C#_1.1 the .NET Framework

C# was initially designed by ***Microsoft*** to create code for the ***.NET*** Framework. Second, the ***libraries*** used by ***C#*** are the ones ***defined*** by the ***.NET Framework***. The ***.NET*** Framework defines an environment that supports the development and execution of ***highly distributed, component-based*** applications. It enables ***different*** computer ***languages*** to work ***together*** and provides for ***security***, program ***portability***, and a common programming model for the ***Windows platform***.

- ***Common Language Runtime:*** This is the system that ***manages the execution*** of your program. Along with other benefits, the ***Common Language Runtime*** is the part of the ***.NET Framework*** that enables programs to be ***portable***, supports ***mixed-language programming***, and provides for ***security***.
- ***.NET class library:*** This library gives your program access to the ***runtime environment***. For example, if you want to perform ***I/O***, such as displaying something on the screen, you will use the ***.NET class library*** to do it.
- ***CLR: Common Language Runtime:*** The ***Common Language Runtime (CLR)*** manages the execution of ***.NET*** code. Here is how it works.
  - ***MISL output:*** When you ***compile*** a C# program, the ***output*** of the compiler is ***not executable*** code. Instead, it is a file that contains a special type of ***pseudocode*** called ***Microsoft Intermediate Language, or MSIL*** for short. ***MSIL*** defines a set of portable instructions that are independent of any specific CPU. In essence, ***MSIL*** defines a ***portable assembly language***.
  - ***MISL to .exe:*** It is the job of the ***CLR*** to translate the intermediate code (***MISL***) into executable code (***.exe***) when a program is run. Thus, any program compiled to ***MSIL*** can be run in ***any environment*** for which the ***CLR*** is ***implemented***. This is part of how the ***.NET Framework*** achieves ***portability***.
  - ***Metadata:*** when you compile a C# program: ***metadata*** is produced with ***MSIL***. Metadata describes the ***data used by your program*** and enables your code to interact with other code. The ***metadata*** is contained in the same file as the ***MSIL***.

  NOTE: Although ***MSIL*** is similar in concept to ***Java's bytecode***, the two are not the same.
- ***C# program actually executes as native code:*** ***MISL*** is turned into ***executable code*** using a ***JIT compiler***. ***JIT*** stands for "***just in time***" The process is:
  - When a ***.NET*** program is executed, the ***CLR*** activates the ***JIT*** compiler. The ***JIT*** compiler converts ***MSIL*** into ***native code*** on a demand basis, of a programs part.
  - Thus, your C# program actually executes as ***native code***, even though it was ***initially compiled*** into ***MSIL***. This means that your program runs ***nearly as fast as*** it would if it had been compiled to ***native code directly*** (as ***C/C++*** do), but it gains the ***portability*** and security benefits of ***MSIL***.

C#_1.2 Managed &Unmanaged Code and Common Language Specification (CLS)

- ***Managed Code:*** A C# program is called ***managed*** code. Managed code is executed under the control of the ***CLR***. In a Managed code: The ***compiler*** must produce an ***MSIL*** file targeted for the ***CLR*** (which C# does) and use the ***.NET Framework library*** (which C# does). The benefits of managed code are many, including modern ***memory management***, the ability to ***mix languages***, better ***security***, support for ***version control***, and a clean way for software components to interact.
- ***Unmanaged code:*** The opposite of managed code is unmanaged code. ***Unmanaged*** code does ***not execute under*** the ***CLR***. Thus, all Windows programs prior to the creation of the ***.NET Framework*** use unmanaged code. It is possible for ***managed code and unmanaged code to work together***.
- ***The Common Language Specification (CLS):*** Although all ***managed code*** gains the benefits provided by the ***CLR***, if your code will be used by other programs written in ***different languages***, for maximum usability, it should adhere to the ***Common Language Specification (CLS)***. The ***CLS*** describes a set of features, such as ***data types***, that different languages have in common.
- ***C++ and CLR:*** Initially, Microsoft added what are called the managed extensions to C++. However, this approach has been rendered obsolete and is replaced by a set of ***extended keywords*** and ***syntax*** defined by the ***Ecma C++/CLI Standard***. (***CLI*** stands for ***Common Language Infrastructure***). Although ***C++/CLI*** make it possible to ***port*** existing code to the ***.NET Framework***, new ***.NET*** development is much easier in ***C#*** because it was originally designed with ***.NET*** in mind.

