Table of Contents

Unit -2 C# Language Basics

Writing Console and GUI Applications; Identifiers and keywords; Writing comments; Data Types; Expressions and Operators; Strings and Characters; Arrays; Variables and Parameters; Statements (Declaration, Expression, Selection, Iteration and Jump Statements); Namespaces

A First C# Program

Here is a program that adds 10 and 5 and prints the result, 15, to the screen. The double forward slash indicates that the remainder of a line is a comment.

```
using System;      //importing namespace
class Program     //declaring class
    {
        static void Main(string[] args)   //main method
        {
            int a=10,b=5,sum;   //declaring variables
            sum = a + b;        //calculating sum
            Console.WriteLine("Sum = "+sum); //displaying the result
        }
    }
```

Output:

```
Sum = 15
```

Compilation Process

The C# compiler compiles source code, specified as a set of files with the .cs extension, into an assembly. An assembly is the unit of packaging and deployment in .NET. An assembly can be either an application or a library. A normal console or Windows application has a Main method and is an .exe file.

A library is a .dll and is equivalent to an .exe without an entry point. Its purpose is to be called upon (referenced) by an application or by other libraries. The .NET Framework is a set of libraries.

The name of the C# compiler is csc.exe. You can either use an IDE such as Visual Studio to compile, or call csc manually from the command line.

To compile manually, first save a program to a file such as MyFirstProgram.cs, and then go to the command line and invoke csc (located in **C:\Windows\Microsoft.NET\Framework\v4.0.30319**) as follows:

csc MyFirstProgram.cs

This produces an application named MyFirstProgram.exe

To produce a library (.dll), do the following:

csc /target:library MyFirstProgram.cs

Identifiers and Keywords

Identifiers are names that programmers choose for their classes, methods, variables, and so on. These are the identifiers in our example program, in the order they appear:

```
System   Test   Main   x   Console   WriteLine
```

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case-sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g., myVariable), and all other identifiers should be in Pascal case (e.g., MyMethod).

Keywords are names that mean something special to the compiler. These are the keywords in our example program:

```
using   class   static   void   int
```

Most keywords are reserved, which means that you can't use them as identifiers. Here is the full list of C# reserved keywords (**Total 77**):

```
abstract  do        in         protected   true
as        double    int        public      try
base      else      interface  readonly    typeof
bool      enum      internal   ref         uint
break     event     is         return      ulong
byte      explicit  lock       sbyte       unchecked
case      extern    long       sealed      unsafe
catch     false     namespace  short       ushort
char      finally   new        sizeof      using
checked   fixed     null       stackalloc  virtual
class     float     object     static      void
const     for       operator   string      volatile
continue  foreach   out        struct      while
decimal   goto      override   switch
default   if        params     this
delegate  implicit  private    throw
```

Avoiding conflicts

If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the @ prefix. For instance:

```
class class  {...}      // Illegal
class @class {...}      // Legal
```

The @ symbol doesn't form part of the identifier itself. So @myVariable is the same as myVariable.

Contextual keywords

Some keywords are contextual, meaning they can also be used as identifiers— without an @ symbol. These are:

```
add          dynamic  in       orderby  var
ascending    equals   into     partial  when
async        from     join     remove   where
await        get      let      select   yield
by           global   nameof   set
descending   group    on       value
```

Literals, Punctuators, and Operators

Literals are primitive pieces of data lexically embedded into the program. The literals we used in our example program are 12 and 30.

Punctuators help demarcate the structure of the program. These are the punctuators we used in our example program:

```
{   }   ;
```

The **braces** group multiple statements into a statement block.

The **semicolon** terminates a statement. (Statement blocks, however, do not require a semicolon.) Statements can wrap multiple lines:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

An **operator** transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, *. We will discuss operators in more detail later in this chapter. These are the operators we used in our example program:

```
.  ()   *   =
```

A **period** denotes a member of something (or a decimal point with numeric literals). **Parentheses** are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments.

An **equals sign** performs assignment. (The double equals sign, ==, performs equality comparison).

Comments

C# offers two different styles of source-code documentation: single-line comments and multiline comments. A single-line comment begins with a double forward slash and continues until the end of the line. For example:

```
int x = 3;   // Comment about assigning 3 to x
```

A multiline comment begins with /* and ends with */. For example:

```
int x = 3;   /* This is a comment that
                spans two lines */
```

Type Basics

A type defines the blueprint for a value. In our example, we used two literals of type int with values 12 and 30. We also declared a variable of type int whose name was x:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

A variable denotes a storage location that can contain different values over time. In contrast, a constant always represents the same value.

```
const int y = 360;
```

All values in C# are instances of a type. The meaning of a value, and the set of possible values a variable can have, is determined by its type.

Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The **int type** is a predefined type for representing the set of integers that fit into 32 bits of memory and is the default type for numeric literals within this range.

