18MCA42C

.NET PROGRAMMING (C#)

UNIT I: Introduction To .NET and C#:

FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

Year	Subject Title	Semester	Sub. Code
2018 - 2019 Onwards	.NET PROGRAMMING (C#)	IV	18MCA42C

Objective: To provide good understanding of the role of Web Development using .net with C#. After the successful completion of the course, the students should have the thorough knowledge on Web development.

UNIT I: Introduction To .NET and C#: Common Language Runtime, .NET frame work, Microsoft Intermediate Languages, Jitters, Unmanaged code. Evolution of C#, Characteristics of C#, how does C# differ from C++ and Java, Data types, Variables and Literals, Boxing and unboxing, Operators and Expressions, Type conversions, Mathematical functions, Decision making and branching, Decision making and looping.

UNIT II: Object Oriented Programming In C#: Methods, Classes and objects, access specifier, Inheritance, abstract class, sealed classes, interfaces, delegates, namespaces, exceptions.

UNIT III: Advanced Features Of C#: Serializing objects, deserialization, XML based serialization, Multi-threading, Reflection Attributes, Properties and Indexers.

UNIT IV: Window Based Programming: Win Forms, Textbox, Buttons, Message Box, List Box, Handling events.

UNIT V: ADO .NET: ADO.Net Object Model - Connecting with database, retrieving results, updating data in database, Deletion. **ASP.NET Using C#:** Web Application Project, Web Forms, Controls.

TEXT BOOKS:

1. E. Balagurusamy, "Programming in C#" TMH, 2006.
2. Ian Griffiths, Matthew Adams and Jesse Liberty, "Programming C# 4.0" O'Reilly Sixth Edition.

REFERENCE BOOKS:

1. Stanley B.Lippman, "C# Primer A Practical Approach", Pearson Education, 2002.
2. Tom archer, "Inside C#", Microsoft Press, 2001.
3. "Microsoft C# Language Specification", Microsoft Press, 2001.

Introduction to .NET Framework

- **.NET** is a software framework which is designed and developed by Microsoft.
- The first version of .Net framework was 1.0 which came in the year 2002.
- It is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc.
- It is used to develop Form-based applications, Web-based applications, and Web services.
- There is a variety of programming languages available on the .Net platform like VB.Net and C# etc.,.
- It is used to build applications for Windows, phone, web etc. It provides a lot of functionalities and also supports industry standards.
- .NET Framework supports more than 60 programming languages in which 11 programming languages are designed and developed by Microsoft.

11 Programming Languages which are designed and developed by Microsoft are:

- C#.NET
- VB.NET
- C++.NET
- J#.NET
- F#.NET
- JSCRIPT.NET
- WINDOWS POWERSHELL
- IRON RUBY
- IRON PYTHON
- C OMEGA
- ASML(Abstract State Machine Language)

Main Components of .NET Framework

1.Common Language Runtime(CLR):

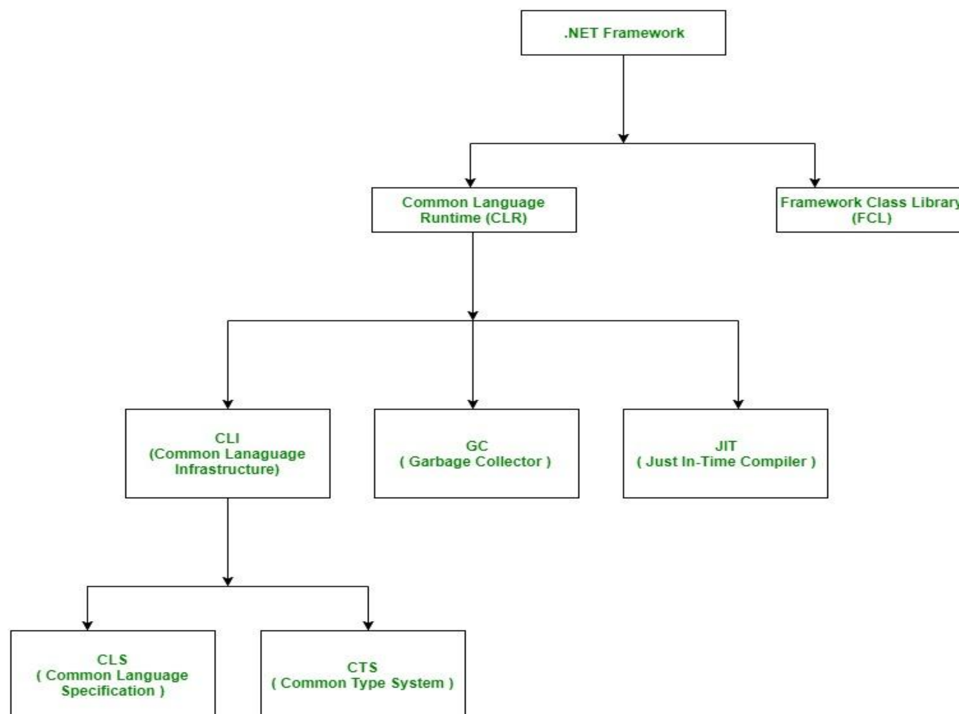
- CLR is the basic and Virtual Machine component of the .NET Framework.

- It is the run-time environment in the .NET Framework that runs the codes and helps in making the development process easier by providing the various services such as remoting, thread management, type-safety, memory management, robustness etc..
- Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language.
- It also helps in the management of code, as code that targets the runtime is known as the Managed Code and code doesn't target to runtime is known as Unmanaged code.

2. Framework Class Library (FCL):

- It is the collection of reusable, object-oriented class libraries and methods etc that can be integrated with CLR.
- Also called the Assemblies.
- It is just like the header files in C/C++ and packages in the java.
- Installing .NET framework basically is the installation of CLR and FCL into the system.

