    575.50 ;
```

We may also use the member variables in expressions on the right-hand side. *Example:*

```
FinalMarks       =    s1.TotalMarks + 5.0 ;
```

11.2.3 Copying Structs

We can also copy values from one struct to another. *Example:*

```
Student s2; // s2 is declared
s2 = s1 ;
```

This will copy all those values from s1 to s2.

Program 11.1 | **A SIMPLE APPLICATION OF STRUCTS**

```
using System;
struct Item
{
     public string name;
     public int code;
     public double price;
}
class StructTest
{
     public static void Main( )
     {
          Item fan;     //create an item
          //Assign values to members
          fan.name  = "Bajaj";
          fan.code  =  123;
          fan.price =  1576.50;
          //Display item details
          Console.WriteLine("Fan name: " + fan.name);
          Console.WriteLine("Fan code: " + fan.code);
          Console.WriteLine("Fan cost: " + fan.price);
     }
}
```

Output of Program 11.1:

```
Fan name     :  Bajaj
Fan code     :  123
Fan cost     :  1576.5
```

Structs with Methods:

We have seen that values may be assigned to the data members using **struct** objects and the dot operator. We can also assign values to the data members using what are known as *constructors*.

A constructor is a method which is used to set values of data members at the time of declaration. Consider the code below:

```
sruct Number
{
     int number;                        // data member
     public Number ( int value )        // constructor
     {
          number = value;
     }
}
```

The constructor method has the same name as **struct** and declared as public. The constructor is invoked as follows:

```
Number n1 = new Number(100);
```

Program 11. 2 | **USING METHODS IN STRUCTS**

```
using System;
struct Rectangle
{
     int a, b;
     public Rectangle ( int x, int y )   //constructor
     {
          a =  x;
          b = y;
     }
     public int Area( )                  //a method
     {
          return ( a * b);
     }
     public void Display ( )             //another method
     {
          Console.WriteLine("Area = " + Area( ) );
     }
}
class TestRectangle
{
     public static void Main( )
     {
          Rectangle rect = new Rectangle ( 10, 20 );
          rect.Display ( );   // invoking Display ( ) method
     }
}
```

Program 11.2 produces the following output:

```
Area = 200
```

Differences between Structs and Classes:

Table 11.1 *Class and struct differences*

CATEGORY	CLASSES	STRUCTS
Data Type	Reference type and therefore stored on the heap	Value type and therefore stored on the stack. Behave like simple data types
Inheritance	Support inheritance	Do not support inheritance
Default values	Default value of a class type is null	Default value is the value produced by 'zeroing out' the fields of the struct
Field initialization	Permit initialization of instance fields	Do not permit initialization of instance fields
Constructors	Permit declaration of parameterless constructors	Do not permit declaration of parameterless constructors
Destructors	Supported	Not supported
Assignment	Assignment copies the reference	Assignment copies the values.

Enumerations

An enumeration is a user-defined integer type which provides a way for attaching names to numbers, thereby increasing the comprehensibility of the code. The **enum** keyword automatically enumerates a list of words by assigning them values 0, 1, 2 and so on. This facility provides an alternative means of creating 'constant' variable names. The syntax of an **enum** statement is illustrated below:

```
enum Shape
{
    Circle,          //ends with comma
    Square,          //ends with comma
    Triangle         //no comma
}
```

This can be written in one line as follows:

```
enum Shape {Circle, Square, Triangle}
```

Program 11.4 | IMPLEMENTING ENUM TYPE

```
using System;
class Area
{
    public enum Shape
    {
        Circle,
        Square
    }
    public void AreaShape ( int x, Shape shape)
    {
        double area;
        switch (shape)
        {
            case    Shape.Circle:
                    area = Math.PI * x * x;
                    Console.WriteLine("Circle Area = "+area);
                    break;
            case    Shape.Square:
                    area = x * x ;
                    Console.WriteLine("Square Area = "+area);
                            break;
                    default:
                        Console.WriteLine("Invalid Input"); break;
        }
```

```
        }
}


class EnumTest:
{
    public static void Main( )
    {

        Area area = new Area ( );
        area.AreaShape ( 15, Area.Shape.Circle);
        area.AreaShape ( 15, Area.Shape.Square);
        area.AreaShape ( 15, (Area.Shape) 1 );
        area.AreaShape ( 15, (Area.Shape) 10 );
    }
}
```

Program 11.4 will produce the following output:

```
Circle Area     =     706.8583
Square Area     =     225.00000
Square Area     =     225.00000
Invalid Input
```

Enumerator Initialization:

by default, the value of the first enum member is set to 0, and that of each subsequent member is incremented by one. However, we may assign specific values for different members, if we so desire.

Example:

```
enum Colour
{
    Red    = 1,
    Blue   = 3,
    Green  = 7,
    Yellow = 5
}
```

We can also have expressions, as long as they use the already defined enum members.

Example:

```
enum Colour
{


    Red      = 1,
    Blue     = Red + 2,
    Green    = Red + Blue + 3,
    Yellow   = Blue + 2
}
```