We can perform functions such as arithmetic with instances of the int type as follows:

```
int x = 12 * 30;
```

Another predefined C# type is **string**. The string type represents a sequence of characters, such as ".NET" or "http://oreilly.com". We can work with strings by calling functions on them as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);                // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);                     // Hello world2015
```

The predefined bool type has exactly two possible values: true and false. The bool type is commonly used to conditionally branch execution flow based with an if statement. For example:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The **System namespace** in the .NET Framework contains many important types that are not predefined by C# (e.g., DateTime).

Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either **implicit or explicit**: implicit conversions happen automatically, and explicit conversions require a cast.
In the following example, we implicitly convert an int to a long type (which has twice the bitwise capacity of an int) and explicitly cast an int to a short type (which has half the capacity of an int):

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee they will always succeed.
- No information is lost in conversion.

Conversely, explicit conversions are required when one of the following is true:

- The compiler cannot guarantee they will always succeed.
- Information may be lost during conversion.

Value Types Versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types

Value types comprise most built-in types (specifically, all numeric types, the char type, and the bool type) as well as custom struct and enum types.
Reference types comprise all class, array, delegate, and interface types. (This includes the predefined string type.)
The fundamental difference between value types and reference types is how they are handled in memory.

Value types

The content of a value type variable or constant is simply a value. For example, the content of the built-in value type, int, is 32 bits of data.
You can define a custom value type with the **struct** keyword:

```
public struct Point { public int X; public int Y; }
```

or more tersely:

```
public struct Point { public int X, Y; }
```

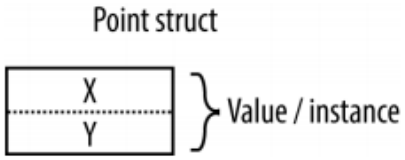

Fig: A value-type instance in memory

The assignment of a value-type instance always copies the instance. For example:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

  Point p2 = p1;             // Assignment causes copy

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Change p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

Figure below shows that p1 and p2 have independent storage.

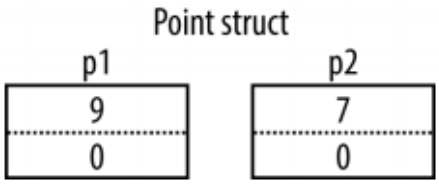

Fig: Assignment copies a value-type instance

Reference types

A reference type is more complex than a value type, having two parts: an object and the reference to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the Point type from our previous example rewritten as a class, rather than a struct.

```
public class Point { public int X, Y; }
```

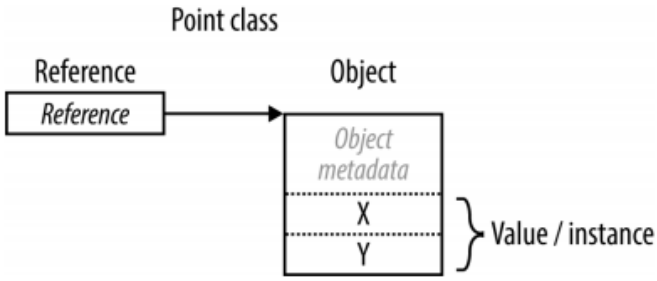

Fig: A reference-type instance in memory

Numeric Types

C# has the predefined numeric types:

C# type	System type	Suffix	Size	Range
Integral—signed				
sbyte	SByte		8 bits	-2^7 to 2^7-1
short	Int16		16 bits	-2^{15} to $2^{15}-1$
int	Int32		32 bits	-2^{31} to $2^{31}-1$
long	Int64	L	64 bits	-2^{63} to $2^{63}-1$
Integral—unsigned				
byte	Byte		8 bits	0 to 2^8-1
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$
Real				
float	Single	F	32 bits	$\pm(\sim10^{-45}$ to $10^{38})$
double	Double	D	64 bits	$\pm(\sim10^{-324}$ to $10^{308})$
decimal	Decimal	M	128 bits	$\pm(\sim10^{-28}$ to $10^{28})$

Numeric Conversions

Converting between integral types

Integral type conversions are implicit when the destination type can represent every possible value of the source type. Otherwise, an explicit conversion is required. For example:

```
int x = 12345;       // int is a 32-bit integer
long y = x;          // Implicit conversion to 64-bit integral type
short z = (short)x;  // Explicit conversion to 16-bit integral type
```

Converting between floating-point types

A float can be implicitly converted to a double, since a double can represent every possible value of a float. The reverse conversion must be explicit.Converting between floating-point and integral types

All integral types may be implicitly converted to all floating-point types:

```
int i = 1;
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

When you cast from a floating-point number to an integral type, any fractional portion is truncated; no rounding is per- formed.

The static class `System.Convert` provides methods that round while converting between various numeric types.

Implicitly converting a large integral type to a floating-point type preserves magnitude but may occasionally lose precision. This is because floating-point types always have more magnitude than integral types, but may have less precision. Rewriting our example with a larger number demonstrates this:

```
int i1 = 100000001;
float f = i1;          // Magnitude preserved, precision lost
int i2 = (int)f;       // 100000000
```

Decimal conversions

All integral types can be implicitly converted to the decimal type, since a decimal can represent every possible C# integral-type value. All other numeric conversions to and from a decimal type must be explicit.

