ICS4U0 Unit 0 – Introduction to C# & Review of Essential Programming Concepts

SUMMARY OF UNIT 0

- **Elements of Programming**
  Variables, "If" Statements, Loops, Assignment Statements

- **Data Types**
  Numeric, String, Boolean, etc.

- **Working with Strings**

- **Structuring Data**
  Arrays and Lists

- **Object-Oriented Concepts**
  Objects, (Data) Fields, Properties, Methods, Static (Class) versus Instance

- **Event-Driven Programming**

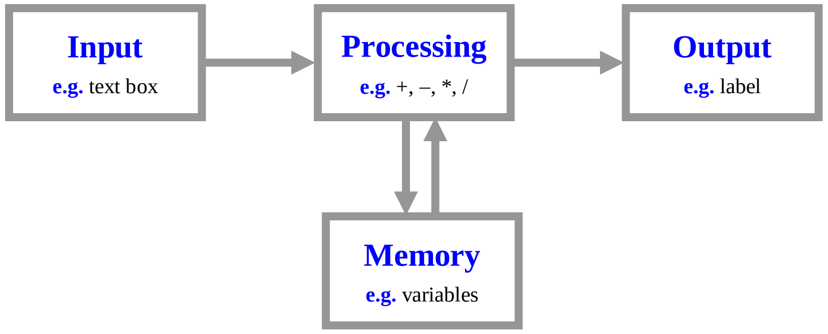
- **Examples of Problems to be Investigated**
  Greatest Common Divisor of Two Integers
  Generating all Prime Numbers from 1 to *n*
  Credit Card Number Validation
  Converting between Roman and Indo-Arabic Numbers

- **What the Heck is the .NET Framework?**

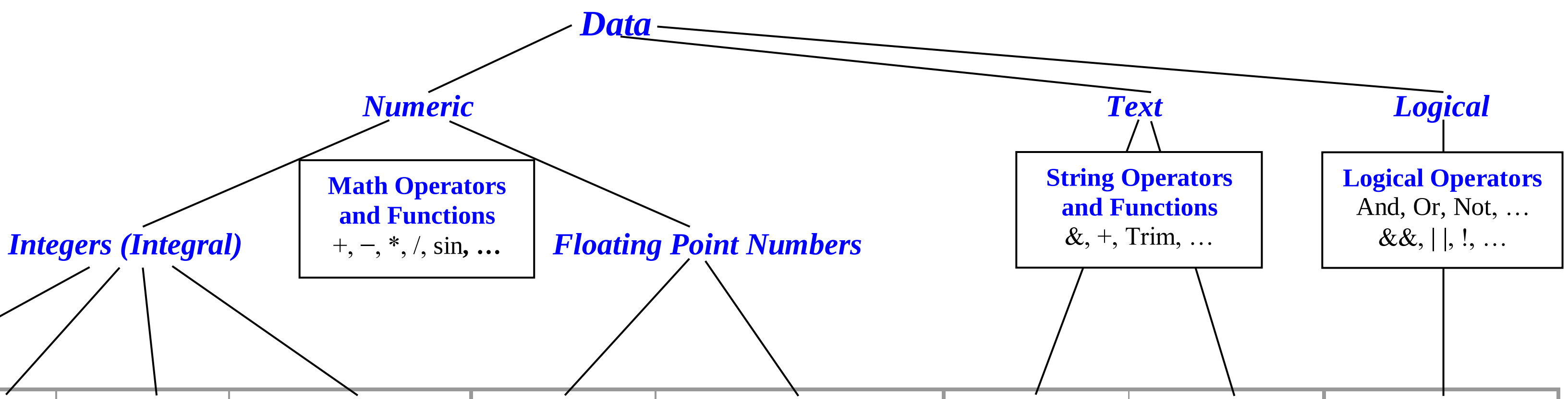
- **High-Level versus Low-Level Coding**

A Computer as a Data Processing Machine

A computer can be viewed as a ***data processing machine***. Since data come in various forms that require ***different amounts of memory***, ***different encoding schemes*** and ***different kinds of operations***, programming languages offer many diverse ***data types***.



The following diagram shows the ***some of the most commonly used*** data types in VB and C#.



VB (.Net)	**SByte** 1 byte storage –128 to 127 (-2^7 to 2^7-1)	**Byte** 1 byte storage 0 to 255 (0 to $2^8 - 1$)	**Integer** 4 bytes storage –2147483648 to 2147483647 (-2^{31} to $2^{31} - 1$)	**Long** 8 bytes storage –9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63} - 1$)	**Single** 4 bytes storage –3.4028235E38 to –1.401298E–45 for negative values; 1.401298E–45 to 3.4028235E38 for positive values	**Double** 8 bytes storage –1.79769313486231570E308 to –4.94065645841246544E–324 for negative values; 4.94065645841247E–324 to 1.79769313486232E308 for positive values	**String** Storage depends on platform 0 to about 2 billion Unicode characters	**Char** 2 bytes storage 0 to 65535 This type is actually an integral (integer) type. Each integer in the range represents a Unicode character.	**Boolean** Storage depends on platform **True** or **False**
C# (.Net)	**sbyte** Same as VB "SByte" type	**byte** Same as VB "Byte" type	**int** Same as VB "Integer" type	**long** Same as VB "Long" type	**float** Same as VB "Single" type	**double** Same as VB "Double" type	**string** Same as VB "String" type	**char** Same as VB "Char" type	**bool** Storage depends on platform **true** or **false**

Assignment Statements and Expressions

Definitions

An ***assignment statement*** takes the following form:

Identifier = Expression

Name of any programming entity that can be given a value.
e.g. Variable Name, Property Name, etc

Any statement that can be evaluated.
e.g. 0.13*Price

Assignment statements are ***used to give values to variables*** (or any programming entity that can have a value).

Examples

VB	***C#***

`'Increase the debt by the amount of money borrowed`
`TotalDebt = TotalDebt + MoneyBorrowed`
 	
`'Join given name to surname and assign to "FullName"`
`FullName = GivenName & " " & Surname`
	`//Increase the debt by the amount of money borrowed`
	`totalDebt = totalDebt + moneyBorrowed;`
	
	`//Join given name to surname and assign to "fullName"`
	`fullName = givenName + " " + surname;`

Exercises

1. Write assignment statements in both VB and C# for each of the following situations. (For your convenience, a table is given that shows some of the most commonly used operators in VB and C#.)

	Arithmetic Operators							***Relational (Comparison) Operators***						***Conditional Operators***		
VB	+	–	*	/	\	**Mod**	^	=	<	>	<=	>=	< >	**And**	**Or**	**Not**
C#	+	–	*	/	/	%	N/A	= =	<	>	<=	>=	!=	&&	| |	!

Task to Complete	***VB Assignment Statement***	***C# Assignment Statement***
(a) Increase by 1 the count of the number of vowels found in a string.		
(b) Decrease the bank balance by the amount of money withdrawn.		
(c) Calculate the number of whole hours in a given number of seconds.		
(d) Calculate the number of seconds remaining once all whole hours have been removed.		
(e) Copy the value of the variable “Position” to the variable “NewPosition.”		
(f) Change the value of the variable “Customer” to “Chris Rock.”		
(g) Triple the amount of money won by being a contestant on Jeopardy.		

2. Write assignment statements in both VB and C# for each of the following formulas. ***Remember to use MEANINGFUL variable names!*** (You'll have to do some research to find out how to use the square root and the power functions in C#.)

Formula	***VB Assignment Statement***	***C# Assignment Statement***
(a) $F = ma$		
(b) $F_G = -\frac{Gm_1m_2}{r^2}$		
(c) $A = \frac{bh}{2}$		
(d) $A = \frac{h(a+b)}{2}$		
(e) $E_0 = m_0c^2$		
(f) $A = \pi r^2$		
(g) $V = \frac{4}{3}\pi r^3$		
(h) $c = \sqrt{a^2+b^2}$		
(i) $m = \frac{m_0}{\sqrt{1-\frac{v^2}{c^2}}}$		

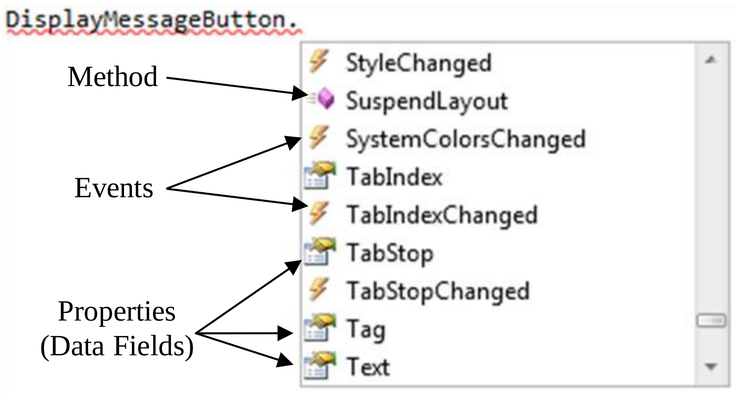
3. How many of the above formulas do you recognize? Explain as many as you can.

What is the Difference between an Object and a Variable?

Variables

- A ***variable*** has a ***very simple structure*** compared to an object.
- A ***variable*** is used ***only*** to store a ***single value*** in RAM.
- A ***variable*** can only store one value at a time. When a ***new value*** is assigned to a variable, its ***old value*** is overwritten with the new value.
- In VB.Net, if "`Option Explicit On`" is specified, variables must be ***declared explicitly***. (Keep in mind that it is a ***very bad idea*** to use "`Option Explicit Off`." Doing so will make it impossible for VB to detect misspelled variable names.)
- In C# and most other programming languages, variables must be ***declared explicitly***. Programming languages in which variables must be declared are called ***strongly-typed***.

Objects



- As shown in the diagram at the right, ***objects*** have a ***very complex structure*** compared to variables.
- An ***object*** is a collection of ***properties*** and ***methods***. (In object-oriented programming, the more general term for a property is "***data field***." Note also that the .NET framework includes ***events*** as members of certain objects.)
- An ***object can store many different values*** (properties, data fields) and ***can perform a variety of different actions*** (methods).
- Values can be assigned to the ***properties / data fields*** of objects ***but not to the objects themselves***.

Questions

1. Why are variables so important in programming?

2. What does it mean to declare a variable? What is a strongly-typed language?

3. An ***identifier*** is a name given to a program entity that can be used to refer to it. Identifiers include ***variable names***, ***object names***, ***function names*** and ***method names***. What are the rules for creating valid identifiers in VB and in C#?

Variable Declarations in C#

Main Idea

Variables are used to ***save*** information for later use, that is, at some later point in program execution.

Variables in the .NET Environment

In the .NET programming languages, all data types are actually implemented as objects, making the line between variables and objects somewhat blurry. In older object-oriented languages, there is a very sharp distinction between variables and objects (as outlined on page 7). Nonetheless, we can continue to think of variables in exactly the same manner as long as we keep in mind that in .NET we can call object methods using variable names, which is often not the case outside the .NET environment.

Examples

VB Variable Declarations	*Corresponding C# Variable Declarations*

`'In VB, the data type must be stated for each`
`'variable in the declaration statement.`
 	
`Dim Age As Integer, Weight As Integer`
	`//In C, C# C++ and Java, the data type name`
	`//is used to declare variables. It is`
	`//listed at the beginning of the`
	`//declaration statement EXACTLY ONCE!`
	`int age, weight;`

`'In older versions of VB, declaration and`
`'initialization had to be done in separate.`
`'statements. In VB.Net, variable declaration and`
`'variable initialization can be done in a single`
`'statement.`
`Dim DiscountRate As Single = 0.15`
`Dim ExchangeRate As Single = 1.24`
	`//In C, C#, C++ and Java, variables can be`
	`//initialized in a declaration statement.`
	`float discountRate=0.15, exchangeRate=1.24;`

`Dim upperCaseA As Char = "A"C`
 	
`Dim lowerCaseZ As Char = "T"C;`
 	
`Dim upperCaseA As Char = ChrW(&H41)`
 	
`Dim lowerCaseZ As Char = ChrW(&H7A)`
 	
`Dim upperCaseA As Char = ChrW(65)`
 	
`Dim lowerCaseZ As Char = ChrW(122)`
	`//The "char" data type is used to store`
	`//character codes for Unicode characters. The`
	`//values can be specified in a number of ways.`
	
	`//Method 1: Specify the character literal`
	`//enclosed in single quotation marks.`
	`char upperCaseA='A', lowerCaseZ='z';`
	
	`//Method 2: Specify the Unicode hexadecimal`
	`//representation.`
	`char upperCaseA='\u0041', lowerCaseZ='\u007A';`
	
	`//Method 3: Specify the hexadecimal`
	`//escape sequence representation.`
	`char upperCaseA='\x0041', lowerCaseZ='\x007A';`
	
	`//Method 4: Cast the integral decimal values.`
	`//See p.29 for more info on the cast operator.`
	`char upperCaseA=(char)65, lowerCaseZ=(char)122;`

`Dim Name As String = "Hollywood Blonde Jabroni"`	`string name = "Hollywood Blonde Jabroni";`
`Dim AnswerFound As Boolean = False`	`bool answerFound = false;`

Research Question

What is the purpose of variable declarations?

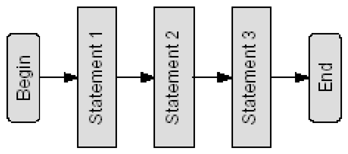
CONTROL FLOW – SEQUENCE, SELECTION AND REPETITION

Concepts

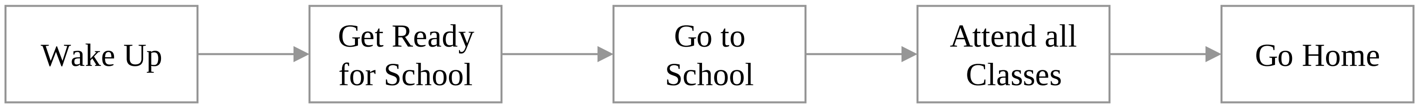
In computer science ***control flow*** (or alternatively, ***flow of control***) refers to the order in which the individual instructions are executed. As outlined below, there are ***three main control flow structures*** in most programming languages. Theoretical computer scientists have shown that the programming structures ***sequence***, ***selection*** and ***repetition*** are necessary and sufficient for being able to write programs to solve any ***computable*** problem, that is, any problem that can in principle be solved by a computer.