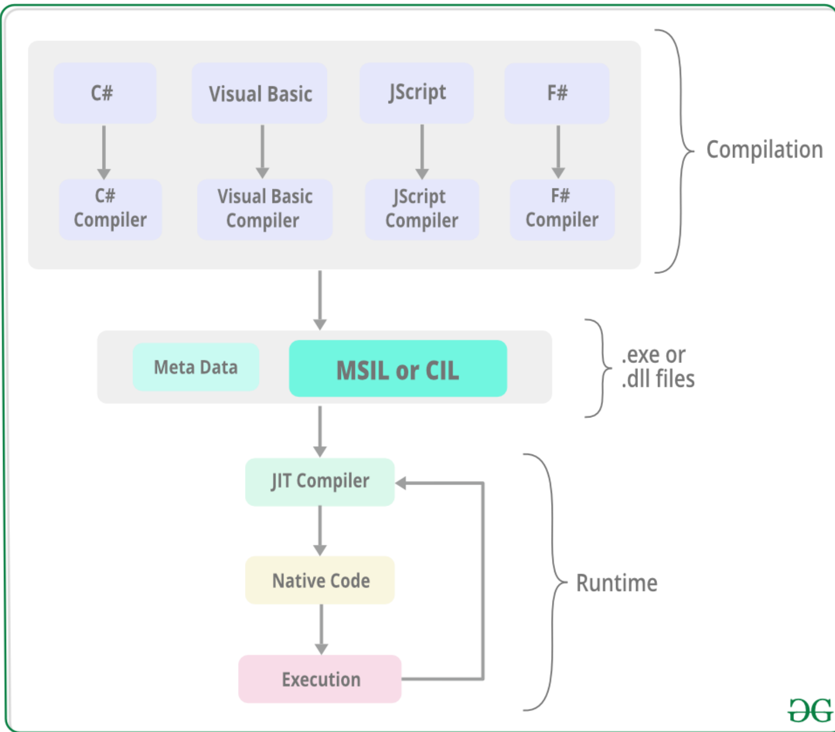
Overview of .NET Framework



CIL or MSIL | Microsoft Intermediate Language or Common Intermediate Language

- The Microsoft Intermediate Language (MSIL), also known as the Common Intermediate Language (CIL) is a set of instructions that are platform independent and are generated by the language-specific compiler from the source code.
- The MSIL is platform independent and consequently, it can be executed on any of the Common Language Infrastructure supported environments such as the Windows *.NET* runtime.
- The MSIL is converted into a particular computer environment specific machine code by the JIT compiler.
- This is done before the MSIL can be executed.
- Also, the MSIL is converted into the machine code on a requirement basis i.e. the JIT compiler compiles the MSIL as required rather than the whole of it.

Execution process in Common Language Runtime (CLR): The execution process that includes the creation of the MSIL and the conversion of the MSIL into machine code by the JIT compiler is given as follows:

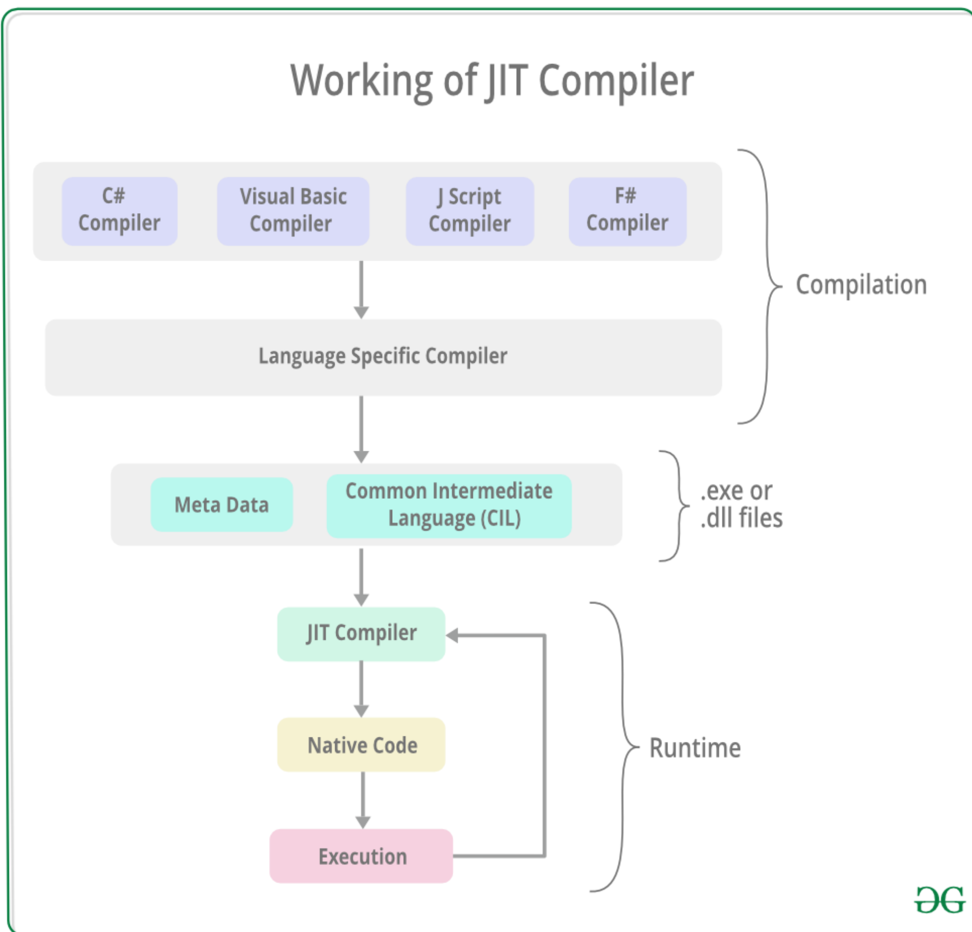


- The source code is converted into the MSIL by a language-specific compiler in the compile time of the CLR.
- Also, along with the MSIL, metadata is also produced in the compilation.
- The metadata contains information such as the definition and signature of the types in the code, runtime information, etc.
- A Common Language Infrastructure (CLI) assembly is created by assembling the MSIL. This assembly is basically a compiled code library that is used for security, deployment, versioning, etc. and it is of two types i.e. process assembly (EXE) and library assembly (DLL).
- The JIT compiler then converts the Microsoft Intermediate Language(MSIL) into the machine code that is specific to the computer environment that the JIT compiler runs on. The MSIL is converted into the machine code on a requirement basis i.e. the JIT compiler compiles the MSIL as required rather than the whole of it.
- The machine code obtained using the JIT compiler is then executed by the processor of the computer.