Operators in C #:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	**Description**	**Example**
+	Adds two operands	A + B = 30

-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed

		binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, which is 0000 1111

Assignment Operators

There are following assignment operators supported by C#:

Operator	**Description**	**Example**
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Miscellaneous Operators

There are few other important operators including **sizeof, typeof** and **? :**supported by C#.

Operator	**Description**	**Example**
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.

as	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Conditional operator (ternary operator)

The conditional operator (more commonly called the ternary operator, as it's the only operator that takes three operands) has the form q ? a : b, where if condition q is true, a is evaluated, else b is evaluated. For example:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

The conditional operator is particularly useful in LINQ queries

Strings and Characters

C#'s char type (aliasing the System.Char type) represents a Unicode character and occupies 2 bytes. A char literal is specified inside single quotes:

```
char c = 'A';       // Simple character
```

Escape sequences express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning. For example:

```
char newLine = '\n';
char backSlash = '\\';
```

The escape sequence characters are shown below.

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Char Conversions

An implicit conversion from a char to a numeric type works for the numeric types that can accommodate an unsigned short. For other numeric types, an explicit conversion is required.

String Type

C#'s string type (aliasing the **System.String** type) represents an immutable sequence of Unicode characters. A string literal is specified inside double quotes:

```
string text = "Hello World !";
```

String is a reference type, rather than a value type. Its equality operators, however, follow value type semantics:

```
String a ="BCA";
String b="BCA";
Console.Write(a==b);    // prints True
```

The escape sequences that are valid for char literals also work inside strings:

```
string a = "Here's a tab:\t";
```

The cost of this is that whenever you need a literal backslash, you must write it twice:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

To avoid this problem, C# allows **verbatim** string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

A verbatim string literal can also span multiple lines:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// True if your IDE uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);
```

You can include the double-quote character in a verbatim literal by writing it twice:

```
string xml = @"<customer id=""123""></customer>";
```

String concatenation

The + operator concatenates two strings:

```
string s = "a" + "b";
```

One of the operands may be a non-string value, in which case ToString is called on that value. For example:

```
string s = "a" + 5;  // a5
```

Using the + operator repeatedly to build up a string is inefficient: a better solution is to use the **System.Text.StringBuilder** type.

String interpolation

A string preceded with the **$** character is called an **interpolated** string. Interpolated strings can include expressions inside braces:

```
int x = 4;
Console.Write ($"A square has {x} sides");  // Prints: A square has 4 sides
```

String comparisons

string does not support < and > operators for comparisons. You must use the string's **CompareTo** method.

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            string a = "BCA";
            string b = "BCA";
            if (a.CompareTo(b) == 0)
            {
```

```
                Console.Write("Both Strings are Equal !");
            }
        }
    }
```

Output:

```
Both Strings are Equal !
```

Arrays

An array represents a fixed number of variables (called elements) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type. For example:

```
int[] a=new int[5];      //array of 5 integer elements

char[] c=new char[10];     //array of 10 characters
```

Square brackets also **index** the array, accessing a particular element by position:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

This prints “e” because array indexes start at 0. We can use a for loop statement to iterate through each element in the array. The for loop in this example cycles the integer i from 0 to 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels[i]);           // aeiou
```

An array initialization expression lets you declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

Program for calculating sum of all array elements.

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            int[] a = new int[10];
            int n, i, sum = 0;
```

```
            Console.WriteLine("Enter total elements!");
            n=Convert.ToInt32(Console.ReadLine());

            Console.WriteLine("Enter array elements!");
            for(i=0;i<n;i++)
                a[i]= Convert.ToInt32(Console.ReadLine());

            for (i = 0; i < n; i++)
                sum = sum + a[i];

            Console.WriteLine("Sum = "+sum);

        }
    }
```

Output:

```
Enter total elements!
5
Enter array elements!
10
5
10
5
10
Sum = 40
```

Multidimensional Arrays

Multidimensional arrays come in two varieties**: rectangular and jagged**.
Rectangular arrays represent an n-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array, where the dimensions are 3 by 3:

```
int[,] matrix = new int[3,3];
```

A rectangular array can be initialized as follows (to create an array identical to the previous example):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Example Program,

```
class Rectangular
{
    static void Main(string[] args)
    {
        int[,] vals = new int[4, 2] {
            { 9, 99 },
            { 3, 33 },
            { 4, 44 },
            { 1, 11 }
        };

        for (int i = 0; i < 4; i++)
        {
            for (int j = 0; j < 2; j++)
            {
                Console.WriteLine(vals[i,j]);
            }
        }
            /* Using for each loop
            foreach (var val in vals)
            {
                Console.WriteLine(val);
            }*/
            Console.ReadKey();
    }
}
```

Jagged arrays

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array.