C#_1.3 Compile and Run first program

- ***Locating csc.exe compiler:*** Normally ***csc.exe*** is already in a Windows system where ***.NET*** is already installed. It is in the directory : ***C:\Windows\Microsoft.NET\Framework\v3.5*** or ***C:\Windows\Microsoft.NET\Framework\v4.0.30319***, depend on the latest .NET update.
- ***Adding csc.exe to system path:*** *Control pannel → Advanced System Settings → Environment Variavles (right , last)*
  - Add path into *user variables : path (here add the path)* and *system variables : path (here add the path)*
  - If variables doesn't exist : then click "***create a path***".
  - Before edit the path make sure you copied the desired directory of the ***csc.exe***. Eg: ***. . . ; C:\Windows\Microsoft.NET\Framework\v4.0.30319***
- ***First program:*** Give the source file name ***Example.cs***

```
/* This is a simple C# program. Call this program Example.cs. */
using System;
class Example {     static void Main(){          // A C# program begins with a call to Main().
                                                 Console.WriteLine("Done C, C++ and Java. YO!! ");     }}
```

  - ***Compiling the Program:*** Execute the C# ***compiler***, ***csc.exe***, specifying the name of the source file on the command line: ***C:\>csc Example.cs***
    - The ***csc*** compiler creates a file called ***Example.exe*** that contains the ***MSIL*** version of the program. Although ***MSIL*** is not executable code, it is still contained in an ***exe*** file. The ***CLR*** automatically invokes the ***JIT*** compiler when you attempt to execute ***Example.exe***. If ***.NET*** Framework is not installed, the program will not execute, because the ***CLR*** will be missing.
  - ***Running the Program:*** To actually run the program, just type its name on the command line, as: ***C:\>Example***
  - ***Naming source file:*** The name of a C# program can be chosen arbitrarily. Unlike ***Java*** in which the name of a program file is very important, this is not the case for C#. For example, the preceding sample program could have been called ***Sample.cs***, ***Test.cs***, or even ***MyProg.cs***.
    - By convention, C# programs use the ***.cs*** file extension,
  - ***Comment:*** Multi-line comment ***/* . . . */***, single-line comment ***//. . .*** and ***XML documenting*** comment

- ***using System;*** This line indicates that the program is using the ***System*** namespace. In C#, a ***namespace*** defines a ***declarative region***. ***names*** declared in one ***namespace*** will not conflict with the same ***names*** declared in ***another***. The ***namespace*** used by the program is ***System***. The ***using*** keyword states that the program is ***using the names in the given namespace***.
- ***class Example {*** This line uses the keyword class to declare that a new class is being defined.
- ***static void Main(){*** This line begins the Main( ) method.
- ***static***: A method that is modified by static can be called before an object of its class has been created. This is necessary because ***Main()*** is called at program startup.
- ***void*** indicates that ***Main()*** does not return a value.
- The empty parentheses that follow ***Main*** indicate that no information is passed to ***Main()***.
- ***Console.WriteLine("C# . . . power.");*** ***WriteLine()*** the built-in method, displays the string that is passed to it. By connecting ***Console*** with ***WriteLine( )***, noticing the compiler that ***WriteLine( )*** is a member of ***Console*** class.
- All statements in ***C#*** end with a ***semicolon***. A block does not end with a ***semicolon***.
- C# is case-sensitive. Forgetting this can cause serious problems. For example, typing main instead of Main, or writeline instead of WriteLine, the preceding program will be incorrect.
- ***Console.WriteLine("A simple C# program.");*** can be rewritten as ***System.Console.WriteLine("A simple C# program.");*** In this case there is no need of ***using System;*** statement. ***WriteLine()*** and ***Write()*** are similar to Java's ***println()*** and ***print()*** *[recall 1.2.4 Java part]*.