Just-In-Time(JIT) Compiler in .NET

- Just-In-Time compiler(JIT) is a part of **Common Language Runtime (CLR)** in *.NET*
- It is responsible for managing the execution of *.NET* programs regardless of any *.NET* programming language.
- A language-specific compiler converts the source code to the intermediate language.
- This intermediate language is then converted into the machine code by the Just-In-Time (JIT) compiler.
- This machine code is specific to the computer environment that the JIT compiler runs on.
- The JIT compiler is required to speed up the code execution and provide support for multiple platforms.

Working of JIT Compiler:



Types of Just-In-Time Compiler

There are **3** types of JIT compilers

1. Pre-JIT Compiler:

- All the source code is compiled into the machine code at the same time in a single compilation cycle using the Pre-JIT Compiler.
- This compilation process is performed at application deployment time.

2. **Normal JIT Compiler:**

- The source code methods that are required at run-time are compiled into machine code the first time they are called by the Normal JIT Compiler.
- After that, they are stored in the cache and used whenever they are called again.

3. **Econo JIT Compiler:**

- The source code methods that are required at run-time are compiled into machine code by the Econo JIT Compiler.
- After these methods are not required anymore, they are removed.

Advantages of JIT Compiler

- The JIT compiler requires **less memory usage** as only the methods that are required at run-time are compiled into machine code by the JIT Compiler.
- **Page faults are reduced** by using the JIT compiler as the methods required together are most probably in the same memory page.
- **Code optimization based on statistical analysis** can be performed by the JIT compiler while the code is running.

Disadvantages of JIT compiler:

- The JIT compiler requires **more startup time** while the application is executed initially.
- The **cache memory is heavily used by the JIT compiler** to store the source code methods that are required at run-time.

Note: Much of the disadvantages of the JIT compiler can be handled using the Ahead-of-time (AOT) compilation. This involves compiling the MSIL into machine code so that runtime compilation is not required and the machine code file can be executed natively.

Difference between Managed and Unmanaged code in .NET

Managed code is the code which is managed by the CLR(Common Language Runtime) in *.NET Framework*. Whereas the Unmanaged code is the code which is directly executed by the operating system. Below are some important differences between the Managed code and Unmanaged code:

MANAGED CODE	UNMANAGED CODE
It is executed by managed runtime environment or managed by the CLR.	It is executed directly by the operating system.
It provides security to the application written in .NET Framework.	It does not provide any security to the application.
Memory buffer overflow does not occur.	Memory buffer overflow may occur.
It provide runtime services like Garbage Collection, exception handling, etc.	It does not provide runtime services like Garbage Collection, exception handling, etc.
The source code is compiled in the intermediate language known as *IL or MSIL or CIL*.	The source code direclty compiled into native langugae.
It does not provide low-level access to the programmer.	It provide low-level access to the programmer.

Introduction to C#

- It is a general-purpose, modern and object-oriented programming language pronounced as "C Sharp".
- It was developed by Microsoft led by Anders Hejlsberg and his team within the *.NET* initiative and was approved by the European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO).
- C# is among the languages for Common Language Infrastructure.
- C# is a lot similar to Java syntactically and is easy for users who have knowledge of C, C++ or Java.

characteristics of c#

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#

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

EVOLUTION OF C# -

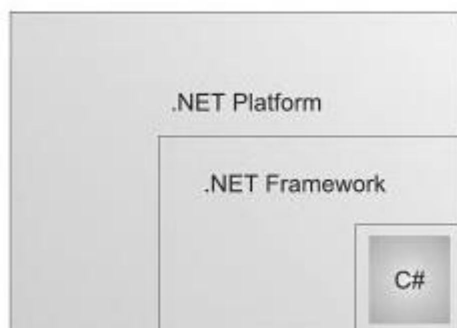


Fig. 1.2 *C# inside the .NET*

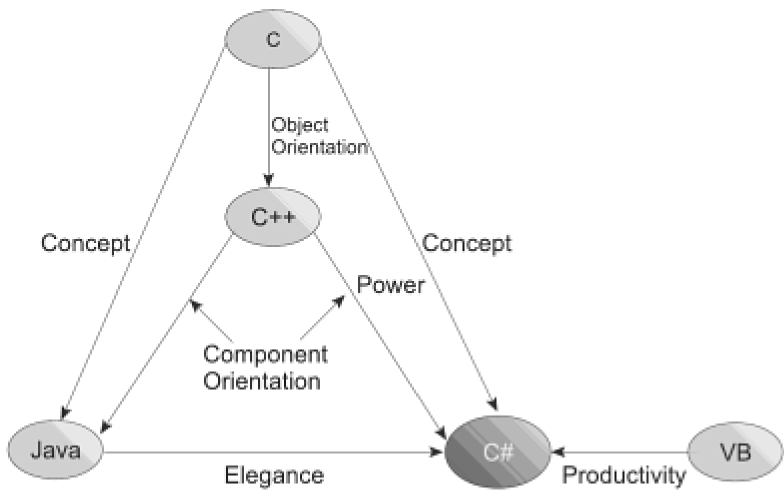


Fig. 1.3 *Evaluation of C# language*

HOW DOES C# DIFFER FROM C++?