A jagged array can be initialized as follows (to create an array identical to the previous example with an additional element at the end):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

Example Program,

```
class Jagged
{
    static void Main(string[] args)
    {
        int[][] jagged = new int[][]
        {
            new int[] { 1, 2 },
            new int[] { 1, 2, 3 },
            new int[] { 1, 2, 3, 4 }
        };

        foreach (int[] array in jagged)
        {
            foreach (int e in array)
            {
                Console.Write(e + " ");
            }
            Console.Write('\n');
        }

        Console.ReadKey();
    }
}
```

Simplified Array Initialization Expressions

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Bounds Checking

All array indexing is bounds-checked by the runtime. An **IndexOutOfRangeException** is thrown if you use an invalid index:

```
int[] arr = new int[3];
arr[3] = 1;              // IndexOutOfRangeException thrown
```

As with Java, array bounds checking is necessary for type safety and simplifies debugging. Generally, the performance hit from bounds checking is minor, and the JIT (Just-In-Time) compiler can perform optimizations, such as determining in advance whether all indexes

will be safe before entering a loop, thus avoiding a check on each iteration. In addition, C# provides “unsafe” code that can explicitly bypass bounds checking.

Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a local variable, parameter (value, ref, or out), field (instance or static), or array element.

The Stack and the Heap

The stack and the heap are the places where variables and constants reside. Each has very different lifetime semantics.

Stack

The stack is a block of memory for storing local variables and parameters. The stack logically grows and shrinks as a function is entered and exited. Consider the following method:

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

This method is recursive, meaning that it calls itself. Each time the method is entered, a new int is allocated on the stack, and each time the method exits, the int is deallocated.

Heap

The heap is a block of memory in which objects (i.e., reference-type instances) reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program’s execution, the heap starts filling up as new objects are created. The runtime has a garbage collector that periodically deallocates objects from the heap, so your program does not run out of memory. An object is eligible for deallocation as soon as it’s not referenced by anything that’s itself “alive.”

```
using System;
using System.Text;

class Test
{
  static void Main()
  {
    StringBuilder ref1 = new StringBuilder ("object1");
    Console.WriteLine (ref1);
    // The StringBuilder referenced by ref1 is now eligible for GC.

    StringBuilder ref2 = new StringBuilder ("object2");
    StringBuilder ref3 = ref2;
    // The StringBuilder referenced by ref2 is NOT yet eligible for GC.

    Console.WriteLine (ref3);                   // object2
  }
}
```

Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within a class type, or as an array element, that instance lives on the heap.

Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an unsafe context, it's impossible to access uninitialized memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called (unless marked as optional.
- All other variables (such as fields and array elements) are automatically initialized by the runtime.

For example, the following code results in a compile-time error:

```
static void Main()
{
  int x;
  Console.WriteLine (x);        // Compile-time error
}
```

Fields and array elements are automatically initialized with the default values for their type. The following code outputs 0, because array elements are implicitly assigned to their default values:

```
static void Main()
{
  int[] ints = new int[2];
  Console.WriteLine (ints[0]);    // 0
}
```

The following code outputs 0, because fields are implicitly assigned a default value:

```
class Test
{
  static int x;
  static void Main() { Console.WriteLine (x); }   // 0
}
```

Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	`null`
All numeric and enum types	`0`
`char` type	`'\0'`
`bool` type	`false`

You can obtain the default value for any type with the default keyword,

```
decimal d = default (decimal);
```

Parameters

A method has a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method **Addition** has a two parameters named **a and b**, of type **int**:

```
using System;
class Program
    {
        static void Addition(int a,int b)
        {
            int sum = a + b;
            Console.WriteLine("Sum = "+sum);
        }

        static void Main(string[] args)
        {
            Addition(10,20);
        }
    }
```

Output:

```
Sum = 30
```

You can control how parameters are passed with the ref and out modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
(None)	Value	Going *in*
`ref`	Reference	Going *in*
`out`	Reference	Going *out*

Passing arguments by value

By default, arguments in C# are passed by value, which is by far the most common case. This means a copy of the value is created when passed to the method:

```
class Test
{
  static void Foo (int p)
  {
    p = p + 1;                // Increment p by 1
    Console.WriteLine (p);    // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (x);                  // Make a copy of x
    Console.WriteLine (x);    // x will still be 8
  }
}
```

Assigning **p** a new value does not change the contents of **x**, since **p** and **x** reside in different memory locations.

The ref modifier

To pass by reference, C# provides the **ref** parameter modifier. In the following example, **p and x refer to the same memory locations**:

```
class Test
{
  static void Foo (ref int p)
  {
    p = p + 1;               // Increment p by 1
    Console.WriteLine (p);   // Write p to screen
  }

  static void Main()
  {
    int x = 8;
    Foo (ref  x);            // Ask Foo to deal directly with x
    Console.WriteLine (x);   // x is now 9
  }
}
```

The out modifier

The out modifier is most commonly used to get multiple return values back from a method. For example:

```
class OutParam
{
    static void Pass(int a, int b, out int x, out int y)
    {
        x = a;
        y = b;
    }
    static void main()
    {
        int x, y;
        Pass(10, 20,out x, out y);
        Console.WriteLine(x); //displays 10
        Console.WriteLine(y); //displays 20
    }
}
```