C#_1.4 *Variable Declarations, Data-types, Operator Basic, Basic if & for and STATEMENT BLOCK is same as JAVA [recall 1.3, 1.4 Java part].*

C#_1.5 The C# Keywords

The C# Reserved Keywords									
abstract	as	base	bool	break	byte	case	catch	char	checked
class	const	continue	decimal	default	delegate	do	double	else	enum
event	explicit	extern	false	finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal	is	lock	long	namespace
new	null	object	operator	out	override	params	private	protected	public
readonly	ref	return	sbyte	sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while			
The C# Contextual Keywords									
from	get	group	into	join	let	orderby	partial	select	set
value	where	yield							

@-qualified keywords: Although you cannot use any of the ***C#*** keywords as ***identifiers***, C# does allow you to precede a keyword with an ***@,*** allowing it to be a ***legal*** identifier. For example, ***@for*** is a valid identifier. In this case, the identifier is actually ***for*** and the ***@ is ignored***. Frankly, using ***@-qualified keywords*** for identifiers is not recommended, except for special purposes.

C#_1.6 The C# Class Library

As we know ***WriteLine( )*** and ***Write( )*** methods are members of the ***Console*** class, which is part of the ***System*** namespace, which is defined by the ***.NET Framework's class library***. The *C# environment* relies on the *.NET Framework class library* to provide support for such things as ***I/O***, ***string handling***, ***networking***, and ***GUIs***. Thus, C# as a totality is a combination of the ***C# language*** itself, plus the ***.NET standard classes***. Part of becoming a ***C# programmer*** is learning to use the ***standard library***.

C#_1.7 C#'s Value Types

VALUE types and REFERENCE types: C# contains two general categories of built-in data types: ***value*** types and ***reference*** types. For a ***value*** type, a variable holds an ***actual value***, such 101 or 98.6. For a ***reference*** type, a variable holds a ***reference to the value***. Similar to JAVA's ***primitive*** type and ***reference*** type.

- ***Integers:*** C# defines nine integer types: ***char***, ***byte***, ***sbyte***, ***short***, ***ushort***, ***int***, ***uint***, ***long***, and ***ulong***. The ***char*** type is primarily used for representing characters.
- ***Floates:*** C# defines three floating-point types: ***float***, ***double***, ***decimal***.
- ***Booleans:*** The ***bool*** type represents ***true/false*** values. C# defines the values true and false using the reserved words ***true*** and ***false***. Thus, a variable or expression of type ***bool*** will be one of these two values. When a ***bool*** value is output by `WriteLine()`, "***True***" or "***False***" is displayed

Type	*keyword*	*Meaning*	*Bit width*	*Range*
Boolean	`bool`	Represents true/false values		
Character	`char`	Character		
Integer	`byte`	8-bit unsigned integer	8	*0 to 255*
	`sbyte`	8-bit signed integer	8	*–128 to 127*
	`short`	Short integer	16	*–32,768 to 32,767*
	`ushort`	Unsigned short integer	16	*0 to 65,535*
	`int`	Integer	32	*–2,147,483,648 to 2,147,483,647*
	`uint`	Unsigned integer	32	*0 to 4,294,967,295*
	`long`	Long integer	64	*–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807*
	`ulong`	Unsigned long integer	64	*0 to 18,446,744,073,709,551,615*
floating	`float`	Single-precision floating point	32	*of 1.5E–45 to 3.4E+38*
	`double`	Double-precision floating point	64	*of 5E–324 to 1.7E+308.*
	`decimal`	Numeric type for financial calculations	128	*1E–28 to 7.9E+28*

- For a ***signed*** value, if the ***high-order bit*** were set to ***1***, the number would then be interpreted as ***–1*** (assuming the ***two's complement*** format).
- ***The decimal Type:*** Perhaps the most interesting C# numeric type is ***decimal***, which is intended for use in ***monetary calculations***. The decimal type utilizes ***128 bits*** to represent values within the range ***1E–28 to 7.9E+28***. The ***decimal*** type is not supported by ***C, C++, or Java*** as a built-in type. Thus, within its direct line of descent, it is ***unique***. The ***decimal*** type eliminates ***rounding errors*** and can accurately represent up to ***28*** decimal places (or ***29*** places, in some cases).
  - ***decimal*** values must be followed by an ***m*** or ***M***. Because ***without*** the ***suffix***, these values would be interpreted as ***standard floating-point*** constants, which are not compatible with the ***decimal*** data type Eg: `decimal balance; balance = 1000.10m;`
- The ***`Sqrt()`*** method is defined by the ***`System.Math`*** class returns a ***`double`***.
- There is *no conversion defined between* ***`bool`*** *and* ***`integer`*** values. For example, ***1 does not convert to true***, and ***0 does not convert to false***.
- A ***character*** variable can be assigned a value by enclosing the ***character*** inside ***single quotes***. For example, this assigns ***X*** to the variable ***ch***: `char ch; ch = 'X';`
  - Although ***char*** is defined by ***C#*** as an ***integer*** type, it *cannot be freely mixed with* ***integers*** in all cases. This is because there is no **automatic type conversion** from ***integer*** to ***char***. For example, the following fragment is invalid:
    `char ch; ch = 10;` Error!!! Won't work the assignment involves ***incompatible types***. To resolve this we need ***type cast***.