Sequence

When a program segment uses the principle of sequence, its execution proceeds in a ***linear fashion***. That is, the statements are executed in sequence, one after the other. This is similar to walking through a completely enclosed tunnel. You must continue walking through the tunnel until you find an exit.

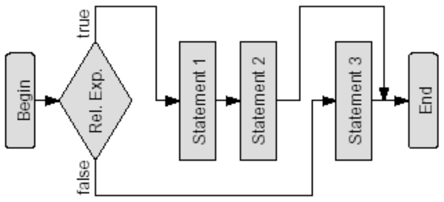


The following is an example of ***sequence*** in everyday life.

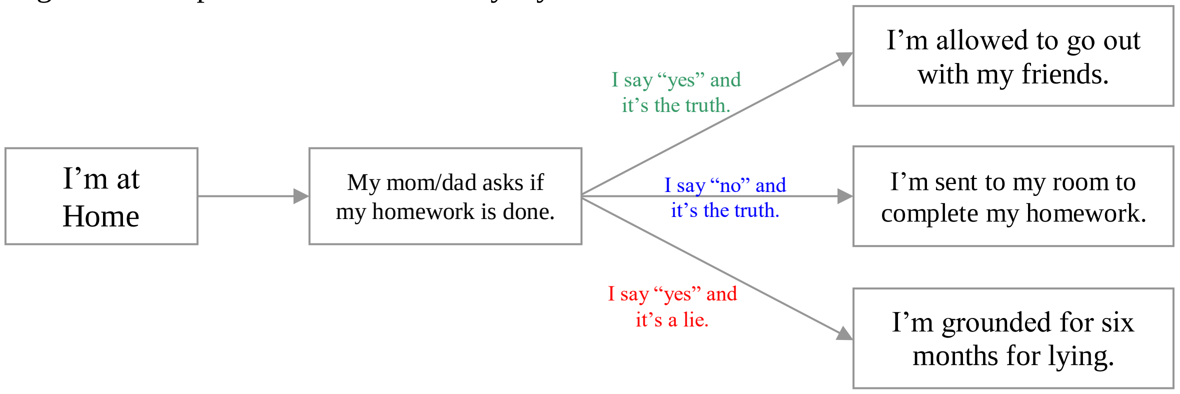


Selection ("If" Statements)

When a program segment uses the principle of selection, its execution ***does not*** proceed in a ***linear fashion***. One or more groups of statements are given as ***possible*** statements to be executed. Based on a ***condition*** or a ***set of conditions***, one group of statements is ***selected*** and ***executed*** while the others are ignored. This is similar to walking through a completely enclosed tunnel and suddenly reaching a point at which the tunnel branches into two or more tunnels. In this case, you must ***choose*** or ***select*** which tunnel to follow.



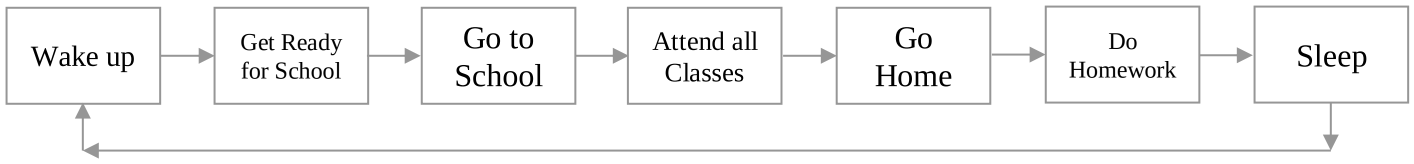
The following is an example of ***selection*** in everyday life.



I'm at Home
My mom/dad asks if my homework is done.
I say "yes" and it's the truth.
I'm allowed to go out with my friends.
I say "no" and it's the truth.
I'm sent to my room to complete my homework.
I say "yes" and it's a lie.
I'm grounded for six months for lying.

Repetition (Loops)

When repetition is employed, a program segment is executed in a ***repetitive fashion***. A group of statements is repeated ***a certain number of times*** (***counted loop***), ***while*** a certain condition is true (***conditional while loop***) or ***until*** a certain condition is true (***conditional loop until***). This is similar to jogging around the block a ***certain number of times***, ***while*** you feel energetic or ***until*** fatigue sets in. The following is an example of ***repetition*** in everyday life.



“If” Statement Details

VB “If” Statement Examples	*C# “if” Statement Examples*

`If Mark >= 80 Then`
`   Message = "Wow! What a great effort!"`
`End If`
	`if (mark >= 80)`
	`   message = "Wow! What a great effort!";`

`If Mark >= 80 Then`
`   Message = "Wow! What a great effort!"`
`Else`
`   Message = "Keep trying to improve!"`
`End If`
	`if (mark >= 80)`
	`   message = "Wow! What a great effort!";`
	`else`
	`   message = "Keep trying to improve!";`

`If Mark >= 80 And Mark <= 100 Then`
`   Message = "Wow! What a great effort!"`
`ElseIf Mark >= 70 Then`
`   Message = "Pretty good! Keep it up!"`
`ElseIf Mark >= 60 Then`
`   Message = "Not bad, you’re getting there!"`
`ElseIf Mark >= 50 Then`
`   Message = "You’re skating on thin ice!"`
`ElseIf Mark >= 0 And Mark <50 Then`
`   Message = "Get your butt in gear dude!"`
`Else`
`   Message = "What are you on dude?"`
`End If`
	`if (mark >= 80 && mark <= 100)`
	`   message = "Wow! What a great effort!";`
	`else if (mark >= 70)`
	`   message = "Pretty good! Keep it up!";`
	`else if (mark >= 60)`
	`   message = "Not bad, you’re getting there!";`
	`else if (mark >= 50)`
	`   message = "You’re skating on thin ice!";`
	`else`
	`   message = "What are you on dude?";`

`If Temperature >= 25 Then`
`   Message1 = "Time to wear shorts!"`
`   Message2 = "Turn on the AC dude!"`
`ElseIf Temperature >= 10 And Temperature < 25 Then`
`   Message1 = "Too cold for shorts but still OK!"`
`   Message2 = "Turn off the AC dude!"`
`ElseIf Temperature >= 0 And Temperature < 10 Then`
`   Message1 = "Time to get out the winter coats!"`
`   Message2 = "Turn on the heat man!"`
`Else`
`   Message1 = "Holy crap it’s cold!"`
`   Message2 = "Don’t turn off the heat!"`
`End If`
	`if (temperature >= 25)`
	`{`
	`   message1 = "Time to wear shorts!";`
	`   message2 = "Turn on the AC dude!";`
	`}`
	`else if (temperature >= 10 && temperature < 25)`
	`{`
	`   message1="Too cold for shorts but still OK!";`
	`   message2="Turn off the AC dude!";`
	`}`
	`else if (temperature >= 0 && temperature < 10)`
	`{`
	`   message1="Time to get out the winter coats!";`
	`   message2="Turn on the heat man!";`
	`}`
	`else`
	`{`
	`   message1 = "Holy crap it’s cold!";`
	`   message2 = "Don’t turn off the heat!";`
	`}`

Research Question

Why are braces (i.e. “{“ and “}”) used in C# “**`if`**” statements? Are they always necessary?

Loop Details

VB Loop Examples / C# Loop Examples

```
Sum = 0
For X As Integer = 1 To 5
   Sum = Sum + X
Next X
```

```
//As with "if" statements, braces are not needed
//if there is only one statement to be executed
sum = 0;
for (int x=1; x<= 5; x++)
   sum = sum + x;
```

x++ is a shortcut for x=x+1

++x is also a shortcut for x=x+1
There is a subtle difference between the two that will become clear later in the course.

```
Sum = 0
Count = 0
For X As Integer = 1 To 5
   Sum = Sum + X
   Count = Count + 1
Next X
Average = Sum / Count
```

```
sum = 0;
count = 0;
for (int x=1; x<= 5; ++x)
{
   sum += x; //Shortcut for sum=sum+x
   count += 1; //Shortcut for count=count+1
}
average = sum/count;
```

```
Sum = 0
Count = 99
For Num As Integer = 3 To Count Step 3
   Sum = Sum + Num
   Score = Score - Sum
   If Score Mod 2 = 0 Then
      Message = "Even Score"
   Else
      Message = "Odd Score"
   End If
Next Num
```

```
sum = 0;
count = 10;
for (int num=3; num<=count; num+=3)
{
   sum += num;
   score -= sum;
   if (score % 2 == 0)
      message = "Even Score";
   else
      message = "Odd Score";
}
```

```
'Euclid's method for computing
'the gcd of two integers
a = 356
b = 512
Do
   remainder = a Mod b
   a = b
   b = remainder
Loop Until b = 0
gcd = a
```

```
//There is no "until" keyword in C#. "Until
//b is equal to zero" is logically equivalent to
//"while b is not equal to zero."
a = 356;
b = 512;
do
{
   remainder = a % b;
   a = b;
   b = remainder;
}while (b != 0);
gcd = a;
```

```
'Calculate the number of times '2' divides into
'"Num" before a quotient of 0 is obtained
NumDivisionsByTwo = 0
Num = Num \ 2
Do While Num > 0
   NumDivisionsByTwo = NumDivisionsByTwo + 1
   Num = Num \ 2
Loop
```

```
//Calculate the number of times '2' divides into
//"Num" before a quotient of 0 is obtained
numDivisionsByTwo = 0;
num/=2;
while (num > 0)
{
   numDivisionsByTwo++;
   num/=2;
}
```

More VB Loop Examples

```
Sum = 0
For X = 0 To 1000 Step 10
   Sum = Sum + X
Next X
```

```
Sum = 0
Count = 0
For X = 10000 To 0 Step -10
   Sum = Sum + X
   Count = Count + 1
Next X
Average = Sum / Count
```

More C# Loop Examples

```
//As with "if" statements, braces are not needed
//if there is only one statement to be executed
sum = 0;
for (x=0; x<=1000; x+=10)
   sum = sum + x;
```

This is a shortcut for x=x+10

```
sum = 0;
count = 0;
for (x=10000; x>=0; x-=10)
{
   sum += x; //Shortcut for sum=sum+x
   count++; //Shortcut for count=count+1
}
average = sum/count;
```

This is a shortcut for x=x-10

Important Exercises on Loops and If Statements

1. As shown in the example, complete a memory map for each loop. In addition, state the purpose of each loop.

```
int count = 0;
int sum = 0;
for (int x=1; x<= 5; x++)
{
   sum += x;
   count++;
}
```

Values Before Entering Loop

Values After Exiting Loop

x	sum	count
–	0	0
1	1	1
2	3	2
3	6	3
4	10	4
5	15	5
–	15	5

The ***purpose*** of the given C# "for" loop is to

- add the integers from 1 to 5 inclusive (i.e. 1+2+3+4+5)
- count the number of integers that were added (5)

These values are stored using the variables "sum" and "count" respectively.

Loop	*Memory Map*	*Problem Solved*

```
int sum = 0;
int count = 0;
for (int x=1000; x>0; x-=200)
{
   sum += x;
   count++;
}
average = sum/count;
```

x	sum	count

The purpose of the given "**for**" loop is to

____

____

____

```
int a = 356;
int b = 512;
do
{
   int remainder = a % b;
   a = b;
   b = remainder;
}while (b != 0);
gcd = a;
```

The given "**do**" loop implements the

____

algorithm for computing the ______

____

```
int num=1023;
int numDivisionsByTwo = 0;
num/=2;
while (num > 0)
{
   numDivisionsByTwo++;
   num/=2;
}
```

The purpose of the given "**while**" loop is

____

____

____

____

2. ***On paper***, write C# loops to perform each of the following tasks. Do not use a computer for this question except for verifying that your code is correct.

(a) Add up the numbers 1 + 2 + 3 + 4 + ... until the **Sum** > **100**.

(b) Determine how many numbers 2 +4 + 6 + 8 + ... are needed to give a **Sum** > **1000**.

(c) Determine the sum of all powers of 2 (i.e. 1, 2, 4, 8, 16, 32, …) that are less than 1000000.

(d) Output the smallest number (other than 1) that divides evenly into 2701.

3. The ancient Greek civilization had a keen interest in philosophy, mathematics, science, literature and the pursuit of knowledge for its own sake. In fact, the Greeks had much less interest in the practical applications of their intellectual inquiries because they believed that nature existed primarily for the wonderment of humans. Nature was there to be explored, contemplated and even worshipped, but not to be tampered with or altered. Largely due to this attitude, in a few short centuries the Greeks developed a body of knowledge and a system of rational thought that was unrivalled in ancient times. In fact, it was the rediscovery of ancient Greek learning that spawned the Renaissance, a pivotal period without which our modern technological society probably would not exist.

Among other things, the Greeks were interested in the properties of numbers, including numbers that the Greeks called ***perfect***. An integer is called ***perfect*** if the sum of its ***proper divisors*** is equal to the number itself. Two examples of perfect numbers are 6 and 28 because 6 = 1 + 2 + 3 and 28 = 1 + 2 + 4 + 7 + 14.

(a) What is a ***proper divisor*** of a number?

(b) Without writing a program, find a perfect number greater than 28.

(c) Consider the steps that you used in question 3(b) to find a perfect number greater than 28. Without referring to specific numbers, list steps that can be followed to determine whether an integer is perfect.

(d) Write a program that can determine whether a given number is perfect.

(e) Write a program that finds ***all*** perfect numbers less than 10000.

(f) The numbers 220 and 284 are called an ***amicable pair*** because the sum of the proper divisors of 220 is 284 and the sum of the proper divisors of 284 is 220. Write a C# program that determines whether a given pair of integers forms an amicable pair.