The params modifier

The params parameter modifier may be specified on the last parameter of a method so that the method accepts any number of arguments of a particular type. The parameter type must be declared as an array. For example:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];                       // Increase sum by ints[i]
    return sum;
  }

  static void Main()
  {
    int total = Sum (1, 2, 3, 4);
    Console.WriteLine (total);              // 10
  }
}
```

Optional parameters

A parameter is optional if it specifies a default value in its declaration:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Optional parameters may be omitted when calling the method:

```
Foo();      // 23
```

Operator Precedence and Associativity

When an expression contains multiple operators, precedence and associativity determine the order of their evaluation. Operators with higher precedence execute before operators of lower precedence. If the operators have the same precedence, the operator's associativity determines the order of evaluation.

Precedence

The following expression:

```
1 + 2 * 3
```

is evaluated as follows because * has a higher precedence than +:

```
1 + (2 * 3)
```

Left-associative operators

Binary operators (except for assignment, lambda, and null coalescing operators) are left-associative; in other words, they are evaluated from left to right. For example, the following expression:

```
8 / 4 / 2
```

is evaluated as follows due to left associativity:

```
( 8 / 4 ) / 2     // 1
```

You can insert parentheses to change the actual order of evaluation:

```
8 / ( 4 / 2 )     // 4
```

Right-associative operators

The assignment operators, lambda, null coalescing, and conditional operator are right-associative; in other words, they are evaluated from right to left. Right associativity allows multiple assignments such as the following to compile:

```
x = y = 3;
```

This first assigns 3 to y, and then assigns the result of that expression (3) to x.

Null Operators

C# provides two operators to make it easier to work with nulls: the null coalescing operator and the null-conditional operator.

Null Coalescing Operator

The ?? operator is the null coalescing operator. It says "If the operand is non-null, give it to me; otherwise, give me a default value." For example:

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 evaluates to "nothing"
```

If the left-hand expression is non-null, the right-hand expression is never evaluated.

Null-conditional Operator

The ?. operator is the null-conditional or “Elvis” operator. It allows you to call methods and access members just like the standard dot operator, except that if the operand on the left is null, the expression evaluates to null instead of throwing a **NullReferenceException**:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // No error; s instead evaluates to null
```

Statements

Functions comprise statements that execute sequentially in the textual order in which they appear. A statement block is a series of statements appearing between braces (the {} tokens).

Declaration Statements

A declaration statement declares a new variable, optionally initializing the variable with an expression. A declaration statement ends in a semicolon. You may declare multiple variables of the same type in a comma-separated list. For example:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

A constant declaration is like a variable declaration, except that it cannot be changed after it has been declared, and the initialization must occur with the declaration.

```
const double c = 2.99792458E08;
c += 10;                          // Compile-time Error
```

Local variables

The scope of a local variable or local constant extends throughout the current block. You cannot declare another local variable with the same name in the current block or in any nested blocks. For example:

```
static void Main()
{
  int x;
  {
    int y;
    int x;            // Error - x already defined
  }
  {
    int y;            // OK - y not in scope
  }
  Console.Write (y);  // Error - y is out of scope
}
```

Expression Statements

Expression statements are expressions that are also valid statements. An expression statement must either change state or call something that might change state.
Changing state essentially means changing a variable. The possible expression statements are:

- Assignment expressions (including increment and decrement expressions)
- Method call expressions (both void and non-void)
- Object instantiation expressions

Here are some examples:

```
// Declare variables with declaration statements:
string s;
int x, y;
System.Text.StringBuilder sb;

// Expression statements
x = 1 + 2;                // Assignment expression
x++;                      // Increment expression
y = Math.Max (x, 5);      // Assignment expression
Console.WriteLine (y);    // Method call expression
sb = new StringBuilder(); // Assignment expression
new StringBuilder();      // Object instantiation expression
```

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. C# program control statements can be put into the following categories: selection, iteration, and jump.

- **Selection statements** allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements (that is, iteration statements form loops).
- **Jump statements** allow your program to execute in a nonlinear fashion.

C#'s Selection Statements

C# supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if

The if statement is C#'s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (condition) statement1;
else statement2;

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            int a = 10, b = 20;
            if (a > b)  //false
                Console.WriteLine(a+ " is greatest!");
            else
                Console.WriteLine(b + " is greatest!");
        }
    }
```

Output:

```
20 is greatest!
```

Here, if a is less than b, then it print's a greater. Otherwise, b greater will be printed. In no case are they are printed.