Changes Introduced

01. C# compiles straight from source code to executable code, with no object files.
02. C# does not separate class definition from implementation. Classes are defined and implemented in the same place and therefore there is no need for header files.
03. In C#, class definition does not use a semicolon at the end.
04. The first character of the **Main( )** function is capitalized. The **Main** must return either **int** or **void** type value.
05. C# does not support #**include** statement. (Note that **using** is not the same as #**include).**
06. All data types in C# are inherited from the **object** super class and therefore they are objects.
07. All the basic value types will have the same size on any system. This is not the case in C or C++. Thus C# is more suitable for writing distributed applications.
08. In C#, data types belong to either **value types** (which are created in a stack) or **reference types** (which are created in a heap).
09. C# checks for uninitialized variables and gives error messages at compile time. In C++, an uninitialized variable goes undetected thus resulting in unpredictable output.
10. In C#, structs are value types.
11. C# supports a native string type. Manipulation of strings are easy.
12. C# supports a native Boolean data type and bool-type data cannot be implicitly or explicitly cast to any data type except object.
13. C# declares **null** as a keyword and considers it as an intrinsic value.
14. C# does not support pointer types for manipulating data. However, they are used in what is known as 'unsafe' code.
15. Variable scope rules in C# are more restrictive. In C#, duplicating the same name within a routine is illegal, even if it is in a separate code block.
16. C# permits declaration of variables between **goto** and label.
17. We can only create objects in C# using the **new** keyword.
18. Arrays are classes in C# and therefore they have built-in functionality for operations such as sorting, searching and reversing.
19. Arrays in C# are declared differently and behave very differently compared to C++ arrays.
20. C# provides special syntax to initialize arrays efficiently.
21. Arrays in C# are always reference types rather than value types, as they are in C++ and therefore stored in a heap.
22. In C#, expressions in **if** and **while** statements must resolve to a **bool** value. Accidental use of the assignment operator (=) instead of equality operator = = will be caught by the compiler.
23. C# supports four iteration statements rather than three in C++ . The fourth one is the **foreach** statement.
24. C# does not allow silent fall-through in switch statements. It requires an explicit jump statement at the end of each case statement.
25. In C#, **switch** can also be used on string values.
26. C# does not support the labeled break and therefore jumping out of nested loops can be messy.

27. The set of operators that can be overloaded in C# is smaller compared to C++.
28. C# can check overflow of arithmetic operations and conversions using **checked** and **unchecked** keywords.
29. C# does not support default arguments.
30. Variable method parameters are handled differently in C#.
31. In exception-handling, unlike in C++, we cannot throw any type in C#. The thrown value has to be a reference to a derived class or S**ystem.Exception** object.
32. C# requires ordering of **catch** blocks correctly.
33. General catch statement **catch (…)** in C++ is replaced by simple **catch** in C#.
34. C# does not provide any defaults for constructors.
35. Destructors in C# behave differently than in C++.
36. In C#, we cannot access static members via an object, as we can in C++.
37. C# does not support multiple code inheritance.
38. Casting in C# is much safer than in C++.
39. When overriding a virtual method, we must use the **override** keyword.
40. Abstract methods in C# are similar to virtual functions in C++, but C# abstract methods cannot have implementations.
41. Command-line parameters array behave differently in C# as compared to C++.

C++ features dropped

The following C++ features are missing from C#:

1. Macros
2. Multiple inheritance
3. Templates
4. Pointers
5. Global variables
6. **Typedef** statement
7. Default arguments
8. Constant member functions or parameters
9. Forward declaration of classes

Enhancements to C++

C# modernizes C++ by adding the following new features:

1. Automatic garbage collection
2. Versioning support
3. Strict type-safety
4. Properties to access data members
5. Delegates and events
6. Boxing and unboxing
7. Web services

HOW DOES C# DIFFER FROM JAVA ?

01. Although C# uses .NET runtime that is similar to Java runtime, the C# compiler produces an executable code.
02. C# has more primitive data types.
03. Unlike Java, all C# data types are objects.
04. Arrays are declared differently in C#.
05. Although C# classes are quite similar to Java classes, there are a few important differences relating to constants, base classes and constructors, static constructors, versioning, accessibility of members etc.
06. Java uses **static final** to declare a class constant while C# uses **const.**
07. The convention for Java is to put one public class in each file and in fact, some compilers require this. C# allows any source file arrangement.
08. C# supports the **struct** type and Java does not.
09. Java does not provide for operator overloading.
10. In Java, class members are virtual by default and a method having the same name in a derived class overrides the base class member. In C#, the base member is required to have the **virtual** keyword and the derived member is required to use the **override** keyword.
11. The **new** modifier used for class members has no complement in Java.
12. C# provides better versioning support than Java.
13. C# provides **static** constructors for initialization.
14. C# provides built-in delegates and events. Java uses interfaces and inner classes to achieve a similar result.
15. In Java, parameters are always passed by value. C# allows parameters to be passed by reference by using the **ref** keyword.
16. C# adds **internal**, a new accessibility modifier. Members with internal accessibility can be accessed from other classes within the same project, but not from outside the project.
17. C# includes native support for properties, Java does not.
18. Java does not directly support enumerations.
19. Java does not have any equivalent to C# indexers.
20. Both Java and C# support interfaces. But, C# does not allow type definitions in interfaces, while Java interfaces can have **const** type data.
21. In Java, the **switch** statement can have only integer expression, while C# supports either an integer or string expressions.
22. C# does not allow free fall_through from case to case.
23. C# provides a fourth type of iteration statement, **foreach** for quick and easy iterations over collections and array type data.
24. Catch blocks should be ordered correctly in C#.
25. C# checks overflows using **checked** statements.
26. C# uses **is** operator instead of **instanceof** operator in Java.
27. C# allows a variable number of parameters using the **params** keyword.
28. There is no labeled **break** statement in C#. The **goto** is used to achieve this.

Literals, Variables and Data Types

C# keywords

ct	event	namespace	static
	explicit	new	string
	extern	null	struct
	false	object	switch
	finally	operator	this
	fixed	out	throw
	float	override	true
	for	params	try
	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

Literals are the way in which the values that are stored in variables are represented. We shall discuss these in detail in the next section.