(g) Write a program that finds ***all*** amicable pairs less than 10000.

4. To a mathematician, a prime number serves the same purpose as a chemical element does to a chemist. Just as all molecules are made using the elements of the periodic table, all numbers can be constructed using nothing but primes. Another way of putting this is that the primes are the basic building blocks of all numbers. Prime numbers are not just a whimsical curiosity of mathematicians. Without them, it would not be possible to conduct secure transactions over the Internet! (The details of how prime numbers ensure the security of Web transactions will follow in a future unit and from time to time in class discussions. For more information consult http://en.wikipedia.org/wiki/Public_key_encryption)

(a) What is a prime number?

(b) Rewrite your definition in 4(a) using the concept of proper divisor.

(c) Explain how you could use your code for finding all proper divisors of a number to determine whether a given number is prime. Would this be an efficient method?

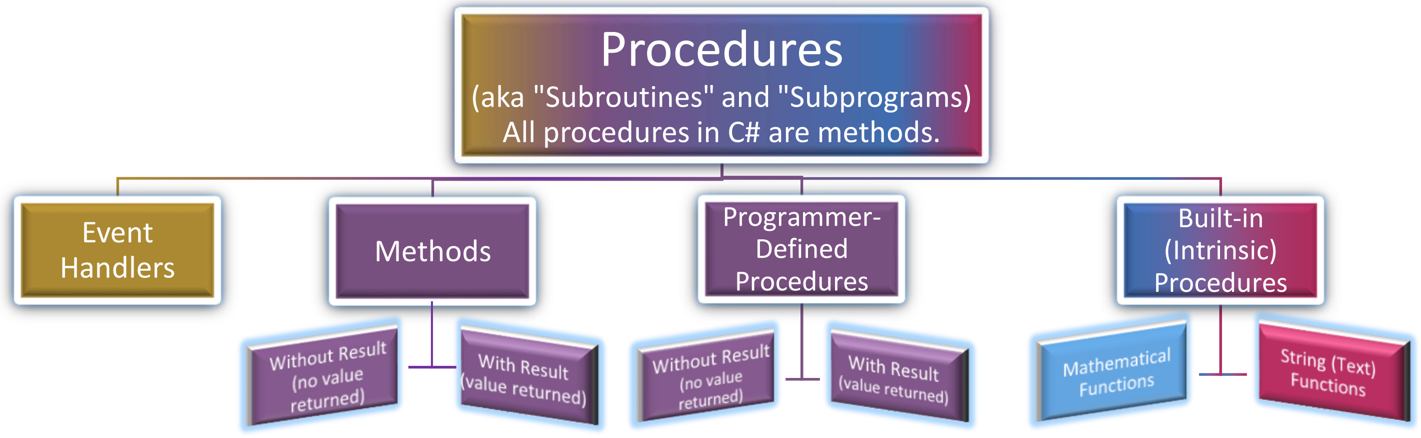
(d) Write a C# program that can determine whether a given number is prime. (When you test your program, use relatively small integers as input. Otherwise, you may spend a great deal of time waiting for your program to produce its output.)

(e) Primes are considered the building blocks of all numbers because every integer can be written as a product of primes. For example, 24 = 2(2)(2)(3) = 2^3(3) and 105 = 3(5)(7). Write a C# program that can find the prime factorization of a given integer. (Don't forget to observe the ***caveat*** of 4(d).)

PROCEDURES IN C#

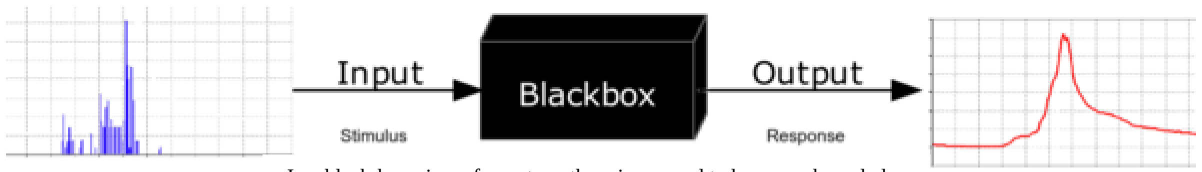
Overview of Procedures

- **Procedure (aka "Subroutine" and "Subprogram")**
  A procedure is a ***named*** set of programming instructions.
  The instructions can be executed whenever needed simply by specifying the ***name*** of the procedure.
- **Procedure Call**
  Calling a procedure means to use the procedure's name to execute the instructions contained within the body of the procedure.
- **Function**
  A procedure that returns a value (i.e. has an "output" value) is often called a ***function***. In C, all procedures are called functions, even those that don't return a value.
- **Method**
  A procedure that belongs to a class or an object is called a ***method***. In C#, ***all*** procedures are defined within classes, meaning that ***all procedures are methods***.
- **Programmer-Defined Procedure**
  A procedure that ***is not built in*** to a programming language but is created by a user of the language.
- **Intrinsic ("Built-in") Procedure**
  A procedure that ***is built in*** to a programming language.


Procedures
(aka "Subroutines" and "Subprograms)
All procedures in C# are methods.
Event Handlers
Methods
Programmer-Defined Procedures
Built-in (Intrinsic) Procedures
Without Result (no value returned)
With Result (value returned)
Without Result (no value returned)
With Result (value returned)
Mathematical Functions
String (Text) Functions

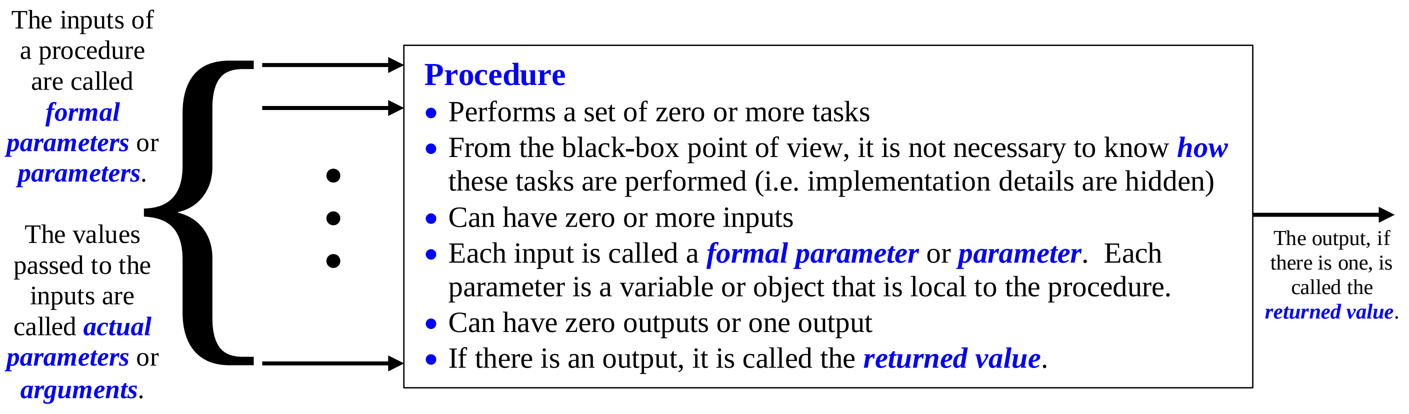
"Black-Box" View of Procedures

In science, computing, and engineering, a **black box** is a device, system or object which can be viewed in terms of its inputs and outputs (or transfer characteristics), ***without any knowledge*** of its internal workings. Its implementation is "opaque" (black). Almost anything might be referred to as a black box: a transistor, algorithm, or the human brain. (Source: https://en.wikipedia.org/wiki/Black_box)



In a black-box view of a system, there is no need to have any knowledge of the internal workings of the system. The behaviour of the system is completely characterized by its inputs and corresponding outputs.

(Source: https://en.wikipedia.org/wiki/Black_box)



The inputs of a procedure are called ***formal parameters*** or ***parameters***.

The values passed to the inputs are called ***actual parameters*** or ***arguments***.

Procedure

- Performs a set of zero or more tasks
- From the black-box point of view, it is not necessary to know ***how*** these tasks are performed (i.e. implementation details are hidden)
- Can have zero or more inputs
- Each input is called a ***formal parameter*** or ***parameter***. Each parameter is a variable or object that is local to the procedure.
- Can have zero outputs or one output
- If there is an output, it is called the ***returned value***.

The output, if there is one, is called the ***returned value***.

Examples of Programmer-Defined Methods in C#

Procedure Definition

```
/**
 * "isDivisor" returns "true" if "n" is a divisor of "m"
 * and returns "false" otherwise
 */
```

return type

name of method

```
private bool isDivisor(int m, int n)
{
  if (m % n == 0)
    return true;
  else
    return false;

}//end of "isDivisor"
```

(formal) parameters

private
This means that the procedure is only available to the ***class*** in which it is defined.

method
Any procedure defined within a class is called a ***method***. In C#, all procedures are methods.

```
/**
 * "gcd" uses the Euclidean algorithm
 * to compute the greatest common
 * divisor of "m" and "n"
 */
private int gcd(int m, int n)
{
  do
  {
     int remainder = m % n;
     m = n;
     n = remainder;
  }while (n != 0);
  return m;
}//end of "gcd"
```

Mathematical Basis of the Euclidean Algorithm

If a and b are integers, then

$$a = bq + r$$

for integers q (quotient) and r (remainder).

This means that any number that divides a and b exactly must also divide r exactly. Therefore,

$$\gcd(a, b) = \gcd(b, r)$$

The method shown at the left exploits this property of gcd.

Example Calls

```
bool x = isDivisor(24, 7);

int a = 20, b = 30;
if (isDivisor(a, b))
   b = a;
else
   a = b;
```

actual parameters
(aka ***arguments***)

```
int x = gcd(24, 8);

int a = 10, b = 30;
if (gcd(a, b) == a)
   b = a;
else
   a = b;
```

Procedure Definition

```
/**
 * "nextHighestPalindromeButton_Click" is a C# event-handling method
 * (i.e. an event handler).  Notice that its return type is "void."
 * This means that the method does not return a value (i.e. no output).
 */
```

return type

name of event handler

(formal) parameters

```
private void nextHighestPalindromeButton_Click(object sender, EventArgs e)
{
   int num = Convert.ToInt32(integerTextBox.Text);
   answerLabel.Text = nextHighestPalindrome(num).ToString();

}//end of "nextHighestPalindromeButton_Click"
```

call of programmer-defined method

Parameters

sender: object that contains information about the object on which event took place

e: object that contains information about the event that took place

Example Calls

Normally, event handlers are called automatically by a system called the ***event monitor***.

Although it's rarely done in practice, event handlers can also be called in the same way as all other methods are called. All that is required is that arguments of the right type be supplied.

Advantages of Procedures

1. Eliminate repetition of code and thereby reduce program size

2. Allow for large, complex problems to be broken down logically into a series of smaller, simpler problems

3. Make programs easier to read and understand

4. Improve portability: properly designed procedures can be used in any program without modification

5. Debugging is easier because bugs are localized to procedures

Problems

Write C# methods to solve each of the following problems. Keep in mind the following important points:

- Parameters are always local to the method in which they are defined.
- Ensure that each method that you write is completely self-contained. That is, each method must ***not*** refer to any global information whatsoever.

1. Write a method that determines whether a given number is prime.

2. Write a method that computes the discriminant of any quadratic function.

3. Write a method that solves any linear equation.

4. Write a method that computes the distance between any two points on a two-dimensional Cartesian plane.

5. Write a method that determines whether any given year (Gregorian calendar) since 1582 is a leap year. (See https://www.wwu.edu/skywise/leapyear.html)

6. Write a method that takes any Gregorian calendar date and returns the day of the week of that date. For example, if the input is February 17, 2016, the method returns "Wednesday." (See http://www.timeanddate.com/date/doomsday-weekday.html and https://en.wikipedia.org/wiki/Doomsday_rule)

Alternative Method of Brace Placement

Examples

Format suggested when Clarity is the Primary Concern

```
if (temperature >= 25)
{
   message1 = "Time to wear shorts!";
   message2 = "Turn on the AC dude!";
}
else if (temperature >= 10 && temperature < 25)
{
   message1="Too cold for shorts but still OK!";
   message2="Turn off the AC dude!";
}
else if (temperature >= 0 && temperature < 10)
{
   message1="Time to get out the winter coats!";
   message2="Turn on the heat man!";
}
else
{
   message1 = "Holy crap it's cold!";
   message2 = "Don't turn off the heat!";
}
```

```
sum = 0;
count = 0;
for (int x=1000; x>=0; x-=100)
{
   sum += x;
   count++;
}
average = sum/count;
```

```
a = 356;
b = 512;
do
{
   remainder = a % b;
   a = b;
   b = remainder;
}while (b != 0);
gcd = a;
```

```
num=1023;
numDivisionsByTwo = 0;
num/=2;
while (num > 0)
{
    numDivisionsByTwo++;
    num/=2;
}
```

Format suggested when Brevity is the Primary Concern

```
if (temperature >= 25){
   message1 = "Time to wear shorts!";
   message2 = "Turn on the AC dude!";
}
else if (temperature >= 10 && temperature < 25){
   message1="Too cold for shorts but still OK!";
   message2="Turn off the AC dude!";
}
else if (temperature >= 0 && temperature < 10){
   message1="Time to get out the winter coats!";
   message2="Turn on the heat man!";
}
else{
   message1 = "Holy crap it's cold!";
   message2 = "Don't turn off the heat!";
}
```

```
sum = 0;
count = 0;
for (int x=1000; x>=0; x-=100){
   sum += x;
   count++;
}
average = sum/count;
```

```
a = 356;
b = 512;
do{
   remainder = a % b;
   a = b;
   b = remainder;
}while (b != 0);
gcd = a;
```

```
num=1023;
numDivisionsByTwo = 0;
num/=2;
while (num > 0){
    numDivisionsByTwo++;
    num/=2;
}
```