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
.
.
.
else
  statement;
```

```
int i = 10, j = 20;

if (i == j)
{
    Console.WriteLine("i is equal to j");
}
else if (i > j)
{
    Console.WriteLine("i is greater than j");
}
else if (i < j)
{
    Console.WriteLine("i is less than j");
}
```

Output:

```
i is less than j
```

switch

The switch statement is C#'s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
 case value1:
    // statement sequence
    break;
 case value2:
    // statement sequence
    break;
 .
 .
 .
 case valueN:

    // statement sequence
    break;
 default:
    // default statement sequence
}
```

Here is a simple example that uses a switch statement:

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            int n = 2;

            switch (n)
            {
```

```
                case 1:
                    Console.WriteLine("This is case 1");
                    break;
                case 2:
                    Console.WriteLine("This is case 2");
                    break;
                case 3:
                    Console.WriteLine("This is case 3");
                    break;
                default:
                    Console.WriteLine("Invalid value");
                    break;
            }
        }
    }
```

Output:

```
This is case 2
```

Iteration Statements

C#'s iteration statements are for, while, do-while and for-each loop. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. A loop statement allows us to execute a statement or group of statements multiple times.

for loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A for loop is useful when you know how many times a task is to be repeated.
The syntax of a for loop is –

```
for(initialization; Boolean_expression; update) {
   // Statements
}
```

Following is an example code of the for loop in Java.

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            //find sum of numbers from 1 to 10
            int i, sum = 0;
            for (i = 1; i <= 10; i++)
                sum = sum + i;
            Console.WriteLine("Sum = {0}",sum);
        }
```

```
    }
```

Output:
```
Sum = 55
```

while Loop

A while loop statement in C# programming language repeatedly executes a target statement as long as a given condition is true.
The syntax of a while loop is –

```
while(Boolean_expression) {
   // Statements
}
```

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            //find sum of numbers from 1 to 10
            int i, sum = 0;
            i = 1;
            while (i <= 10)
            {
                sum = sum + i;
                i++;
            }
            Console.WriteLine("Sum = {0}",sum);
        }
    }
```

Output:
```
Sum = 55
```

do while loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.
Following is the syntax of a do...while loop –

```
do {
   // Statements
}while(Boolean_expression);
```

Example

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            //find sum of numbers from 1 to 10
            int i, sum = 0;
            i = 1;
            do
            {
                sum = sum + i;
                i++;
            } while (i <= 10);

           Console.WriteLine("Sum = {0}",sum);
        }
    }
```

Output:

```
Sum = 55
```

foreach Loop

The foreach statement iterates over each element in an enumerable object. Most of the types in C# and the .NET Framework that represent a set or list of elements are enumerable. For example, both an array and a string are enumerable. Here is an example of enumerating over the characters in a string, from the first character through to the last:

```
foreach (char c in "beer")   // c is the iteration variable
  Console.WriteLine (c);

OUTPUT:
b
e
e
r
```

It can be used for retrieving elements of array. It is shown below:

```
class ForEach
{
    static void Main(string[] args)
    {
        int[] a={10,20,30,40,50};
        foreach (int val in a)
        {
            Console.WriteLine(val);
        }
        Console.ReadKey();

    }
}
```

Output:

10
20
30
40
50

Nested Loops

Like all other programming languages, C# allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            int i, j;
            for (i = 1; i <= 5; i++)
            {
                for (j = 1; j <= i; j++)
                {
                    Console.Write("*\t");
                }
                Console.Write("\n");
            }
        }
    }
```

Output:

```
*
*   *
*   *   *
*   *   *   *
*   *   *   *   *
```

Jump Statements

The C# jump statements are **break, continue, goto, return, and throw**. These statements transfer control to another part of your program.

Using break

In C#, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
using System;
class Program
    {
        static void Main(string[] args)
        {
            int i;
            for (i = 1; i <= 10; i++)
            {
                Console.Write("{0}\t",i);
                if (i == 5)
                    break;
            }
            Console.WriteLine("\nLoop Terminated !");
        }
    }
```

Output:

```
1    2    3    4    5
Loop Terminated !
```

As you can see, although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when **i** equals 10.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

In a while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0)       // If i is even,
    continue;             // continue with next iteration

  Console.Write (i + " ");
}

OUTPUT: 1 3 5 7 9
```

Using return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

Example:

```
using System;
class Program
    {
        static int Addition()
        {
            int a = 10, b = 15,sum;
            sum = a + b;
            return sum;
        }

        static void Main(string[] args)
        {
            int result;
            result = Addition();
            Console.WriteLine("Sum of two numbers={0}",result);
        }
    }
```

Output:

Sum of two numbers=25

The goto statement

The goto statement transfers execution to another label within a statement block. The form is as follows:

```
goto statement-label;
```

Or, when used within a switch statement:

```
goto case case-constant;    // (Only works with constants, not patterns)
```

A label is a placeholder in a code block that precedes a statement, denoted with a colon suffix. The following iterates the numbers 1 through 5, mimicking a for loop:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " ");
  i++;
  goto startLoop;
}

OUTPUT: 1 2 3 4 5
```

The throw statement

The throw statement throws an exception to indicate an error has occurred.