C#_1.8 Formatted Output

When outputting lists of data, you have been separating each part of the list with a ***plus*** sign, as: *Console.WriteLine("You ordered " + 2 + " items at $" + 3 + " each.");*
It does not give control over information appearance. Eg: *Console.WriteLine("Here is 10/3: " + 10.0/3.0);* It generates this output: ***Here is 10/3: 3.33333333333333***

- To control how numeric data is formatted, you will need to use a second form of WriteLine( ), shown here, which allows you to embed formatting information similar to C: `WriteLine("format string", arg0, arg1, ... , argN)` Here, the arguments to `WriteLine()` are separated by ***commas*** and ***not plus*** signs.
  - The `format string` contains two items: *regular, printing characters that are displayed as-is* and *format specifiers*. ***Format specifiers*** take this general form: `{argnum, width: fmt}`
    - Here, ***argnum*** specifies the **number of the argument** (starting from ***zero***) to display. The **minimum width of the field** is specified by ***width***, and the **format** is specified by ***fmt***. Both ***width*** and ***fmt*** are ***optional***. Thus, in its simplest form, a format specifier simply indicates which argument to display. For example, ***{0}*** indicates ***arg0***, ***{1}*** specifies ***arg1***, and so on. Eg: `Console.WriteLine("February has {0} or {1} days.", 28, 29);`
      Produces the output: **February has 28 or 29 days**. As you can see, the value ***28*** is substituted for ***{0}***, and ***29*** is substituted for ***{1}***.
    - Preceding statement that specifies minimum field widths: `Console.WriteLine("February has {0,10} or {1,5} days.", 28, 29);`
      It produces the output: **February has 28 or 29 days**. As you can see, ***spaces*** have been added to ***fill*** out the unused portions of the fields. Remember, a ***minimum field*** width is just that: the ***minimum width***. Output can ***exceed*** that width if needed.

- One of the easiest ways to ***specify a format*** is to describe a template that ***WriteLine()*** will use. To do this, show an example of the format that you want, using **#**s to mark the digit positions. For instance, to display 10/3: `Console.WriteLine("Here is 10/3: {0:#.##}", 10.0/3.0);` The output from this statement is: `Here is 10/3: 3.33` Here, the template is **#.##**, which tells ***WriteLine()*** to display two decimal places.
- If you want to display ***monetary*** values, use the ***C*** format specifier. For example: `decimal balance; balance = 12323.09m;` `Console.WriteLine("Current balance is {0:C}", balance);` The output is (in U.S. dollar format): ***Current balance is $12,323.09***

C#_1.9 Literals

- ***Character and string literals:*** character literals are enclosed between ***single quotes***. For example ***'a'*** and ***'%'*** are both ***character literals***. A ***string literal*** is a set of characters enclosed by ***double quotes***. For example, "***this is a test***" is a string. In addition to normal characters, a ***string literal*** can also contain one or more of the ***escape sequences***. List of escape sequences :

Esc. Seq.	\'	\"	\\	\0	\v	\t	\r	\n	\f	\b	\a
Description	Single quote	Double quote	Backslash	Null	Vertical tab	Horizontal tab	Carriage return	Newline	Form feed	Backspace	Alert