Dependent and Independent "if" Statement Structures

See the complete program: S:\Out\Nolfinator\Ics4uo\CSharpExamples\PizzaIfStatementApplication

```
private void summarizeOrderButton_Click(object sender, EventArgs e)
{// Start "summarizeOrderButton_Click" Method

    // LOCAL VARIABLES
    string pizzaSize, pizzaDough, toppings="";

    // ONE "if" statement with multiple clauses is used to determine the pizza size selected by the user.
    // This structure is used when only ONE option can be chosen from 2 or more possibilities.
    if (smallRadioButton.Checked)
        pizzaSize = "small";
    else if (mediumRadioButton.Checked)
        pizzaSize = "medium";
    else if (largeRadioButton.Checked)
        pizzaSize = "large";
    else if (extraLargeRadioButton.Checked)
        pizzaSize = "extra large";
    else if (partySizeRadioButton.Checked)
        pizzaSize = "party size";
    else
        pizzaSize = "super size";
```

This is a single "**if**" statement with multiple clauses. Its purpose is to set the value of the variable "pizzaSize." Only one "**if**" statement is needed because only one radio button can be selected.

```
    // ONE "if" statement with multiple clauses is used to determine the pizza dough selected by the user.
    // This structure is used when only ONE option can be chosen from 2 or more possibilities.
    if (enrichedWhiteRadioButton.Checked)
        pizzaDough = "enriched white";
    else if (wholeWheatRadioButton.Checked)
        pizzaDough = "whole wheat";
    else if (multiGrainRadioButton.Checked)
        pizzaDough = "multi-grain";
    else if (flaxRadioButton.Checked)
        pizzaDough = "flax";
    else if (spinachRadioButton.Checked)
        pizzaDough = "spinach";
    else
        pizzaDough = "rye";
```

This is a single "**if**" statement with multiple clauses. Its purpose is to set the value of the variable "pizzaDough." Only one "**if**" statement is needed because only one radio button can be selected.

```
  // Determine which toppings, if any, have been chosen. Each topping requires its own "if" statement
  // because the choice of any topping is INDEPENDENT of the choice of any other topping.
    if (pepperoniCheckBox.Checked)
        toppings = "pepperoni, ";
    if (mushroomsCheckBox.Checked)
       toppings += "mushrooms, ";
    if (greenPeppersCheckBox.Checked)
        toppings += "green peppers, ";
    if (hotPeppersCheckBox.Checked)
        toppings += "hot peppers, ";
    if (anchoviesCheckBox.Checked)
        toppings += "anchovies, ";
    if (pineapplesCheckBox.Checked)
        toppings += "pineapples, ";
```

Each check box requires a separate "**if**" statement because each check box is independent of all the rest.

```
                              .
                              .
                              .
```

The rest of the "if" statements have been omitted to make it possible to fit the entire method on one page.

```
    // Remove the final comma and space at the end of the 'toppings' string if at least one topping was chosen.
    if (toppings != "")
        toppings = toppings.Substring(0, toppings.Length - 2);
    else
        toppings = "no toppings chosen";

    // OUTPUT
    orderSummaryLabel.Text = "Let me get this straight. You would like a " + pizzaSize + " pizza made with "
                                                  + pizzaDough + " dough and with the following toppings: " + toppings + ".";
} //End of "summarizeOrderButton_Click" Method
```

Questions

1. How many "**if**" statements are there in the "summarizeOrderButton_Click" method? Why are so many "**if**" statements needed?

2. The name of the method on the previous page is "summarizeOrderButton_Click." What is the significance of the names "summarizeOrderButton" and "Click?"

3. Using the pizza order confirmation program as a model, create your first C# program. Instead of choosing a pizza size, type of dough and toppings, your program will allow the user to choose a car make, model and options.

Working with Strings in C#

Examples of Declaration of String "Variables"

Strings in C# are actually implemented as objects. However, the C# syntax for working with strings allows us to think of them as variables. Here are some basic examples of how the **string** data type can be used in C#.

```
string surname = ""; //Set the initial value of the string variable 'surname' to the null string

string givenName = "Rags"; //Set the initial value of the string variable 'givenName' to "Rags"

string secondsText = secondsTextBox.Text; //Set initial value to the text in 'secondsTextBox.'
```

Working with Strings

Once you have created a string variable, you can work with it using the "+" ***string concatenation operator*** and any of the methods in the .NET String class (described in more detail on pages 21 to 23). Since the number of methods in each class tends to be large, it is ***not advisable*** to ***memorize*** the names and purposes of each method. Instead, you should consult sources such as MSDN (Microsoft Development Network – msdn.microsoft.com), or simply "Google" the information that you need to find.

The ***index*** or ***subscript*** of a character in a string is a number that identifies the ***position*** of the character in the string.

What a String Really "Looks Like"

As shown in the following example, a string's value is stored as an array of **`char`** values:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
'N'	'o'	'l'	'f'	'i'	' '	'h'	'a'	't'	'e'	's'	' '	'l'	'a'	'z'	'i'	'n'	'e'	's'	's'	'\0'

Notice that the special character `'\0'`, known as the ***null character*** or ***terminating character***, is used to mark the end of a string. Although a combination of two symbols is used to denote the terminating character, it only counts as a single character. As shown in the diagram below, its Unicode numeric value is zero.

Special characters like `'\0'` are known as ***escape sequences*** or ***control sequences***. See http://en.wikipedia.org/wiki/Escape_sequence for a good description of escape sequences.

As you probably know, the Unicode value (visit www.unicode.org to find out more) of each character is what is actually stored. Therefore, the array really should look like the following:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
78	111	108	102	105	32	104	97	116	101	115	32	108	97	122	105	110	101	115	115	0

Since a **char** value occupies two bytes of memory, each element of the array actually corresponds to two memory locations. (Also, in C#, the actual Unicode values are specified in the format '\u####' as shown in the example on page 8. This was not shown here due to lack of space.)

Now this is not all there is to a string in C#. The .NET "String" class provides a number of ***methods*** that can be used to manipulate strings:

```
Compare, CompareOrdinal, Concat, CompareTo, Contains, CopyTo, EndsWith, Equals, Format,
GetEnumerator, GetHashCode, GetType, IndexOf, IndexOfAny, Insert, IsNormalized, Intern,
IsInterned, IsNullOrEmpty, IsNullOrWhiteSpace, Join, LastIndexOf, LastIndexOfAny,
Normalize, PadLeft, PadRight, ReferenceEquals, Remove, Replace, Split, StartsWith,
SubString, ToCharArray, ToLower, ToLowerInvariant, ToString, ToUpper, ToUpperInvariant,
Trim, TrimEnd, TrimStart
```

The .NET "String" class also provides two ***properties*** (data fields) for strings:

```
Length, Chars
```

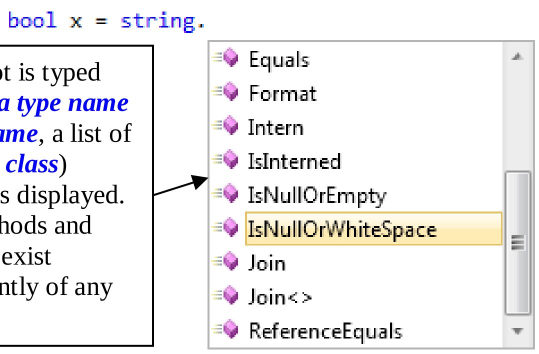
Static (Class) versus Instance

As already pointed out on the previous page, the .NET "String" class provides numerous methods, as well as two properties, that can be used to manipulate strings. To gain a full appreciation of this class, however, it is first necessary to examine the differences between ***static*** and ***instance*** members of a class.

Static Methods and Data Fields/Properties

These are methods or properties/data fields that can be accessed directly from a class ***without*** creating an object.

Example



```
bool x = string.
```

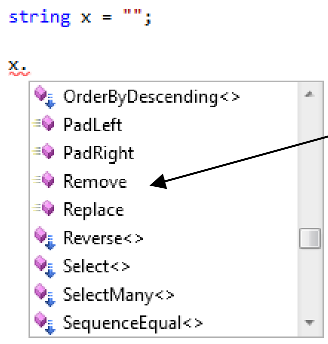
When a dot is typed after a ***data type name*** or ***class name***, a list of ***static*** (aka ***class***) members is displayed. ***Static*** methods and properties exist independently of any object.

```
//The value of 'x' will be 'true.'
//Note that the method 'IsNullOrEmpty' is
//used without creating a string object.
bool x = string.IsNullOrEmpty("");

//The value of 'y' will be 'false.'
//Once again, the method 'IsNullOrEmpty' is
//used without creating a string object.

bool y = string.IsNullOrEmpty(" ");
```

Instance Methods and Data Fields/Properties

These are methods or data fields that can ***only*** be accessed when an object is created. In a sense, these methods or data fields "belong" to the object.

Example



```
string x = "";

x.
```

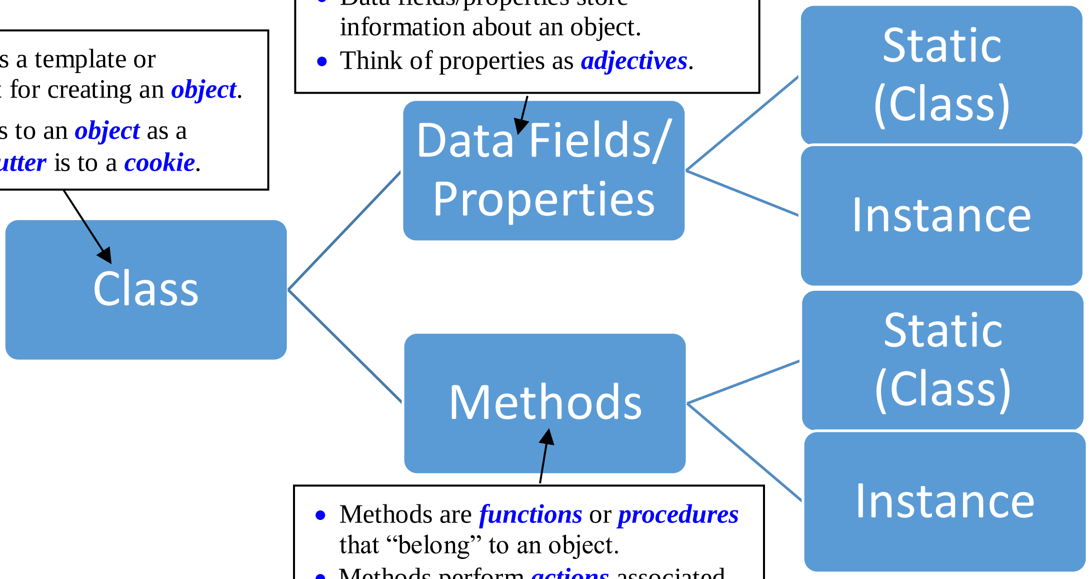
In this example, a string variable called "x" (which is really a .NET "String" object) is created first. When a dot is typed after the object name "x," a list of ***instance*** members is displayed. An ***instance*** method or property always refers to a particular object. Without the object, it does not make any sense to use the property or method.

```
//This time 'x' is a string object.
string x = "Toronto Maple Leafs";

//The following shows an example call of the
//instance method 'IndexOf.' Note that it
//does not make sense to call 'IndexOf'
//independently of 'x.'
//The value of 'y' will be 12 because the first
//'e' in the string 'x' is found at index 12.
int y = x.IndexOf("e");
```

Summary

- A ***class*** is a template or blueprint for creating an ***object***.
- A ***class*** is to an ***object*** as a ***cookie cutter*** is to a ***cookie***.

- Data fields/properties are ***variables*** or ***objects*** that "belong" to an object.
- Data fields/properties store information about an object.
- Think of properties as ***adjectives***.



- Methods are ***functions*** or ***procedures*** that "belong" to an object.
- Methods perform ***actions*** associated with the object.
- Think of methods as ***verbs***.

Demystifying MSDN Technical Information: Getting to Know the .NET String Class

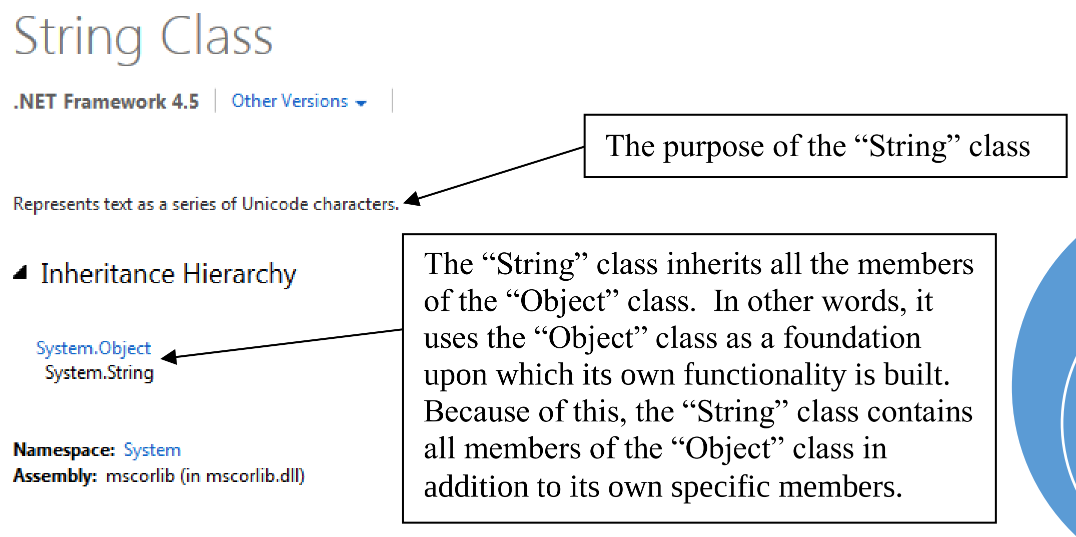
What is MSDN?