If(age<18)
throw new ArithmeticExcecption("Not Eligible to Vote");

Namespaces

A namespace is a domain for type names. Types are typically organized into hierarchical namespaces, making them easier to find and avoiding conflicts. For example, the RSA type that handles public key encryption is defined within the following namespace:

```
System.Security.Cryptography
```

The namespace keyword defines a namespace for types within that block. For example:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

The dots in the namespace indicate a hierarchy of nested namespaces. The code that follows is semantically identical to the preceding example:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

The using Directive

The using directive imports a namespace, allowing you to refer to types without their fully qualified names. The following imports the previous example's Outer.Middle.Inner namespace:

```
using Outer.Middle.Inner;

class Test
{
  static void Main()
  {
    Class1 c;    // Don't need fully qualified name
  }
}
```

using static

From C# 6, you can import not just a namespace, but a specific type, with the using static directive. All static members of that type can then be used without being qualified with the type name. In the following example, we call the Console class's static WriteLine method:

```
using static System.Console;

class Test
{
  static void Main() { WriteLine ("Hello"); }
}
```

Rules Within a Namespace

Name scoping

Names declared in outer namespaces can be used unqualified within inner name- spaces. In this example, Class1 does not need qualification within Inner:

```
namespace Outer
{
  class Class1 {}

  namespace Inner
  {
    class Class2 : Class1  {}
  }
}
```

Name hiding

If the same type name appears in both an inner and an outer namespace, the inner name wins. To refer to the type in the outer namespace, you must qualify its name. For example:

```
namespace Outer
{
  class Foo { }

  namespace Inner
  {
    class Foo { }

    class Test
    {
      Foo f1;          // = Outer.Inner.Foo
      Outer.Foo f2;    // = Outer.Foo
    }
  }
}
```

Repeated namespaces

You can repeat a namespace declaration, as long as the type names within the namespaces don't conflict:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}

namespace Outer.Middle.Inner
{

   class Class2 {}
 }
```

We can even break the example into two source files such that we could compile each class into a different assembly.

Source file 1:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

Source file 2:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Nested using directive

You can nest a using directive within a namespace. This allows you to scope the using directive within a namespace declaration. In the following example, Class1 is visible in one scope, but not in another:

```
namespace N1
{
  class Class1 {}
}

namespace N2
{
  using N1;

  class Class2 : Class1 {}
}

namespace N2
{
  class Class3 : Class1 {}   // Compile-time error
}
```

Advanced Namespace Features

Extern

Extern aliases allow your program to reference two types with the same fully qualified name (i.e., the namespace and type name are identical). This is an unusual scenario and can occur only when the two types come from different assemblies. Consider the following example.

Library 1:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs

namespace Widgets
{
  public class Widget {}
}
```

Library 2:

```
// csc target:library /out:Widgets2.dll widgetsv2.cs

namespace Widgets
{
  public class Widget {}
}
```

Application:

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs

using Widgets;

class Test
{
  static void Main()
  {
    Widget w = new Widget();
  }
}
```

The application cannot compile, because Widget is ambiguous. Extern aliases can resolve the ambiguity in our application:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs

extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();

}}
```

Exercise

Objective Questions

1. **Which is the String method used to compare two strings with each other?**
   A. Compare To() B. Compare()
   C. Copy() D. ConCat()

2. **What will be output of the following conversion?**

```
static void Main(string[] args)
 {
    char a = 'A';
    string b = "a";
    Console.WriteLine(Convert.ToInt32(a));
    Console.WriteLine(Convert.ToInt32(Convert.Tochar(b)));
    Console.ReadLine();
 }
```

   A. 1, 97 B. 65, 97
   C. 65, 97 D. 97, 1

3. **Type of Conversion in which compiler is unable to convert the datatype implicitly is?**
   A. ushort to long B. int to uint
   C. ushort to long D. byte to decimal

4. **Select output of the given set of Code :**

```
static void Main(string[] args)
{
   String name = "Dr.Gupta";
   Console.WriteLine("Good Morning" + name);
}
```

   A. Dr.Gupta B. Good Morning
   C. Good Morning Dr.Gupta D. Good Morning name

5. **Select the output for the following set of code :**

```
 static void Main(string[] args)
  {
     int i = 0, j = 0;
     l1: while (i < 2)
     {
        i++;
        while (j < 3)
        {
```

```
            Console.WriteLine("loop\n");
            goto l1;
          }
        }
        Console.ReadLine();
    }
```

A. loop is printed infinite times
B. loop
C. loop loop
D. Compile time error

6. Select the output for the following set of code:

```
static void Main(string[] args)
{
   int i, s = 0;
   for (i = 1; i <= 10; s = s + i, i++);
   {
      Console.WriteLine(s);
   }
   Console.ReadLine();
}
```

A. Code report error
B. Code runs in infinite loop condition
C. Code gives output as 0 1 3 6 10 15 21 28 36 45
D. Code give output as 55

7. Select the output for the following set of Code:

```
static void Main(string[] args)
{
   int n, r;
   n = Convert.ToInt32(Console.ReadLine());
   while (n > 0)
   {
      r = n % 10;
      n = n / 10;
      Console.WriteLine(+r);
   }
   Console.ReadLine();
}
```

for n = 5432.