  - ***A string consisting of a single character not the same as a character literal:*** You must not confuse ***strings*** with ***characters***. A ***character literal*** represents a ***single letter*** of type ***char***. A ***string*** containing only ***one letter*** is still a ***string***. Although ***strings consist of characters***, they are not the same type. Example: "***k***" is not the same as ***'k'***.
  - ***Verbatim string literal:*** A ***verbatim string*** literal begins with an **@**, which is followed by a ***quoted string***. The contents of the quoted string are accepted ***without modification*** and can ***span two or more lines***. Thus, you can include ***newlines***, ***tabs***, and so on, but you ***don't need to use the escape sequences***. The only exception is that to obtain a double ***quote*** ("), you must use two ***double quotes*** in a ***row*** ("").Here is a program that demonstrates verbatim string literals:
    - The advantage of ***verbatim string*** literals is that you can specify output in your program ***exactly as it will appear on the screen***.

```
using System;                    /* Following contains embedded newlines, embedded tabs */
class Verbatim { static void Main() { Console.WriteLine( @"This is a verbatim
                                                            string literal
                                                            that spans several lines. ");
                                      Console.WriteLine(    @"Here is some tabbed output:
                                                            1   2   3   4
                                                            5   6   7   8       ");
                                      Console.WriteLine(    @"Programmers say, ""I like C#."""); }}
```

```
OUTPUT:

This is a verbatim
string literal
that spans several lines.
Here is some tabbed output:
1          2          3          4
5          6          7          8
Programmers say, "I like C#."
```

  - ***Numerical literals:*** An ***integer literal*** is of type ***int***, ***uint***, ***long***, or ***ulong***, depending upon its value. Second, floating-point literals are of type ***double***. You can explicitly specify literal types by including a ***suffix*** then C#'s default literal types is ignored.
    - To specify a ***long literal***, append an ***l*** or an ***L***. For example, ***12*** is an ***int***, but ***12L*** is a ***long***. To specify an ***unsigned integer*** value, append a ***u*** or ***U***. Thus, ***100*** is an ***int***, but ***100U*** is a ***uint***. To specify an ***unsigned, long*** integer, use ***ul*** or ***UL***. For example, ***984375UL*** is of type ***ulong***.
    - To specify a ***float literal***, append an ***F*** or ***f***. Eg: ***10.19F*** is of type ***float***. You can specify a ***double literal*** by appending a ***D*** or ***d***. (As just mentioned, ***floating-point*** literals are ***double*** by *default*.) To specify a ***decimal literal***, follow its value with an ***m*** or ***M***. For example, ***9.95M*** is a ***decimal literal***.
    - Although ***integer literals*** create an ***int***, ***uint***, ***long***, or ***ulong*** value by default, they can still be assigned to variables of type ***byte***, ***sbyte***, ***short***, or ***ushort*** as long as the value being assigned can be represented by the ***target type***.
    - A ***hexadecimal literal*** begin with ***0x*** (zero followed by x). C# allows integer literals to be specified only in ***decimal*** or ***hexadecimal***. ***Octal*** is not used in C#.

```
count = 0xFF; /* 255 in decimal */          incr = 0x1a; /* 26 in decimal */
```

C#_1.10 Variable INITIALIZATION, DYNAMIC Initialization are same as JAVA

C#_1.11 Implicitly Typed Variables

It is ***possible*** to let the ***compiler determine*** the type of a ***variable*** based on the ***value used to initialize*** it. This is called an ***implicitly typed variable***. An implicitly typed variable is declared using the keyword ***var***, and it *must be initialized*. Example: `var pi = 3.1416;` Because ***pi*** is initialized with a ***floating-point*** literal (whose type is ***double*** by default), the type of ***pi*** is ***double***. If pi been declared like this: `var pi = 3.1416M;` then ***pi*** would have the type ***decimal*** instead.

```
var pi = 3.1416;          /* pi is a double */
var radius = 10;          /* radius is an int */

var msg = "Radius: ";     /* string types.*/
var msg2 = "Area: ";      /* string types.*/
```

```
double area;              /* Explicitly declare area as a double. */
area = pi * radius * radius;

Console.WriteLine(msg + area);
// radius = 12.2; // Error : because radius is int and cannot be assigned a floating-point value.
```

- An ***implicitly typed*** variable is still a ***strongly typed*** variable. Notice this commented-out line in the program: `// radius = 12.2; // Error!` This assignment is invalid because radius is of type ***int***. Thus, it cannot be assigned a ***floating-point*** value.
- The only difference between an ***implicitly*** typed variable and a "normal" ***explicitly*** typed variable is how the type is determined. Once that type has been determined, the variable has a type, and this type is fixed throughout the lifetime of the variable. Thus, the type of ***radius*** cannot be changed ***during the execution*** of the program.
- Only ***one implicitly typed*** variable can be declared at any ***one time***. Therefore, the declaration: `var count = 10, max = 20;` *// Error! won't compile.*

C#_1.12 Life-time and scope of variables:

Local variables are declared within a ***method*** or any ***block***. A ***block*** defines a ***scope***. Thus, each time you start a ***new block***, you are creating a ***new scope***. A ***scope*** determines what names are visible to other parts of your program. It also determines the ***lifetime*** of ***local variables***.