Operators are symbols used in expressions to describe operations involving one or more operands. Operators are considered in detail in Chapter 5.

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

Literals

Literals are value constants assigned to variables (or results of expressions) in a program. C# supports several types of literals as illustrated in Fig. 4.1.

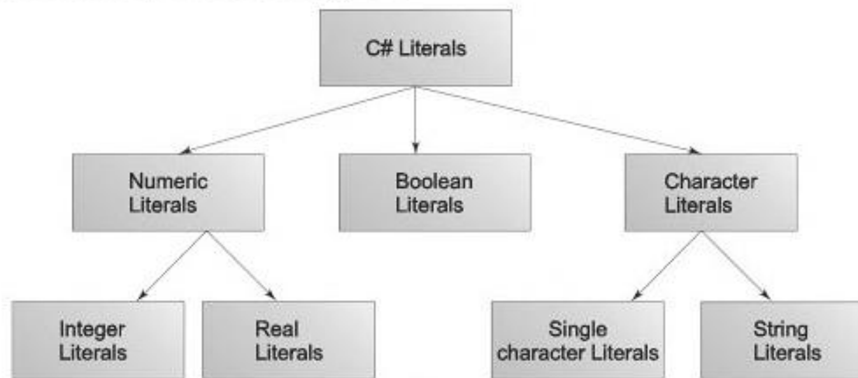


Fig. 4.1 *C# literals*

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

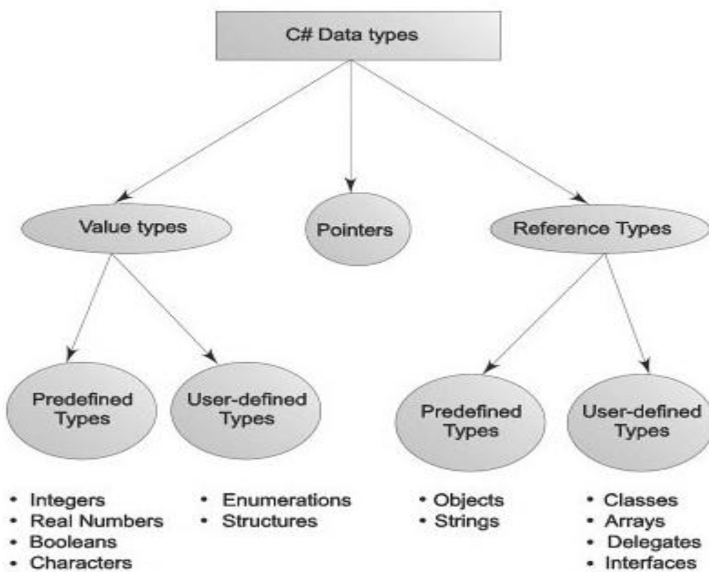


Fig. 4.2 *Taxonomy C# data types*

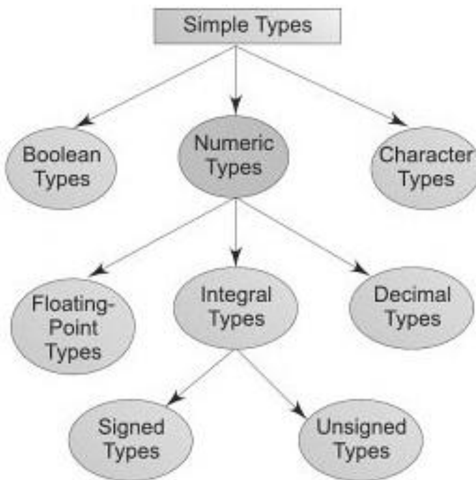


Fig. 4.3 *Categories of simple type data*

Boxing and Unboxing

In object-oriented programming, methods are invoked using objects. Since value types such as **int** and **long** are not objects, we cannot use them to call methods. C# enables us to achieve this through a technique known as *boxing* . Boxing means the conversion of a value type on the stack to a **object** type on the heap. Conversely, the conversion from an **object** type back to a value type is known as *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.

Operators and Expressions

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

FLOATING-POINT ARITHMETIC

```
class FloatPoint
{
      public static void Main()
      {
          float a = 20.5F, b = 6.4F;
          System.Console.WriteLine("  a = " + a);
          System.Console.WriteLine("  b = " + b);
          System.Console.WriteLine("  a+b = " + (a+b));
          System.Console.WriteLine("  a-b = " + (a-b));
          System.Console.WriteLine("  a*b = " + (a*b));
          System.Console.WriteLine("  a/b = " + (a/b));
          System.Console.WriteLine("  a%b = " + (a%b));
      }
}
```

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*

*only if the sum of the values of a and b is equal to the sum of the values of c and d.

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));
      }
}
```

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 --

The operator ++ adds 1 to the operand while –– subtracts 1. Both are unary operators and are used in the following form:

```
++m; or m++;
--m; or m--;
++m; is equivalent to m = m + 1; (or m + = 1;)
--m; is equivalent to m = m - 1; (or m - = 1;)
```

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);
    }
}
```

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

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;

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**."