- MSDN stands for "Microsoft Developer Network."
- MSDN's URL is http://msdn.microsoft.com.
- MSDN contains a wealth of technical information on how to use Microsoft's various software development tools. (**e.g.** Microsoft Visual Studio, .NET Framework, etc.)
- The reference materials provided by MSDN are highly technical and as such, can be difficult to understand.
- The information presented below is intended to help make technical documents easier to comprehend.

Using MSDN to understand the Structure of the .NET String Class

A logical way to begin searching for information on the .NET string class is to Google ".NET string class." Doing so returns ***millions of results***, the first of which is http://msdn.microsoft.com/en-us/library/system.string.aspx. Shown below are a few of the main features of this page.

Introduction to the Class



String Class

.NET Framework 4.5 | Other Versions

Represents text as a series of Unicode characters.

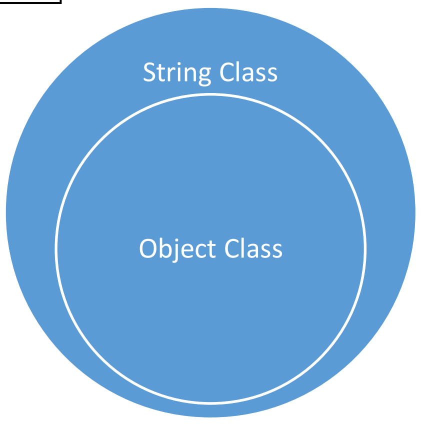
Inheritance Hierarchy

System.Object
System.String

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

The purpose of the "String" class

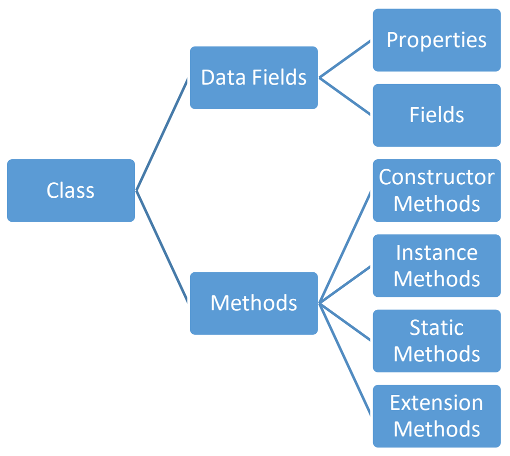
The "String" class inherits all the members of the "Object" class. In other words, it uses the "Object" class as a foundation upon which its own functionality is built. Because of this, the "String" class contains all members of the "Object" class in addition to its own specific members.



Detailed Description of Members of the Class

What follows the introduction to the class is a complete listing of all constructor methods, instance methods, static methods, extension methods, properties and fields. The following table describes how this information is organized as well as the meanings of the various icons that are used in the descriptions of the members.

How the Information is Organized



Meanings of the Various Icons

- Public Method
- Private Method
- Property
- Field
- Static Method
- Extension Method
- Supported by Portable Class Library
- Supported by XNA Framework
- Supported in .NET for Windows Store Apps

Several Helpful Methods found within the .NET String Class

For each of the following …

- …state whether the method is a static method or an instance method
- …describe its purpose
- …give an example of how it could be used.

Method Name	*Static or Instance?*	*Purpose*	*Example*
Compare			
CompareTo			
Contains			
CopyTo			
EndsWith			
Equals			
IndexOf			
IsNullOrWhiteSpace			
PadLeft			
PadRight			
Remove			
Replace			
StartsWith			
SubString			
ToLower			
ToString			
ToUpper			
Trim			
TrimEnd			
TrimStart			

Exercises involving Strings

1. Create a memory map for each code segment. In addition, determine the problem that is solved in each case. (Some variables have intentionally been given silly names to disguise their purpose.)

Code Segment	*Memory Map (Trace Chart)*	*Problem Solved?*

`int harinder=0;`
`string c="";`
`string s="aeiouAEIOU";`
`string a="Laziness is for fools!";`
`for (int i=0; i<a.Length; i++){`
`   c=a.Substring(i,1);`
`   if (s.IndexOf(c)>=0)`
`      harinder++;`
`}`
		By the time the loop has finished executing, the variable "harinder" stores
		______________
		______________
		______________
		______________

`string c="";`
`string s="Einstein";`
`for (int i=s.Length-1; i>=0; i--)`
`   c=c+s.Substring(i,1);`
		By the time the loop has finished executing, the string "c" stores
		______________
		______________
		______________
		______________
		______________

2. ***On paper***, write C# code to perform each of the following tasks. Do not use a computer for this question except for verifying that your code is correct.

(a) Determine whether a given string is a ***palindrome***. (A ***palindrome*** is a word or phrase that reads the same forward or backward. Examples of palindromes include "bob," "madam" and "ten animals I slam in a net.")

(b) Write a program that counts the number of consonants in a given string.

(c) Write a program that counts the number of "double letter" occurrences in a given string. For instance, in the word "occurrence," there is a double "c" and a double "r," making the count equal to two.

String Assignment – Credit Card Validation

Introduction

All credit card accounts are identified by a number, usually consisting of 14 to 16 digits. To distinguish valid credit card numbers from random sequences of digits, a simple method called the ***Luhn Algorithm*** is used. This algorithm involves computing a value called a ***checksum***, which must be divisible by 10. In this assignment, you will design a program that determines whether a credit card number satisfies the Luhn algorithm and other conditions that are specific to the particular credit card company.

Rules for Credit Card Number Validity

1. *Length and Prefix*

In addition to satisfying the Luhn algorithm, each credit card number must…

- …begin with a specific series of one or more digits (called the ***prefix***)
- …have a specific number of digits (called the ***length***)

For instance, Visa numbers must begin with a "4" and be exactly 16 digits long.

Credit Card Type	*Valid Length*	*Valid Prefix*
Visa	**16**	**4**
Master Card Diners Club (in U.S and Canada)	**16**	**51** to **55**
American Express	**15**	**34** or **37**
Discover	**16**	**6011**
Diners Club (Outside U.S. and Canada)	**14**	**36**

2. *Luhn Algorithm Checksum*

The Luhn algorithm, developed by IBM scientist Hans Peter Luhn in 1954, is described below.

- Begin with a checksum of zero.
- Starting at the ***rightmost*** digit, move from ***right to left*** digit by digit and add to the checksum.
- Digits in odd positions are simply added to the checksum.
- Digits in even positions ("alternate digits") must be adjusted in the following way:
  - Multiply each alternate digit by 2.
  - If the product is more than a single digit (i.e. greater than 9), add the two digits to obtain a single digit.
  - Add the result to the checksum.
- The **checksum mod 10** must be equal to 0. That is, the checksum must be ***exactly*** divisible by 10.

Example

Suppose you are testing the following Visa number: 4568926885633463

1. *Length and Prefix*

Clearly, the number has the correct prefix (4) and the correct length (16).

2. *Luhn Algorithm Checksum*

We shall add the digits from ***right to left***, multiplying and adjusting alternate digits as described above.

1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	11th	12th	13th	14th	15th	16th
3	6	4	3	3	6	5	8	8	6	2	9	8	6	5	4
3	6*2 =12 1+2 =**3**	**4**	3*2 =**6**	**3**	6*2 =12 1+2 =**3**	**5**	8*2 =16 1+6 =**7**	**8**	6*2 =12 1+2 =**3**	**2**	9*2 =18 1+8 =**9**	**8**	6*2 =12 1+2 =**3**	**5**	4*2 =**8**

Therefore, checksum = 3+3+4+6+3+3+5+7+8+3+2+9+8+3+5+8 = 80

Since 80 mod 10 = 0, the checksum is valid.

Therefore, the credit card number is valid, since it meets all conditions.

Program Plan

The following shows you how you can divide the large problem of determining credit card validity into a series of smaller, simpler problems.

Step 1: Capture User Input

- What is the credit card type?
- What is the credit card number?

Step 2: Determine Number Validity

- Check length.
- Check prefix.
- Calculate the checksum using the Luhn algorithm.
- If all conditions are satisfied, the number is valid. Otherwise, it is invalid.

Step 3: Display the Result

- Output a message about credit card validity.

Additional Notes

- Please note that your program will only be able to determine whether a credit card's length, prefix and checksum are valid. It will not be able to determine whether a given credit card number has actually been issued by a bank to one of its customers. Such authentication can only be done through databases that are accessible only to authorized merchants.
- Think carefully of which data types you will use.
- The input for the program is a credit card type and a credit card number. The output should be a determination (true or false) of credit card's numerical validity.
- Think before you start programming! Solve simple problems first! Then move on to more challenging ones.
- You are also expected to provide a set of test cases (sample inputs with corresponding sample outputs) for your program that shows program correctness (i.e. that your program produced correct outputs for all inputs).
- If the user input is anything but a valid number, your program should consider the number invalid.
- If you're stuck, make use of online and offline documentation and resources.

Additional Challenge for Extra Credit

- Instead of asking the user for the credit card type, your program can use the prefix to ***determine*** the type.
- Include a feature that allows the user to ***generate*** valid credit card numbers.

Practice Exercises

1. Determine which of these credit card numbers are valid based on the prefix, length and the Luhn algorithm.

a) Is VISA number 4484663142585415 valid?
Check Prefix: 4 (correct)
Check Length: 16 (correct)
Checksum = 5+2+4+1+8+1+2+8+1+6+6+3+4+7+4+8 = 70
70 mod 10 = 0 (correct)
This number is VALID.

b) Is DISCOVER number 601195145328714 valid?
Check Prefix:
Check Length:
Checksum:
This number is

c) Is MASTERCARD number 5358390378156038 valid?
Check Prefix:
Check Length:
Checksum:
This number is

d) Is AMEX number 375627815798423 valid?
Check Prefix:
Check Length:
Checksum:
This number is

2. Change the invalid numbers above (by changing their digits) into valid ones. How many ways are there to do this?

3. From scratch, come up with your own valid credit card number.

4. Suppose that a 16-digit number is chosen at random. What is the probability that the number would satisfy the Luhn algorithm?

Evaluation Guide for Credit Card Validation Program

Categories	*Criteria*	*Descriptors*					*Level*	*Mark*
		Level 4	Level 3	Level 2	Level 1	Level 0		
Knowledge and Understanding (KU)	**Understanding of Programming Concepts**	Extensive	Good	Moderate	Minimal	Insufficient		
	Understanding of the Problem	Extensive	Good	Moderate	Minimal	Insufficient		
Application (APP)	**Correctness** To what degree is the output correct?	Very High	High	Moderate	Minimal	Insufficient		
	Exception Handling How stable is the software?	Highly Stable	Stable	Moderately Stable	Somewhat Unstable	Very Unstable		
	Declaration of Variables To what degree are the variables declared with appropriate data types?	Very High	High	Moderate	Minimal	Insufficient		
	Unnecessary Duplication of Code To what degree has the student avoided unnecessary duplication of code?	Very High	High	Moderate	Minimal	Insufficient		
	Debugging To what degree has the student employed a logical, thorough and organized debugging method?	Very High	High	Moderate	Minimal	Insufficient		
Thinking, Inquiry and Problem Solving (TIPS)	**Algorithm Design and Selection** To what degree has the student used approaches such as solving a specific example of the problem to gain insight into the problem that needs to be solved?	Very High	High	Moderate	Minimal	Insufficient		
	Ability to Design and Select Algorithms Independently To what degree has the student been able to design and select algorithms without assistance?	Very High	High	Moderate	Minimal	Insufficient		
	Ability to Implement Algorithms Independently To what degree is the student able to implement chosen algorithms without assistance?	Very High	High	Moderate	Minimal	Insufficient		
	Efficiency of Algorithms and Implementation To what degree does the algorithm use resources (memory, processor time, etc) efficiently?	Very High	High	Moderate	Minimal	Insufficient		
Communication (COM)	**Indentation of Code** **Insertion of Blank Lines in Strategic Places** (to make code easier to read)	Very Few or no Errors	A Few Minor Errors	Moderate Number of Errors	Large Number of Errors	Very Large Number of Errors		
	Comments • Effectiveness of explaining abstruse (difficult-to-understand) code • Effectiveness of introducing major blocks of code • Avoidance of comments for self-explanatory code	Very High	High	Moderate	Minimal	Insufficient		
	Descriptiveness of Identifier Names Variables, Constants, Objects, Methods, Data Fields, etc **Clarity of Code** How easy is it to understand, modify and debug the code? **Adherence to Naming Conventions** (e.g. lowerCamelCase for variable names, etc)	Masterful	Good	Adequate	Passable	Insufficient		
	User Interface To what degree is the user interface well designed, logical, attractive and user-friendly?	Very High	High	Moderate	Minimal	Insufficient		

USING ARRAYS IN C#

The Concept of an Array

- An ***array*** is a structure that allows you to use ***a single name*** to refer to a ***group of two or more variables***.
- To distinguish one variable in the group from another, a number, called the ***index*** or ***subscript***, is used.
- This concept is similar to the street address of a house. Each house on a given street is identified by the ***same street name***. However, each house also is identified by a ***unique number***, which makes it possible to locate any given house.
- For example, shown at the right is an overhead view of a portion of Centre Street North in Brampton. Since each house on this street is identified by a unique number, there is never any confusion distinguishing one house from another.
- Arrays are used whenever a program needs to process a group (usually a large group) of related data.
- Arrays help you to create shorter and simpler code in many situations because ***loops*** can be used to process the array elements efficiently, regardless of the size of the array.

Important Details about Arrays in C#

- All the elements in an array have the same data type.
- Because C# must allocate memory for each element of an array, avoid creating very large arrays.
- Arrays have both ***upper*** and ***lower*** bounds and the elements of the array are contiguous within those bounds. In C, C++, C# and a host of other languages derived from C, the ***lowest index is always zero***.
- If a program attempts to access an element of an array using an index that is either negative or greater than the upper bound, an "ArgumentOutOfRangeException" is thrown.
- Arrays can be thought of as ***fixed-size lists***. Once an array has been declared and initialized, the number of elements in the array remains ***fixed***.
- C# also provides support for ***Lists***, which can be thought of as ***variable-size arrays*** or ***dynamic arrays***. Lists are essentially arrays that can grow and shrink in size while a program is being executed. Lists in C# are covered later in this unit.