- ***Scopes*** can be ***nested***. ***Local*** variables declared in the ***outer scope*** will be visible to code within the ***inner scope***. Local variables declared within the ***inner scope*** will ***not*** be visible ***outside*** it.
- Within a ***block***, ***local*** variables can be declared at ***any point***, but are ***valid*** only ***after*** they are ***declared***. A variable declared at the ***end*** of a ***block***, it is effectively ***useless***.
- If a *variable declaration* includes an ***initializer***, that variable will be reinitialized each time the block in which it is declared is entered.
- ***Unlike C/C++:*** Although ***blocks*** can be ***nested***, ***no*** variable declared within an ***inner scope*** can have the ***same name*** as a variable declared by an ***enclosing scope***. For example, declaring two separate variables with the same name within a nested loop, will not compile. In ***C/C++***, there is no restriction on the names that you give variables declared in an ***inner scope***.

C#_1.13 Operators : Following operators are same to 1.14 (Java Part)

Arithmetic Operators,	relational operator	short-circuit/ conditional AND-OR,	The Assignment Operator,
Increment and Decrement,	logical operator,	Compound Assignments/shorthand assignment	

NOTE: ***Modulus*** operator **%** is applicable for ***floating-point*** types in ***C#.***

C#_1.14 Type Conversions and Type casts : Same as 1.16 (Java Part)

C#_1.15 Operator Precedence:

Highest														***Lowest***
`()  [ ]` `x--  x++ (Postfix)` `checked` `new` `sizeof` `typeof` `unchecked`	`!` `~` `(type-cast)` `+ (unary)` `- (unary)` `++x (prefix)` `--x (prefix)`	`*` `/` `%`	`+` `-`	`<<` `>>`	`<` `>` `<=` `>=` `is`	`==` `!=`	`&`	`^`	`|`	`&&`	`||`	`??`	`?:`	`=` `op=` `=>`

C#_1.16 C#'s type promotion rules

Here is the algorithm that the rules define for *binary operations*:
IF one operand is ***decimal***, THEN the other operand is promoted to ***decimal*** (unless it is of type ***float*** or ***double***, in which case an ***error results***).
ELSE IF one of the operands is ***double***, the second is promoted to ***double***.
ELSE IF one operand is a ***float*** operand, the second is promoted to ***float***.
ELSE IF one operand is a ***ulong***, the second is promoted to ***ulong*** (unless it is of type ***sbyte***, ***short***, ***int***, or ***long***, in which case an error results).
ELSE IF one operand is a ***long***, the second is promoted to ***long***.
ELSE IF one operand is a ***uint*** and the second is of type ***sbyte***, ***short***, or ***int***, both are promoted to ***long***.
ELSE IF one operand is a ***uint***, the second is promoted to ***uint***.
ELSE both operands are promoted to ***int***.

- *Not all types can be mixed in an expression.* Specifically, there is no implicit conversion from ***float*** or ***double*** to ***decimal***, and it is not possible to mix ***ulong*** with any ***signed integer*** type. To mix these types requires the use of an ***explicit cast***.
- Second, pay special attention to the last rule. It states that if none of the preceding rules applies, then all other operands are promoted to ***int***. Therefore, in an expression, all ***char***, ***sbyte***, ***byte***, ***ushort***, and ***short*** values are promoted to ***int*** for the purposes of calculation. This is called ***integer promotion***. It also means that the outcome of all arithmetic operations will be ***no smaller*** than ***int***.
- It is important to understand that ***type promotions*** apply to the values operated upon only when an ***expression*** is evaluated. For example, if the value of a byte variable is promoted to ***int*** inside an expression, outside the expression, the variable is still a ***byte***. Type promotion only affects the evaluation of an expression.
- Type promotion can, however, lead to somewhat unexpected results. For example, when an arithmetic operation involves two ***byte*** values, the following sequence occurs: First, the ***byte*** operands are promoted to ***int***. Then the operation takes place, yielding an ***int*** result. Thus, the outcome of an operation involving two ***byte*** values will be an ***int***. This is not what you might intuitively expect.

```
byte b; int i;
b = 10;    i = b * b;      // OK, no cast needed
b = 10;    b = (byte) (b * b);      // cast needed!!
```