A. 3245 B. 2354 C. 2345 D. 5423

8. What is the advantage of using 2D jagged array over 2D rectangular array?

A. Easy initialization of elements
B. Allows unlimited elements as well as rows which had ‘0’ or are empty in nature

C. All of the mentioned
D. None of the mentioned

9. Which statement is correct about following set of code ?

int[,]a={{5, 4, 3},{9, 2, 6}};

A. 'a' represents 1-D array of 5 integers
B. a.GetUpperBound(0) gives 9
C. 'a' represents rectangular array of 2 columns and 3 arrays
D. a.GetUpperBound(0) gives 2

10. Which of these data type values is returned by equals() method of String class?

A. char B. Int C. Boolean D. All of the mentioned

11. What will be the output the of given set of code?

```
static void Main(string[] args)
{
   int [] a = {1, 2, 3, 4, 5};
   fun(a);
   Console.ReadLine();
}
static void fun(params int[] b )
{
   for (int i = 0; i < b.Length; i++)
   {
      b[i] = b[i] * 5 ;
      Console.WriteLine(b[i] + "");
   }
}
```

A. 1, 2, 3, 4, 5 B. 5, 10, 15, 20, 25
C. 5, 25, 125, 625, 3125 D. 6, 12, 18, 24, 30

12. What will be the output of the following C# code?

```
static void Main(string[] args)
{
   byte varA = 10;
   byte varB = 20;
   long result = varA & varB;
   Console.WriteLine("{0}  AND  {1} Result :{2}", varA, varB, result);
   varA = 10;
   varB = 10;
   result = varA & varB;
   Console.WriteLine("{0}  AND  {1} Result : {2}", varA, varB, result);
   Console.ReadLine();
}
```

a) 0, 20 b) 10, 10 c) 0, 10 d) 0, 0

13. Which of the following options is not a Bitwise Operator in C#?

a) &, | b) ^, ~ c) <<, >> d) +=, -=

14. What will be the output of the following C# code?

```
bool a = true;
bool b = false;
a |= b;
Console.WriteLine(a);
Console.ReadLine();
```

a) 0 b) 1 c) True d) False

15. Arrange the operators in the increasing order as defined in C#.

!=, ?:, &, ++, &&

a) ?: < && < != < & < ++ b) ?: < && < != < ++ < &

c) ?: < && < & <!= < ++ d) ?: < && < != < & < ++

Answer Key:

1. A	2. C	3. B	4. C	5. C
6. D	7. C	8. B	9. C	10. C
11. B	12. C	13. D	14. C	15. C

Short Answer Questions

1. Explain compilation process of C# program with example.
2. Explain identifiers and keywords used in C#.
3. What do you mean by type conversion? Explain implicit and explicit conversion with example.
4. Differentiate value type and reference type with example.
5. Explain ternary operator with example.
6. Define escape sequence. What are different escape sequence supported by C#? Explain.
7. What is an array? Differentiate rectangular and jagged array with example.
8. Differentiate parameters pass by value and by reference with example.
9. Explain operator precedence with example.
10. What is namespace? Explain the use of namespace in C# program with example.
11. Write a C# program to compare any two strings.
12. Write a C# program to demonstrate use of out and params modifier.
13. WAP to find sum of two numbers.
14. WAP to find product of two numbers.
15. WAP to add, subtract, multiply and divide two numbers.
16. WAP to find simple interest. [si=(p*t*r)/100]

17. WAP to area of rectangle. [area=l*b]
18. WAP to find area of circle. [area=pi*r*r] (use pi as constant)
19. WAP to find largest among two numbers.
20. WAP to find smallest among two numbers.
21. WAP to find largest among three numbers.
22. WAP to find smallest among three numbers.
23. WAP to check whether a number is odd or even.
24. WAP to check whether a number is divisible by 7 or not.
25. WAP to check whether a number is exactly by 5 and 10.
26. WAP to check whether a number a number is divisible by 7 but not by 13.
27. WAP to input CP and SP and check profit or loss. Also find profit or loss amount.
28. WAP to find print numbers from 1 to 10.
29. WAP to find sum of numbers from 5 to 100.
30. WAP to check whether a number is prime or not.
31. WAP to print prime numbers from 1 to 100.
32. WAP to show the use of ternary operator.
33. Write a program to show the use of switch case statement.
34. Write a program to show the use of auto-increment and auto-decrement operators.
35. Write a C# program to generate multiplication table from 1 to 10.
36. Write a C# program to find factorial of any number.
37. Write a program to show the use of break, continue and return.
38. WAP to print following pattern.

a.
```
*
**
***
****
*****
```

b.
```
*****
 ****
  ***
   **
    *
```

c.
```
1
12
123
1234
12345
```

d.
```
1
22
333
4444
55555
```

Long Answer Questions

1. What do you mean by operators? Explain different operators used in C# with examples.
2. What are control statements? Explain different types of control statements with examples in detail.
3. Write a C# program to find sum of 20 numbers in an array.
4. Write a C# program to find maximum and minimum number from array.
5. Write a C# program to sort ‘n’ numbers in ascending order.
6. Write a C# program to sort name of 10 persons in alphabetical order.
7. Write a C# program to add any two matrices.
8. Write a C# program to find product of any two matrices.