Several Examples of Array Declarations

Number of elements in the array.

```
//Create a one-dimensional, empty array of "double"
//values. The elements of the array exist but
//they have not yet been assigned any values.

double[] temperature = new double[4];
```

Index	0	1	2	3
Data	–	–	–	–

In this example, a variable of array type is ***declared***, an array object is created and storage space is allocated for the ***elements*** (also called ***components***) of the array. However, the elements of the array do not yet have values.

```
//Create and initialize an array of "double" values.
//Initial values are given in an initializer list.
//An initializer list is a set of values, separated
//by commas and enclosed in braces.

double[] temperature = new double[4] {0,2,4,6};
```

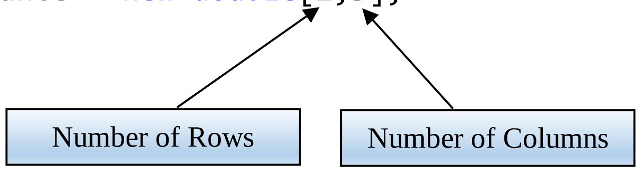
Index	0	1	2	3
Data	0	2	4	6

In C#, arrays are implemented as ***objects***. Therefore, the **`new`** keyword must be used in the declaration of an array to create a new array object. Note that array ***indices*** (singular ***index***, also called ***subscripts***) in C# always start at zero.

```
//Create an array of "string" values. No values
//have been assigned yet to the elements of the array.
string[] name = new string[4];
```

Index	0	1	2	3
Data	–	–	–	–

```
//The following declares a two-dimensional array called
//'distance.' It consists of two rows (horizontal) and
//3 columns (vertical). Its purpose is to store distances
//between points. As with other similar examples, the
//array elements have not yet been assigned values.
double[,] distance = new double[2,3];
```



	0	1	2
0	–	–	–
1	–	–	–

The statements shown at the left can be used to declare and create a ***two-dimensional*** array of `double` values. The row indices run from 0 to 1 and the column indices run from 0 to 2. Without any assignment statements, however, the two-dimensional array is empty (i.e. the elements have not yet been assigned any values).

```
//'distance[i,j]' stores the distance from point 'i' to
//point 'j.' For example, the distance from point 0 to
//point 1 is 10.7.
distance[0,0] = 0;
distance[0,1] = 10.7;
distance[0,2] = 25.3;
distance[1,0] = 10.7;
distance[1,1] = 0;
distance[1,2] = 16.3;
```

	0	1	2
0	0	10.7	25.3
1	10.7	0	16.3

Once the assignment statements at the left are executed, the two-dimensional array (also known as a ***matrix***) will contain the values shown above.

```
//Use an initializer list of initializer lists to initialize the
//two-dimensional array 'distance.'
double[,] distance = new double[2,3] { { 0, 10.7, 25.3 },
                                       { 10.7, 0, 16.3 } };
```

This statement is an alternative (and preferable) method of declaring, creating and initializing the two-dimensional array shown above. Each row of the matrix is enclosed in braces and listed in the desired order.

```
//A two-dimensional array used as a height map for an algorithm
//such as the "diamond-square" algorithm. For the sake of
//simplicity, the array is only 5x5. In reality, it would be
//much larger.
double[,] height = new double[5,5] { { 10, 0, 0, 0, 10 },
                                     { 0, 0, 0, 0, 0 },
                                     { 0, 0, 0, 0, 0 },
                                     { 0, 0, 0, 0, 0 },
                                     { 10, 0, 0, 0, 10 } };
```

We shall study the diamond-square algorithm in detail in the next unit.

Exercises involving Arrays

1. Create a memory map for each code segment. In addition, determine the problem that is solved in each case. (Some variables have intentionally been given silly names to disguise their purpose.)

Code Segment	*Memory Map (Trace Chart)*	*Problem Solved?*

`int[] a = { -1, 5, 3, -6, 3 };`
`int moe = a[0];`
 	
`for (int x = 1; x < a.Length; x++)`
`{`
`    if (a[x] < moe)`
`       moe = a[x];`
`}`
		By the time the loop has finished executing, the variable “moe” stores
		________________
		________________
		________________
		________________
		________________

`Random randomGenerator = new Random();`
`int[] a = new int[6];`
 	
`for (int i = 0; i < a.Length; i++)`
`{`
`    bool rep = false;`
`    int r;`
 	
`    do`
`    {`
`       r = randomGenerator.Next(1, 70);`
`       rep = false;`
 	
`       for (int j = 0; j < i; j++)`
`       {`
`          if (a[j] == r)`
`          {`
`             rep = true;`
`             break; //exit 'for' loop`
`          }`
`       }//end inner for`
 	
`    } while (rep);`
 	
`    a[i] = r;`
 	
`}//end outer for`
		By the time the outer for loop has finished executing, the array “a” stores
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________
		________________

2. The following table lists answers to question 1. Check your answers to ensure that they are correct.

Code Segment	***Memory Map (Trace Chart)***	***Problem Solved?***
see code 1 below	see trace chart 1 below	By the time the loop has finished executing, the variable "moe" stores **the smallest value stored in the array.**
see code 2 below	see trace chart 2 below	By the time the outer for loop has finished executing, the array "a" stores **six random integers ranging from 1 to 69, without repetition (i.e. each random integer is different from all the others).**

```
int[] a = { -1, 5, 3, -6, 3 };
int moe = a[0];

for (int x = 1; x < a.Length; x++)
{
    if (a[x] < moe)
       moe = a[x];
}
```

Data stored in the array "a."

Index	0	1	2	3	4
Data	–1	5	3	–6	3

x	moe
-	-1
1	-1
2	-1
3	-6
4	-6
-	-6

```
Random randomGenerator = new Random();
int[] a = new int[6];

for (int i = 0; i < a.Length; i++)
{
    bool rep = false;
    int r;

    do
    {
       r = randomGenerator.Next(1, 70);
       rep = false;

       for (int j = 0; j < i; j++)
       {
          if (a[j] == r)
          {
             rep = true;
             break; //exit 'for' loop
          }
       }//end inner for

    } while (rep);

    a[i] = r;

}//end outer for
```

Since the given code produces ***random*** integers, it is not possible to predict exactly what will occur when the code is executed. The following is an example of what ***could happen***.

Array Index / i	0	1	2	3	4	5	r
-	-	-	-	-	-	-	-
0	27	-	-	-	-	-	27
1	27	3	-	-	-	-	3
2	27	3	51	-	-	-	51
3	27	3	51	-	-	-	**3**
3	27	3	51	-	-	-	16
4	27	3	51	16	-	-	**27**
4	27	3	51	16	-	-	**51**
4	27	3	51	16	42		42
5	27	3	51	16	42	9	9

Notice the numbers displayed in **red**. Since each of these numbers already occurred for a previous value of "i," a new value of "r" needs to be generated.

3. ***On paper***, write C# static methods to perform each of the following tasks. Do not use a computer for this question except for verifying that your code is correct.

(d) Find the ***largest value*** stored in an array.

(e) Find the ***average*** of the values stored in an array.

(f) Find the ***median*** of the values stored in an array.

(g) Copies the values stored in an array to another array. (Avoid this in practice because it uses a great deal of memory.)

(h) Fill an array of 52 elements with random integers ranging from 0 to 51 without repetition. (This is equivalent to shuffling a deck of 52 cards. Use a diagram to illustrate this.) See question 1 for a hint.

4. Write a C# program for a word "jumble" game (also known as word scramble). The user is given a word in "jumbled" form (the letters are randomly rearranged) and the user is given a limited number of guesses and/or a time limit to figure out the word. For example, if the user is given the string "bmejul," the correct answer would be "jumble."

Lists in C#

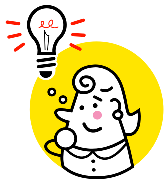
Introduction

ICS4U0 – Roman Converter Project

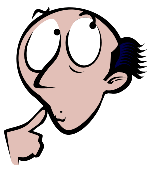
Roman to Hindu-Arabic Converter

Write a program that can convert a number expressed in Roman form to a number expressed in Hindu-Arabic form and vice versa. Your program must

- be able to convert any value from 1 to 3999999 from Hindu-Arabic to Roman or vice versa
- respond ***intelligently*** to ***any*** user input
- conform to the usual conventions of good coding



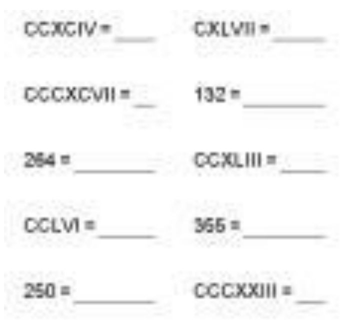
Before setting out to write Code, Consider this…



1. What are the rules for writing numbers using Roman numerals?
2. How can you design an *algorithm* that converts from Hindu-Arabic to Roman?
3. How can you design an *algorithm* that converts from Roman to Hindu-Arabic?
4. How are numbers greater than 3999 represented using Roman numerals?
5. How can you make your program recognize invalid values such as "XXMMMM?"

The **look**
on Mr. Nolfi's face
whenever…

1. …students install software or change computer settings without asking for permission!
2. …students try to write programs to solve problems that they do not know how to solve!



CCXCIV = ____	CXLVII = ______
CCCXCVII = __	132 = ________
264 = ________	CCXLIII = _____
CCLVI = ______	365 = ________
250 = ________	CCCXXIII = ___

STOP! DO NOT WRITE ANY CODE YET! First we need to TRY SPECIFIC EXAMPLES and develop A PLAN!

The table below shows the basic “building blocks” of Roman numbers less than 4000 and their respective values. That is, any Hindu-Arabic number less than 4000 can be written as a Roman number that uses some combination of the symbols listed below. The best way to store the Roman symbols and their values is to use two arrays. (Keep in mind that in C, C++ and C#, ***array indices always begin at zero***. This is not the case in VB, where indices can range from any **Integer** value to any other **Integer** value.)

Index (Subscript) / **Array Name**	0	1	2	3	4	5	6	7	8	9	10	11	12
romanSymbol	`"M"`	`"CM"`	`"D"`	`"CD"`	`"C"`	`"XC"`	`"L"`	`"XL"`	`"X"`	`"IX"`	`"V"`	`"IV"`	`"I"`
romanSymbolValue	1000	900	500	400	100	90	50	40	10	9	5	4	1

Hindu-Arabic to Roman Algorithm Example

Convert 1642 to Roman form.

Operation	*Remainder*	*Quotient*	*Roman String*
	1642	-	“”
÷1000	642	1	“M”
÷900	642	0	“M”
÷500	142	1	“MD”
÷400	142	0	“MD”
÷100	42	1	“MDC”
÷90	42	0	“MDC”
÷50	42	0	“MDC”
÷40	2	1	“MDCXL”
÷10	2	0	“MDCXL”
÷9	2	0	“MDCXL”
÷5	2	0	“MDCXL”
÷4	2	0	“MDCXL”
÷1	0	2	“MDCXLII”

Roman to Hindu-Arabic Algorithm Example

Convert “MCMXLIV” to Hindu-Arabic form.

i	*Character at Index i*	*Character at Index i+1*	*Operation*	*Hindu-Arabic Form*
-	-	-	-	0
0	“M”	“C”	+1000	1000
1	“C”	“M”	–100	900
2	“M”	X	+1000	1900
3	“X”	“L”	–10	1890
4	“L”	“I”	+50	1940
5	“I”	“V”	–1	1939
6	“V”	-	+5	1944

Hindu-Arabic to Roman Algorithm Pseudo-Code

```
store all possible one character and two
   character Roman symbol combinations in
   descending order in an array
store Hindu-Arabic values of above in descending
   order in another array
set roman to null string
set remainder to value of Hindu-Arabic number
for (i=0; i<number elements of array; i++)
{
  set quotient to quotient of remainder divided
    by element "i" of the array storing divisors
  set remainder to remainder of remainder
    divided by element "i" of the same array
  concatenate quotient Roman symbols (of type
    found at element "i" of Roman symbol array)
    to roman
}
```

Roman to Hindu-Arabic Algorithm Pseudo-Code

```
set len to length of the Roman number string
for (i=0; i<len; i++)
{
  set char to character at position "i"
  set value to Hindu-Arabic value of char
  if (i<len-1)
  {
    set nextChar to character at position "i+1"
    set valueNext to Hindu-Arabic value of nextChar
  }
  if (valueNext<=value)
    set HinduArabic to HinduArabic + value
  else
    set HinduArabic to HinduArabic - value
}
```

Exercises

Convert 2007 to Roman form.

Operation	***Remainder***	***Quotient***	***Roman String***

Convert "MCMXCVIII" to Hindu-Arabic form.

i	***Character at Index i***	***Character at Index i+1***	***Operation***	***Hindu-Arabic Form***

Convert 3999 to Roman form.