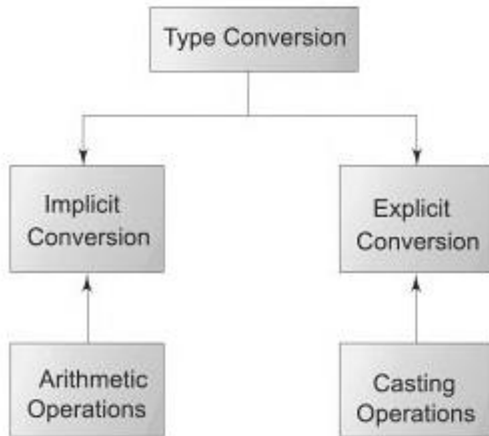


Fig. 5.1 *Type conversions in C#*

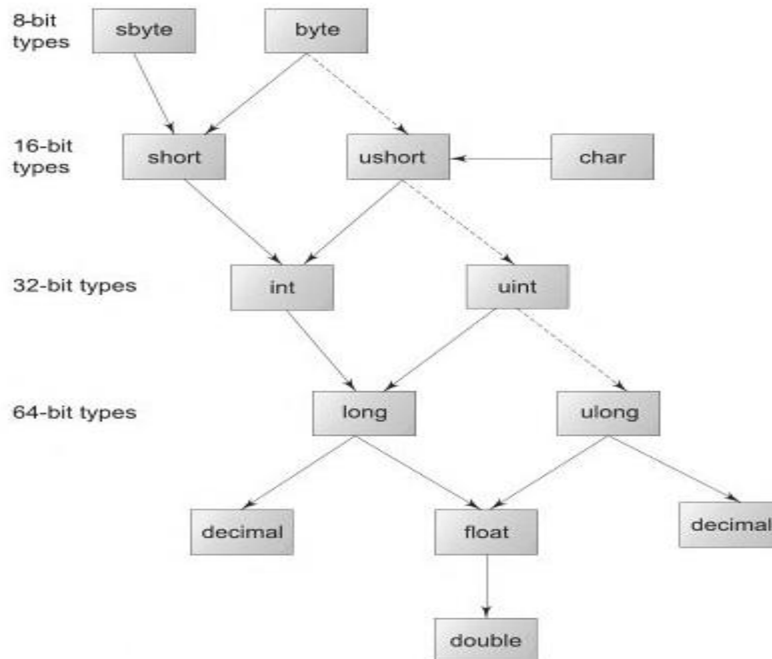


Fig. 5.2 *C# conversion hierarchy chart*

MATHEMATICAL FUNCTIONS

Table 5.11 *Main mathematical methods*

METHOD	DESCRIPTION
Sin ( )	sine of an angle in radians
Cos ( )	cosine of an angle in radians
Tan ( )	tangent of an angle in radians
Asin ( )	inverse of sine
Acos ( )	inverse of cosine
Atan ( )	inverse of tangent
Atan2 ( )	inverse tangent, with x and y co-ordinates specified
Sinh ( )	hyperbolic sine
Cosh ( )	hyperbolic cosine
Tanh ( )	hyperbolic tangent
Sqrt ( )	square root
Pow ( )	number raised to a given power
Exp ( )	exponential
Log ( )	natural logarithm (base e)
Log10 ( )	base 10 logarithm
Abs ( )	absolute value of a number
Min ( )	lower of two numbers
Max ( )	higher of two numbers
Sign ( )	sign of the number

Decision Making and Branching

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 is a powerful decision-making statement and is used to control the flow of execution of statements. It is a *two-way* decision statement and is used in conjunction with an expression. It takes the following form:

If*(expression)*

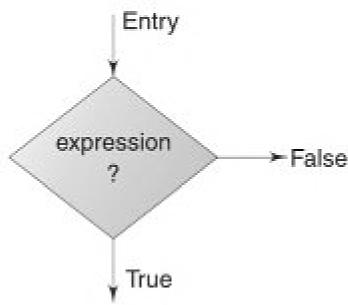


Fig. 6.1 *Two-way branching*

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

SIMPLE IF STATEMENT

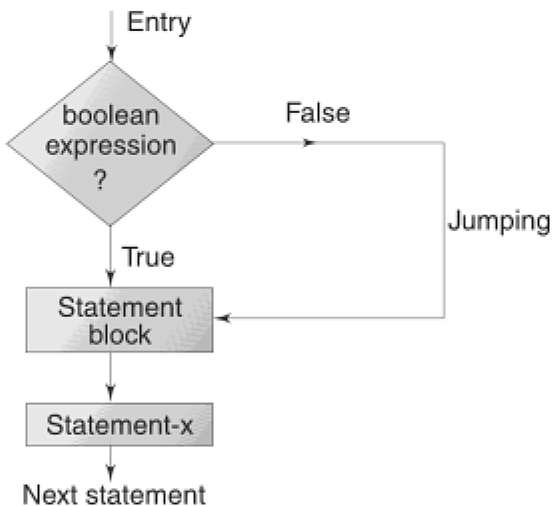


Fig. 6.2 *Flowchart of simple if control*

if else

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
```

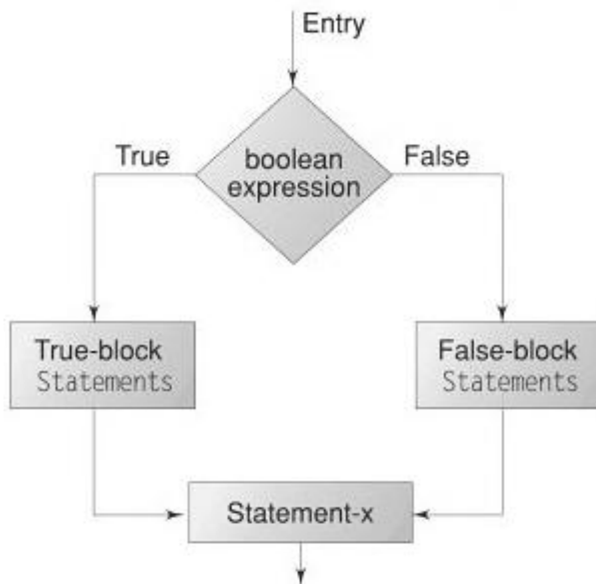
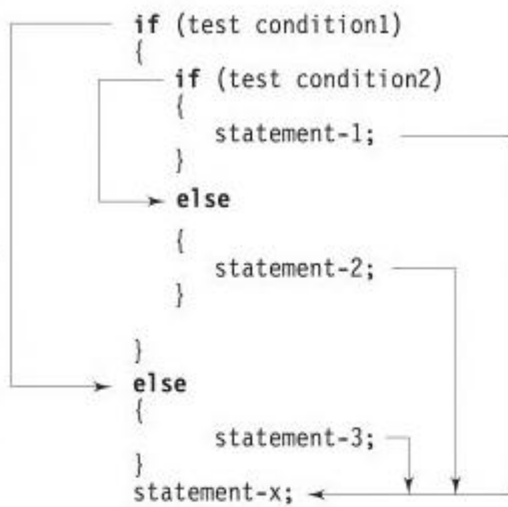


Fig. 6.3 *Flowchart of if....else control*

NESTING OF IF....ELSE STATEMENTS



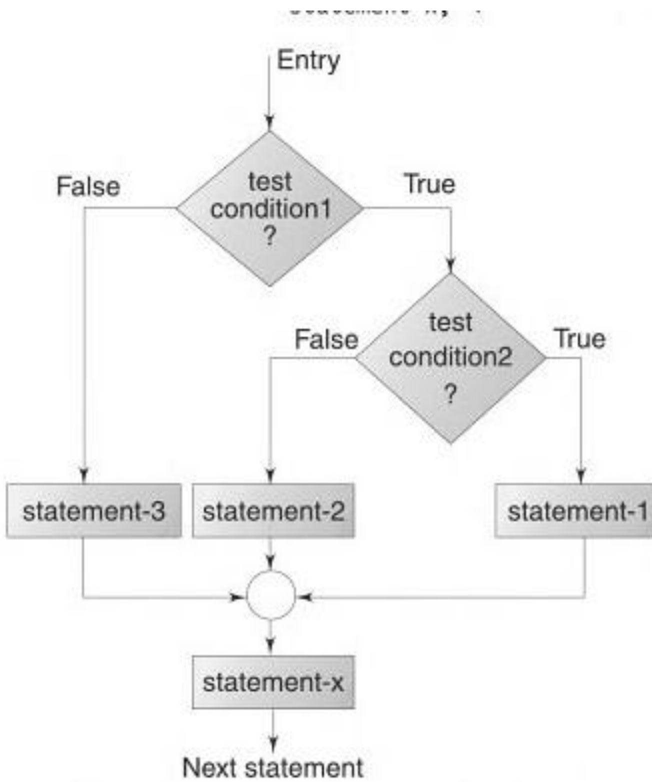
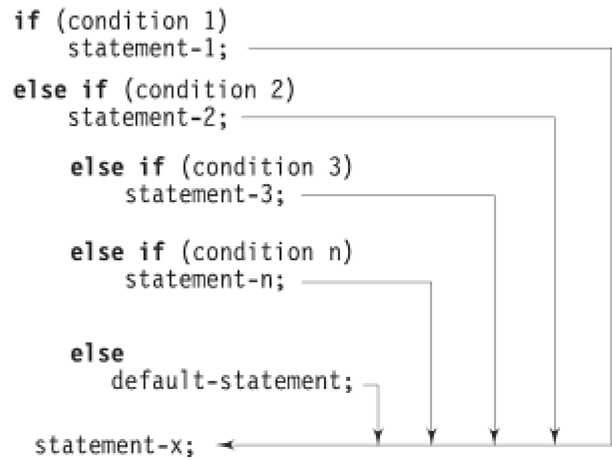


Fig. 6.4 *Flowchart of nested if....else statements*

THE ELSE IF LADDER



THE SWITCH STATEMENT

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;
```

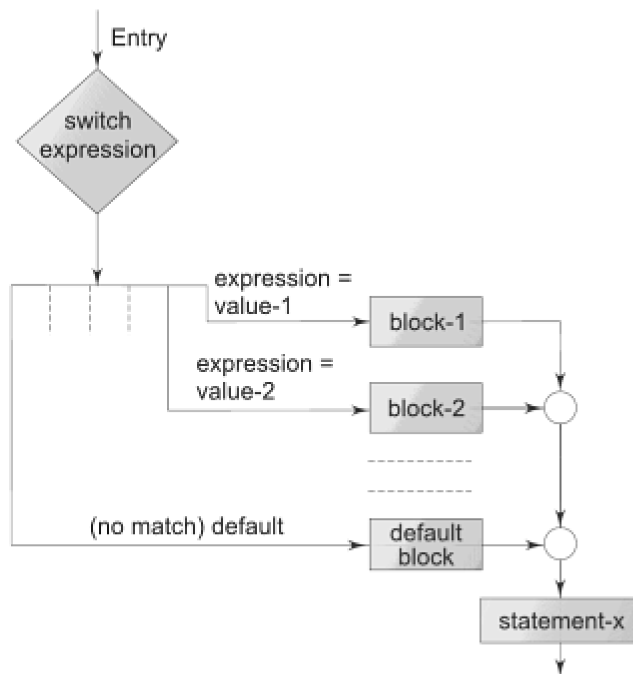


Fig. 6.6 *Selection process of the switch statement*

THE ? : OPERATOR

conditional expression ? expression1 : expression2

Decision Making and Looping

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The C# language provides for four constructs for performing loop operations. They are:

1. The **while** statement
2. The **do** statement
3. The **for** statement
4. The **foreach** statement

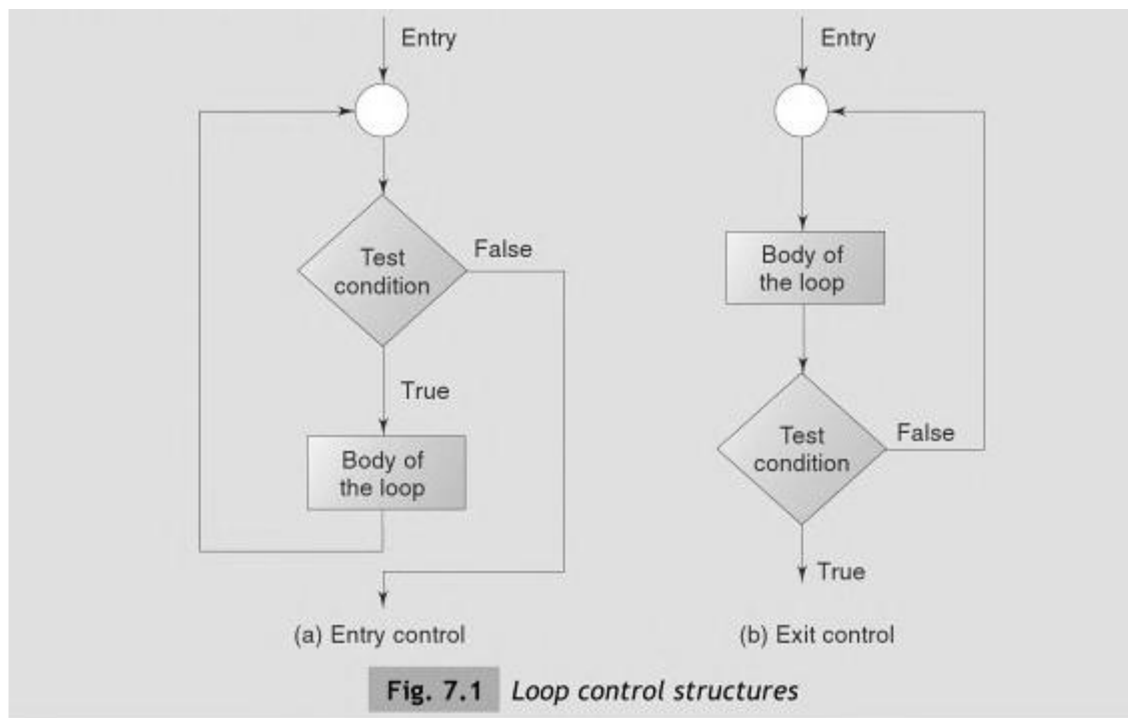


Fig. 7.1 *Loop control structures*

THE WHILE STATEMENT

The simplest of all the looping structures in C# is the **while** statement. The basic format of the **while** statement is

```
initialization;
while(test condition)
{
     Body of the loop
}
```

THE DO STATEMENT –

```
initialization;
do
{



          Body of the loop
 }
 while (test condition);
```

THE FOR STATEMENT

for is another *entry-controlled loop* that provides a more concise loop-control structure. The general form of the **for** loop is

```
for (initialization ; test condition ; increment)
{
    Body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the *test condition*. The test condition is a relational expression, such as i < 10, which determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

Table 7.1 *Comparison of the three loops*

FOR	*WHILE*	*DO*
for(n=1;n<=10;++n)	n = 1;	n = 1;
{	while (n<=10)	do
....	{	{
....	....	....
}	....	....
	n = n+1;	n = n+1;
	}	}
		while (n<=10);

Program 7.4 illustrates the use of the **for** loop for computing and printing the "power of 2" table.

7.4.2 Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in C#. We have used this concept in Program 7.2. Similarly, **for** loops can be nested as follows:

```
..........
..........
for (i = 1; i < 10; ++i)
{
     ..........
     ..........
     for (j = 1; j != 5; ++j)
     {
          ..........
          ..........        Inner loop
     }
     ..........
     ..........                              Outer
}                                            loop
..........
..........
```

JUMPS IN LOOPS

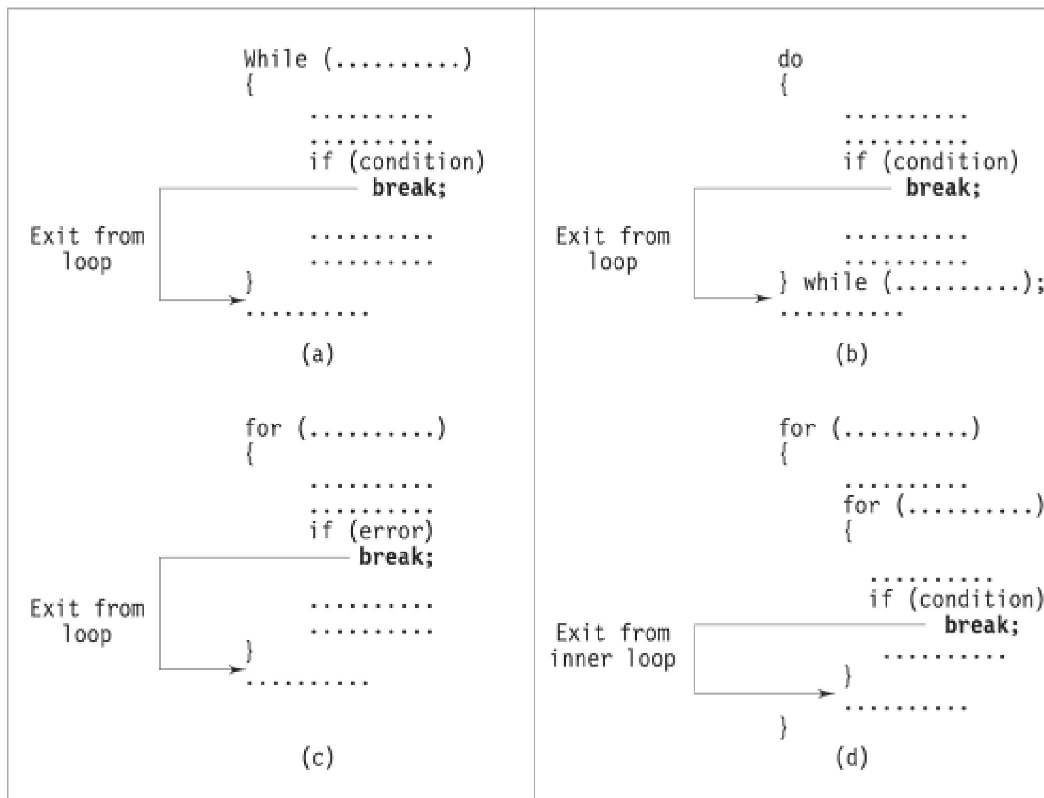


Fig. 7.2 *Exiting a loop with break statement*

Labelled Jumps

```
    public static void Main(String [ ] args)
{
      if (args.Length == 0)
            goto end;
      Console.WriteLine(args.Length);
      end: // Label name
      Console.WriteLine ("end");
}
```

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.