  - This same sort of situation also occurs when performing operations on ***chars***. For example, the *cast back* to ***char*** is needed because of the promotion of ***ch1*** and ***ch2*** to ***int*** within the expression: `char ch1 = 'a', ch2 = 'b'; ch1 = (char) (ch1 + ch2);`
    - ***Without*** the ***cast***, the result of adding ***ch1*** to ***ch2*** would be ***int***, which can't be assigned to a ***char***.
- For the ***unary operations***, operands smaller than ***int*** (***byte***, ***sbyte***, ***short***, and ***ushort***) are promoted to ***int***. Also, a ***char*** operand is converted to ***int***. Furthermore, if a ***uint*** value is negated, it is promoted to ***long***.

C#_1.17 Inputting Characters from the Keyboard

To read a character from the ***keyboard***, call ***`Console.Read()`***. This method waits until the user presses a key and then returns the key. The character is returned as an ***integer***, so it must be ***cast to char*** to assign it to a ***char*** variable. By default, console input is ***line-buffered***, so you must press ***ENTER*** for inputting each character

```
using System;    class KbIn { static void Main() { char ch;
                                  ch = (char) Console.Read();    Console.WriteLine("key is: " + ch);  }}
```

C#_1.18 The if Statement, Nested ifs, The if-else-if Ladder are same as C/C++/JAVA

C#_1.19 The SWITCH Statement, NESTED SWITCH Statements,

The general form of the **switch** statement is

```
switch(expression) { case constant1: statement sequence; break;
                     case constant2: statement sequence ; break;
                     case constant3: statement sequence ; break;
                      . . .
                     default: statement sequence ; break;
                   }
```

- The switch expression must be an integral type, such as ***char***, ***byte***, ***short***, or ***int***; an ***enumeration*** type; or type ***string***.

- ***No fall-through rule:*** In C#, it is an ***error*** for the statement sequence associated with ***one case to continue on into the next case***. This is called the ***no fall-through rule***. This is why ***case sequences*** end with ***break***. ***break*** causes program flow to exit from the ***entire switch*** statement and resume at the next statement ***outside the switch***.
  - ***default*** sequence must also not "fall through," and, usually ends with a ***break***.
  - You can have two or more case labels for the same code sequence,

```
switch(i) { case 1:
            case 2:
            case 3: Console.WriteLine("i is 1, 2 or 3"); break;
            case 4: Console.WriteLine("i is 4"); break; }
```

- In ***C***, ***C++***, and ***Java***, one case may continue on (that is, ***fall through***) into the next case. There are two reasons that C# instituted the ***no fall-through*** rule for cases.
  - First, it allows the order of the cases to be rearranged. Such a rearrangement would not be possible if one case could flow into the next.
  - Second, requiring each case to explicitly end prevents a programmer from accidentally allowing one case to flow into the next.
- ***Nested switch:***

```
switch(ch1) { case 'A': Console.WriteLine("This A is part of outer switch.");
                        switch(ch2) { case 'A': Console.WriteLine("This A is part of inner switch"); break;
                                      case 'B': /* . . .*/ }    break;        // end of inner switch
              case 'B': // ...
```

C#_1.20 For-loop and its variations, While, Do-While & Nested-loops are same as JAVA

C#_1.21 Continue and Break are same as C/C++, *C# doesn't support CONTINUE- BREAK LABEL (Java does)*

C#_1.22 goto-lebel Jump/loop is Supporetd by C# as C/C++

```
x = 1;
loop1:
x++;
if(x < 100) goto loop1;
```

- The ***goto*** does have one important ***restriction***: You ***cannot jump into a block***. Of course, you can jump ***out*** of a ***block***.
- In addition to working with "***normal***" labels, the ***goto*** can be used to ***jump*** to a ***case*** or ***default label*** within a ***switch***. For example, this is a valid switch statement:

```
switch(x) { case 1: // ...
                    goto default;
            case 2: // ...
                    goto case 1;
            default: /* . . . */ break;  }
```