Operation	***Remainder***	***Quotient***	***Roman String***

Convert MMMCDXLIV" to Hindu-Arabic form.

i	***Character at Index i***	***Character at Index i+1***	***Operation***	***Hindu-Arabic Form***

Roman Converter Evaluation Guide

Victim: ______________________________

Categories	***Criteria***	***Descriptors*** Level 4	Level 3	Level 2	Level 1	Level 0	***Mark***
Knowledge and Understanding (KU)	**Degree of Completeness** ☐ be able to convert any value from 1 to 3999999 from Hindu-Arabic to Roman or vice versa	Very High (All features imple-mented)	High (Most features imple-mented)	Moderate (Some important features imple-mented)	Minimal (A few features imple-mented)	Insufficient (Little to nothing imple-mented)	— 20
Application (APP)	**Correctness** To what degree does the program produce correct output?	Very High	High	Moderate	Minimal	Insufficient	— 20
	Avoidance of Code Duplication To what degree has the student used methods (i.e functions) to avoid duplication of code? (i.e. to avoid copy & paste coding)	Very High	High	Moderate	Minimal	Insufficient	
	Data Validation and Exception Handling To what degree are exceptions caught and handled? To what degree can the program detect invalid input?	Very High	High	Moderate	Minimal	Insufficient	
Thinking, Inquiry and Problem Solving (TIPS)	**Independence** To what degree has the student been able to implement the solution ***without*** asking for assistance?	Very High	High	Moderate	Minimal	Insufficient	— 30
	Research When problems are encountered during the design, implementation and validation phases, to what degree has the student consulted resources ***before*** asking for help?	Very High	High	Moderate	Minimal	Insufficient	
	Algorithm/Implementation Efficiency ☐ To what level does the algorithm use resources (memory, processor time, etc) efficiently? ☐ To what degree are appropriate data types used?	Very High	High	Moderate	Minimal	Insufficient	
Communication (COM)	**Indentation of Code** **Insertion of Blank Lines in Strategic Places** (to make code easier to read)	Very Few or no Errors	A Few Minor Errors	Moderate Number of Errors	Large Number of Errors	Very Large Number of Errors	— 30
	Comments (Internal Documentation) ☐ Effectiveness of explaining abstruse (difficult-to-understand) code ☐ Effectiveness of introducing major blocks of code ☐ Avoidance of comments for self-explanatory code	Very High	High	Moderate	Minimal	Insufficient	
	Descriptiveness of Identifier Names Variables, Constants, Objects, Methods, Classes, etc **Method and Class Design** ☐ Methods are self-contained (can be used in other programs without modification) ☐ Parameters and return types are logical ☐ Class structure is logical and efficient **Clarity of Code** How easy is it to understand, modify and debug code? **Adherence to Naming Conventions** ☐ lowerCamelCase used for variable, object, methods ☐ UpperCamelCase used for classes and constructors ☐ ALL_UPPER_CASE used for constants	Masterful	Good	Adequate	Passable	Insufficient	

Inquiry: Using a Challenging Problem to Expand our Knowledge

Problem

Given an integer *n*, find an integer *m* (if one exists) such that ***all*** the following conditions hold:

- $m > n$
- *m* contains exactly the same digits as *n*
- There is no other integer *q* containing exactly the same digits as *n* such that $n < q < m$.

Less formally, find the ***smallest integer m*** that contains the ***same digits*** as *n* but is also greater than *n*.

Specific Examples

Input (n)	***Output (m)***
9	Does not exist
11	Does not exist
32	Does not exist
23	32
323	332
676	766
521	Does not exist
777	Does not exist
1000	Does not exist
1001	1010
1321	2113
4981	8149
54981	58149
1472443	1473244
1472483	1474823
1474823	1474832
98765432	Does not exist

Your Task

1. Develop a "brute-force" algorithm to solve this problem. Such algorithms are also called ***exhaustive searches*** because they solve problems by blindly searching through the set of all possible solutions.
2. Develop a cleverer algorithm for solving this problem. Compare the efficiency of this algorithm to that of the brute-force algorithm.
3. Implement both algorithms in C#.

Technical Aspects of C#

Strongly-Typed Languages (Type Safety)

Contrary to what our friend Georgie W. Bush might believe, the meaning of "strongly-typed" has nothing at all to do with typing! This term actually is used to describe certain programming languages that have strict rules governing how data types are used. There is no universally agreed-upon definition of what it means to be strongly-typed. Nonetheless, certain important data type rules in C# can help us get the gist of what it means.

Hmm, let me think about that one. Oh yes, a strongly-typed language means that the keys on a keyboard must be pressed very hard. Does anyone know if Mexican is strongly-typed?

Strongly-Typed Features of C#

- All variables ***must have*** a defined data type.
- Implicit conversion of data types is generally ***not allowed***.
- Strict type checking is performed at run-time.

Examples of how Strong Typing is enforced in C#

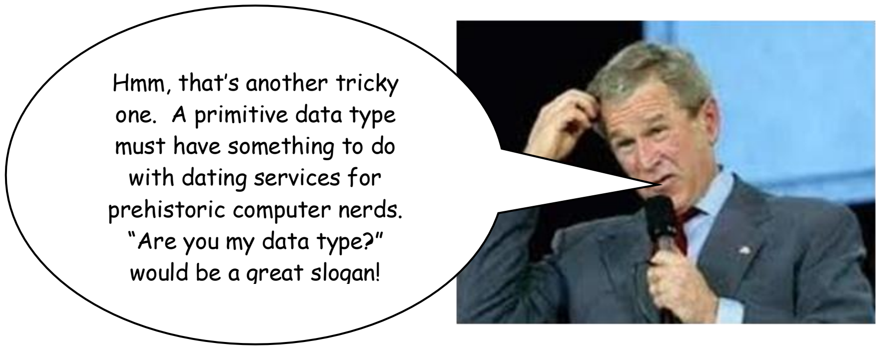
```
int x=3✓    // The variable 'x' cannot be used unless it is first declared.

int x="3";✗ // NOT ALLOWED in C#! VB would perform an implicit conversion from type
            // 'String' to type 'Integer.' C# does not perform such conversions!

int x=Convert.toInt32("3"); ✓ // An explicit data type conversion is performed.
                              // The string "3" is explicitly converted to
                              // a 32-bit signed integer (i.e. the 'int' type).

int x=3;
long y=3;

y=x; ✓      // This is allowed. An implicit conversion is made from 'int' to 'long.'
            // It is safe to perform an implicit conversion in this case because a
            // 'long' is a 64-bit integral type, which means it has plenty of room to
            // accept a 32-bit 'int' integral value.

x=y; ✗      // This is NOT allowed. The type 'long' is a 64-bit integer while the type
            // 'int' is only a 32-bit integer. An implicit conversion would result
            // in a loss of 32 bits of data!

x=(int)y;✓  // The 'cast' operator is formed by enclosing a data type in parentheses.
            // This operator forces a conversion to the specified type even if the
            // conversion results in a loss of data. This is called TYPE COERCION.
```

Type Safety

Because there is a great deal of confusion regarding exactly what it means, some writers avoid the term "strongly-typed" in favour of the term "type safety."

Primitive Data Types

Primitive data types are basic data types that are provided by programming languages. As with the term "strongly-typed," there is no consensus on precisely what constitutes a primitive data type. Once again, a general description is given for the sake of getting across an intuitive notion of what is meant by "primitive data type."


Hmm, that's another tricky one. A primitive data type must have something to do with dating services for prehistoric computer nerds. "Are you my data type?" would be a great slogan!

Primitive Data Type

In general, ***primitive data types*** are the "simplest" data types provided by a programming language. Admittedly, this description is rather vague because it is not clear exactly what is meant by "simplest." The following terms, which are closely related to the idea of a primitive data type, may help to clarify matters.

- ***Basic Type***
  A ***basic data type*** is a type provided by a programming language that serves as a building block for creating more complicated types called ***composite types***. In general, basic types cannot be decomposed (i.e. broken down) into simpler components. In a sense, basic data types are like the elements of the periodic table. All known substances are either elements or made up of some combination of elements.
- ***Built-In Type***
  A ***built-in data type*** is a data type for which a programming language has built-in support.

Comparison of Primitive Data Types in Java and C#

Java

- Java has a small number of primitive data types: **byte, short, int, long, float, double, char, boolean**. All other types are built from these basic types.
- Primitive data types ***are not*** implemented as objects in Java. This means that they have no further substructure. They are basic entities that cannot be decomposed into simpler types.
- Java provides built-in support for primitive data types through ***wrapper classes***. For example, the wrapper class for the "**int**" type is called "Integer." The "Integer" class contains methods for working with "**int**" values.
- Therefore, primitive data types in Java are both ***basic*** and ***built-in***.

Example: Create a String Object in Java

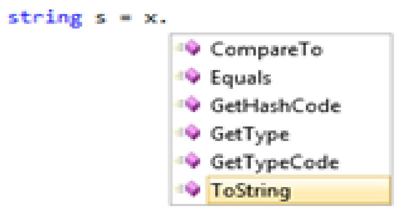
```
int x=3;
String s = new String(Integer.toString(x));
```

C#

- C# has a larger number of primitive data types: **byte, sbyte, short, ushort, int, uint, long, ulong, float, double, char, bool, object, string, decimal**
- Primitive data types ***are*** implemented as objects in C#. This means that they do have a further substructure, making it possible to call object methods on primitive data types. Like molecules made from the elements of the periodic table, they are composite entities.
- Since primitive data types are implemented as objects in C#, there is no need for wrapper classes within C# to provide support for them. The wrapper classes for the C# primitive data types are contained within the .NET framework.
- Therefore, primitive data types in C# are ***built-in*** but not basic. All primitive data types in C# are composite types.

Example: Create a String Variable in C#

```
int x=3;
string s = x.ToString();
```



```
string s = x.
```

Primitive Data Types in C#

The following table lists the primitive data types in C# along with their corresponding .NET framework classes.

Note that the symbol "e" is used to denote powers of 10 for numbers expressed in scientific notation.

e.g. $-3.402823e38 = -3.402823\times10^{38} = -340282300000000000000000000000000000000$

Short Name	***.NET Class***	***Type***	***Width (bits)***	***Range***
byte	Byte	Unsigned Integer	8	0 to 255 (0 to 2^8–1)
sbyte	SByte	Signed Integer	8	−128 to 127 (-2^7 to 2^7–1)
int	Int32	Signed Integer	32	−2147483648 to 2147483647 (-2^{31} to 2^{31}–1)
uint	UInt32	Unsigned Integer	32	0 to 4294967295 (0 to 2^{32}–1)
short	Int16	Signed Integer	16	−32,768 to 32,767 (-2^{15} to 2^{15}–1)
ushort	UInt16	Unsigned Integer	16	0 to 65535 (0 to 2^{16}–1)
long	Int64	Signed Integer	64	−9223372036854775808 to 9223372036854775807 (-2^{63} to 2^{63}–1)
ulong	UInt64	Unsigned Integer	64	0 to 18446744073709551615 (0 to 2^{64}–1)
float	Single	Single-Precision Floating-Point Type (7 Significant Digits)	32	−3.402823e38 to 3.402823e38
double	Double	Double- Precision Floating-Point Type (15 to 16 Significant Digits)	64	−1.79769313486232e308 to 1.79769313486232e308
char	Char	A Single Unicode Character	16	Unicode Symbols used in Text
bool	Boolean	Logical Boolean Type	8	true or false
object	Object	Base Type of all other Types		
string	String	A Sequence of Unicode Characters		
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits. Compared to floating-point types, the decimal type has a greater precision and a smaller range, which makes it suitable for financial and monetary calculations.	128	Approximate Range $\pm1.0 \times 10^{-28}$ to $\pm7.9 \times 10^{28}$

Exercises

Study the given C# declarations. Then complete the table.

```
int a;
long b=3;
short c=1;
byte d=1;
char e=(char)1024; // Force a conversion from 'int' to 'char.' The decimal value 1024 can
                   // also be specified explicitly as the hexadecimal Unicode value
                   // '\u0400' or as the hexadecimal escape sequence '\x0400'
float f=3.8e21f; // The 'f' at the end means that the value should be stored as a 'float'
double g=3.232552e152; // Unless specified otherwise, floating point values are stored as
                       // 'double' values. A 'd' can also be used to indicate that the
                       // number should be stored as a 'double' value: e.g. 2.0d
string h="If thinking makes your brain hurt it’s probably due to lack of practice!";
```

C# Statement	*Is it Allowed? If so, explain why. If not, suggest a correction.*
1. `b=a;`	
2. `a=b;`	
3. `h=f;`	
4. `f=h;`	
5. `e=c;`	
6. `c=e;`	
7. `g=f;`	
8. `f=g;`	
9. `f=d;`	
10. `d=f;`	
11. `d=e;`	
12. `e=d;`	
13. `f=b;`	

WHAT THE HECK IS THE .NET FRAMEWORK?

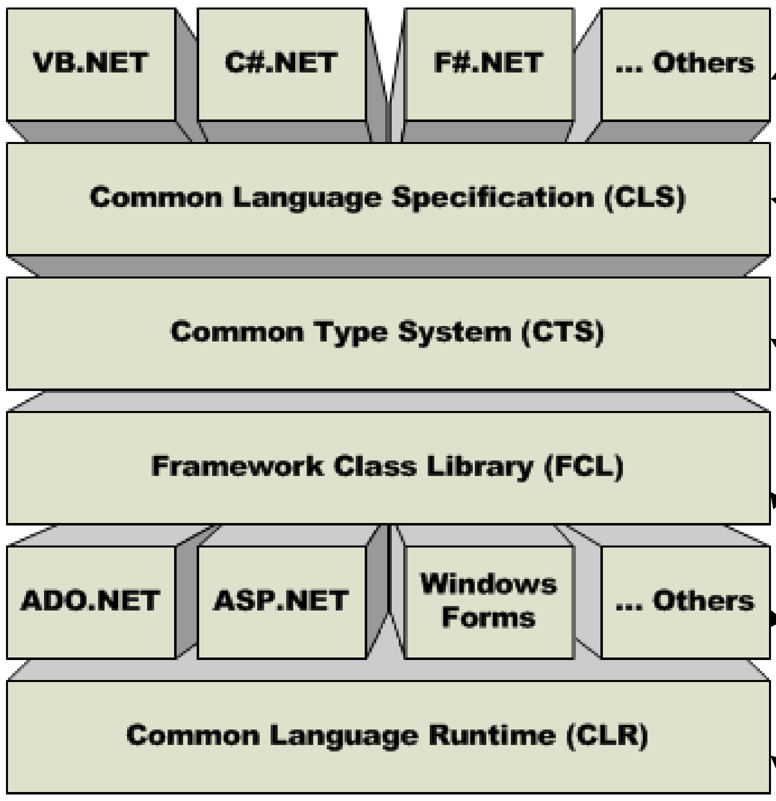
Overview of the .NET Framework

Source: http://en.wikipedia.org/wiki/.NET_Framework

The .NET Framework (pronounced *dot net*) is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large library and provides language interoperability (each language can use code written in other languages) across several programming languages. Programs written for the .NET Framework execute in a software environment (as contrasted to hardware environment), known as the Common Language Runtime (CLR), an application virtual machine that provides services such as security, memory management and exception handling. The class library and the CLR together constitute the .NET Framework.

The .NET Framework's Base Class Library provides user interface, data access, database connectivity, cryptography, Web application development, numeric algorithms and network communications. Programmers produce software by combining their own source code with the .NET Framework and other libraries. The .NET Framework is intended to be used by most new applications created for the Windows platform. Microsoft also produces an integrated development environment largely for .NET software called Visual Studio.

Visual Overview of the .NET Framework


VB.NET
C#.NET
F#.NET
... Others
Common Language Specification (CLS)
Common Type System (CTS)
Framework Class Library (FCL)
ADO.NET
ASP.NET
Windows Forms
... Others
Common Language Runtime (CLR)

Languages supported by the .NET framework include .NET versions of C++, C#, F#, VB and many others. (See List of CLI languages.)

A CLS is a set of rules for translating source code into bytecode (p-code). When languages use the same CLS, different parts of a program can be written in different languages.

The .NET CTS is a standard that specifies how data types are represented in memory. This standard allows programs written in different languages to share data easily.

The FCL is a collection of reusable classes, interfaces and value types that provide basic functionality for applications.

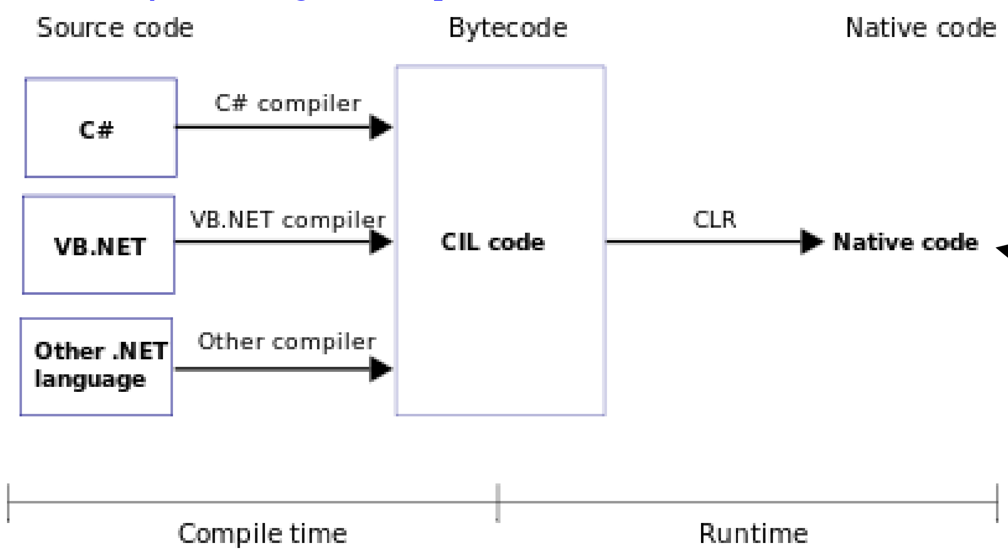
ADO.NET: Framework for data access
ASP.NET: Framework for server-side Web apps
Windows Forms: Framework for Windows GUI

The CLR is the virtual machine component of the .NET framework. It is responsible for managing the execution of .NET programs. All programs written for the .NET framework, regardless of programming language, are executed by the CLR. It provides exception handling, garbage collection and thread management. The CLR is common to all versions of the .NET framework.

See Also

Common Language Infrastructure (CLI)

Overview of .NET Program Compilation and Execution

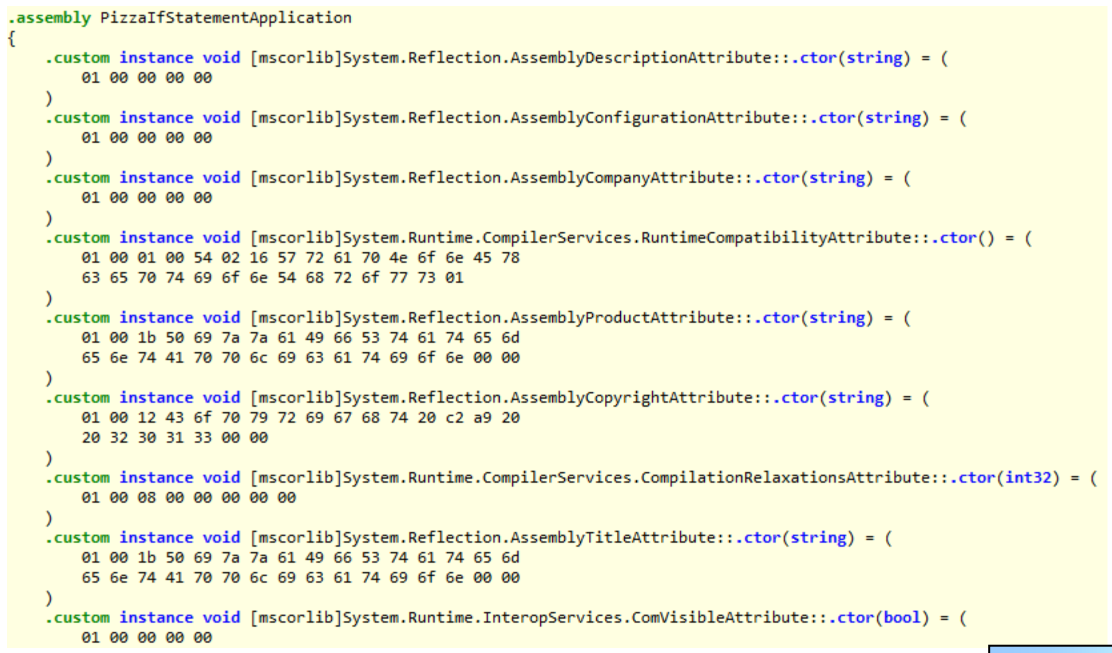


- Outside of Microsoft circles, native code is more commonly known as machine code.
- The "human-readable" form of machine code is called assembly code.
- A central processing unit, also known as a ***CPU*** or ***processor***, cannot execute instructions written in high-level languages like VB and C#.
- CPUs can only execute machine language instructions.

What .NET CIL Code (aka Bytecode or p-code) Looks Like

CIL: Common Intermediate Language (see http://en.wikipedia.org/wiki/Common_Intermediate_Language)

The following is a sample of what .NET CIL code looks like. For the sake of restricting this code to a single page, a small portion of the code has been omitted.

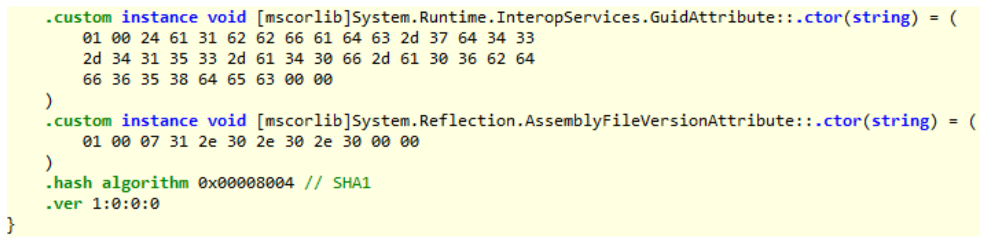


```
.assembly PizzaIfStatementApplication
{
    .custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = (
        01 00 00 00 00
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) = (
        01 00 00 00 00
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = (
        01 00 00 00 00
    )
    .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = (
        01 00 01 00 54 02 16 57 72 61 70 4e 6f 6e 45 78
        63 65 70 74 69 6f 6e 54 68 72 6f 77 73 01
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = (
        01 00 1b 50 69 7a 7a 61 49 66 53 74 61 74 65 6d
        65 6e 74 41 70 70 6c 69 63 61 74 69 6f 6e 00 00
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = (
        01 00 12 43 6f 70 79 72 69 67 68 74 20 c2 a9 20
        20 32 30 31 33 00 00
    )
    .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = (
        01 00 08 00 00 00 00 00
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = (
        01 00 1b 50 69 7a 7a 61 49 66 53 74 61 74 65 6d
        65 6e 74 41 70 70 6c 69 63 61 74 69 6f 6e 00 00
    )
    .custom instance void [mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) = (
        01 00 00 00 00
```

.

.

.



```
    .custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string) = (
        01 00 24 61 31 62 62 66 61 64 63 2d 37 64 34 33
        2d 34 31 35 33 2d 61 34 30 66 2d 61 30 36 62 64
        66 36 35 38 64 65 63 00 00
    )
    .custom instance void [mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = (
        01 00 07 31 2e 30 2e 30 2e 30 00 00
    )
    .hash algorithm 0x00008004 // SHA1
    .ver 1:0:0:0
}
```

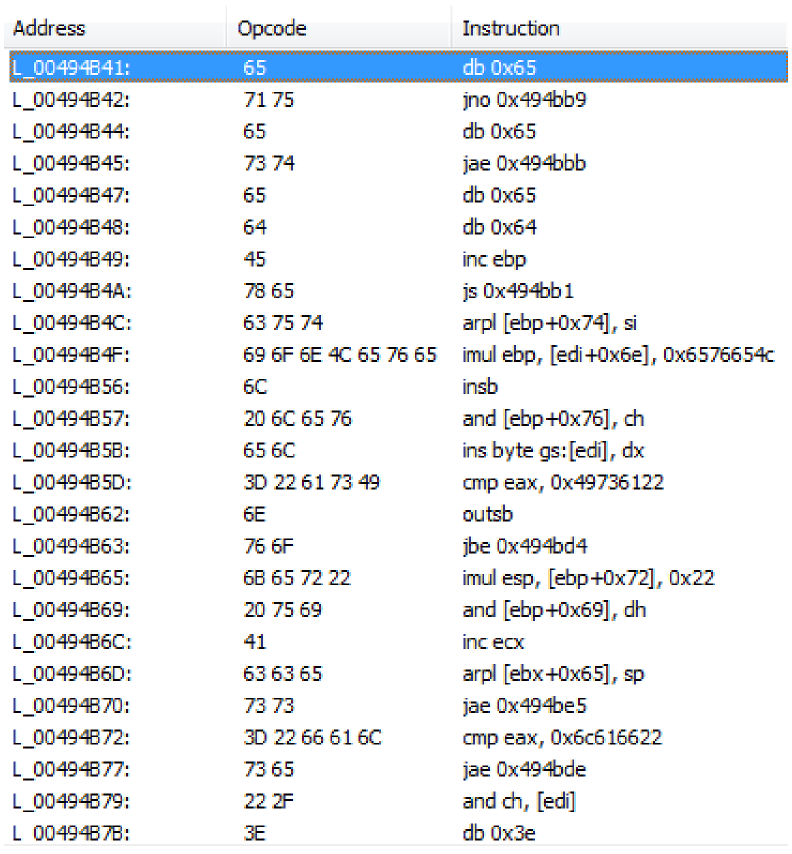
The CIL code shown at the left was produced by opening "PizzaIfStatementApplication.exe" using a free program called ILSpy. (ILSpy stands for "Intermediate Language Spy.")

"PizzaIfStatementApplication.exe" is one of the executable files associated with the C# example found on page 18 of these notes.

What Assembly Code Looks Like

Assembly code is essentially the "human-readable" form of machine code. While machine code is purely numeric, assembly code contains short abbreviations that represent machine instructions. For example, the abbreviation "**jae**" found in the sample assembly code below stands for "**jump if above or equal**." Using abbreviations instead of numbers makes it possible for specialists in assembly code to understand, modify and create machine instructions. Nowadays, it is not often necessary to work directly with assembly code because programs are much easier to understand, modify and create using high-level languages like C# and VB. Occasionally, however, it is necessary to work with assembly code, especially in applications for which speed is critical.

Assembler:	A program that translates assembly code into machine code.
Disassembler:	A program that translates machine code into assembly code.
Compiler:	A program that translates source code written in a high-level language such as C# or VB to a lower-level language such as CIL, assembly language or machine language.



Address	Opcode	Instruction
L_00494B41:	65	db 0x65
L_00494B42:	71 75	jno 0x494bb9
L_00494B44:	65	db 0x65
L_00494B45:	73 74	jae 0x494bbb
L_00494B47:	65	db 0x65
L_00494B48:	64	db 0x64
L_00494B49:	45	inc ebp
L_00494B4A:	78 65	js 0x494bb1
L_00494B4C:	63 75 74	arpl [ebp+0x74], si
L_00494B4F:	69 6F 6E 4C 65 76 65	imul ebp, [edi+0x6e], 0x6576654c
L_00494B56:	6C	insb
L_00494B57:	20 6C 65 76	and [ebp+0x76], ch
L_00494B5B:	65 6C	ins byte gs:[edi], dx
L_00494B5D:	3D 22 61 73 49	cmp eax, 0x49736122
L_00494B62:	6E	outsb
L_00494B63:	76 6F	jbe 0x494bd4
L_00494B65:	6B 65 72 22	imul esp, [ebp+0x72], 0x22
L_00494B69:	20 75 69	and [ebp+0x69], dh
L_00494B6C:	41	inc ecx
L_00494B6D:	63 63 65	arpl [ebx+0x65], sp
L_00494B70:	73 73	jae 0x494be5
L_00494B72:	3D 22 66 61 6C	cmp eax, 0x6c616622
L_00494B77:	73 65	jae 0x494bde
L_00494B79:	22 2F	and ch, [edi]
L_00494B7B:	3E	db 0x3e

The assembly code shown at the left was produced by disassembling "PizzaIfStatementApplication.exe" using a free program called Explorer Suite.

Only a small portion of the assembly code is shown.