

PROGRAMMING C# IN VISUAL STUDIO

Week 9, Oct 22 2009

Parametric Modeling with BIM

C# OVERVIEW

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Diagnostics;
```

```
namespace Hello_World
```

```
{
```

```
class Program
```

```
{
```

```
static void Main(string[] args)
```

```
{
```

```
// Comment for the Hello World!
```

```
Console.WriteLine("Hello World!");
```

```
Console.ReadLine();
```

```
}
```

```
}
```

```
}
```

Keywords

namespace

class

Function Main()

Comment

namespace

class1

class2

.....

class

function

function

.....

C#: KEYWORDS

- Keywords are the character string tokens used to define the C# language.
 - >> Keywords **cannot** be used as variable names or any other form of identifier, unless prefaced with the **@** character.
 - >> All C# keywords consist entirely of lowercase letters.

C#: KEYWORDS LIST

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

C#: DOT OPERATOR(.)

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;

namespace Hello_World
{
    class Program
    {
        static void Main(string[] args)
        {
            // Comment for the Hello World!
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

DOT OPERATOR(.)



C#: COMMENTS

Annotating the Code:

> Single-line comment

The text from the beginning marker to the end of the current line is ignored by the compiler.

> Delimited comment

The text between the start and end markers is ignored by the compiler.

> Documentation comment

Comments of this type contain XML text that is meant to be used by a tool to produce program documentation.

C#: COMMENTS

Single-line comment

>> The beginning marker: `//`

example:

```
// Single-line comment
```

Delimited comment

>> The start marker: `/*`

>> The end marker: `*/`

example:

```
/*  
This text is ignored by the compiler.  
Unlike single-line comments, delimited comments like this  
one can span several lines.  
*/
```

C#: COMMENTS

Documentation comment

>> The beginning marker: *///*

Comments of this type contain XML text that is meant to be used by a tool to produce program documentation.

example:

```
/// <summary>  
/// This class does...  
/// </summary>  
class Program  
{  
...  
}
```


C#: COMMENTS

Documentation comment

Tags:

<c>
<para>
<see>*
<code>
<param>*
<seealso>*
<example>
<paramref>
<summary>

<exception>*
<permission>*
<typeparam>*
<include>*
<remarks>
<typeparamref>
<list>
<returns>
<value>

Example:

```
/// <summary>  
/// Enter description for method bb.  
/// </summary>  
/// <param name="s">Describe parameter.</param>  
/// <param name="y">Describe parameter.</param>  
/// <param name="z">Describe parameter.</param>  
/// <returns>Describe return value.</returns>  
public int bb(string s, ref int y, void* z)  
{  
    .....  
    return 1;  
}
```

C#:TYPES

A Type is a Template.

Some types, such as short, int, and long, are called simple types, and can only store a **single** data item. Other types can store **multiple** data items. An array, for example, is a type that can store multiple items of the same type.

A type is defined by the following elements:

- A name
- A data structure to contain its **data members**
- Behaviors and constraints (**function members**)

C#: TYPES

Predefined Types

C# provides 15 predefined types and there include 13 simple types and 2 non-simple types. The names of all the predefined types consist of all lowercase characters. The predefined simple types include the following:

Eleven numeric types, including

- Various lengths of **signed** and **unsigned** integer types.
- Floating point types—**float** and **double**.
- A high-precision decimal type called **decimal**. Unlike float and double, type decimal can represent decimal fractional numbers exactly. It is often used for monetary calculations.

A Unicode character type, called **char**.

A Boolean type, called **bool**. Type bool represents Boolean values and must be one of two values—either **true** or **false**.

C#:TYPES

Predefined Types

Various lengths of **signed** and **unsigned** integer types.

Type	Range	Size
sbyte (SByte)	-128 to 127	Signed 8-bit integer
byte (Byte)	0 to 255	Unsigned 8-bit integer
short (Int16)	-32,768 to 32,767	Signed 16-bit integer
ushort (UInt16)	0 to 65,535	Unsigned 16-bit integer
int (Int32)	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint (UInt32)	0 to 4,294,967,295	Unsigned 32-bit integer
long (Int64)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong (UInt64)	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

If the value represented by an integer literal exceeds the range of ulong, a compilation error will occur.

C#:TYPES

User-Defined Types

Besides the 15 predefined types provide by C#, you can also create your own user-defined types. There are **six** kinds of types you can create:

- **class** types
- **struct** types
- **array** types
- **enum** types
- **delegate** types
- **interface** types

Types are created using a type declaration, which includes the following information:

- The **kind of type** you are creating
- The **name** of the new type
- A declaration (name and specification) of each of the type's **members**—except for array and delegate types, which do not have named members.

C#: TYPES

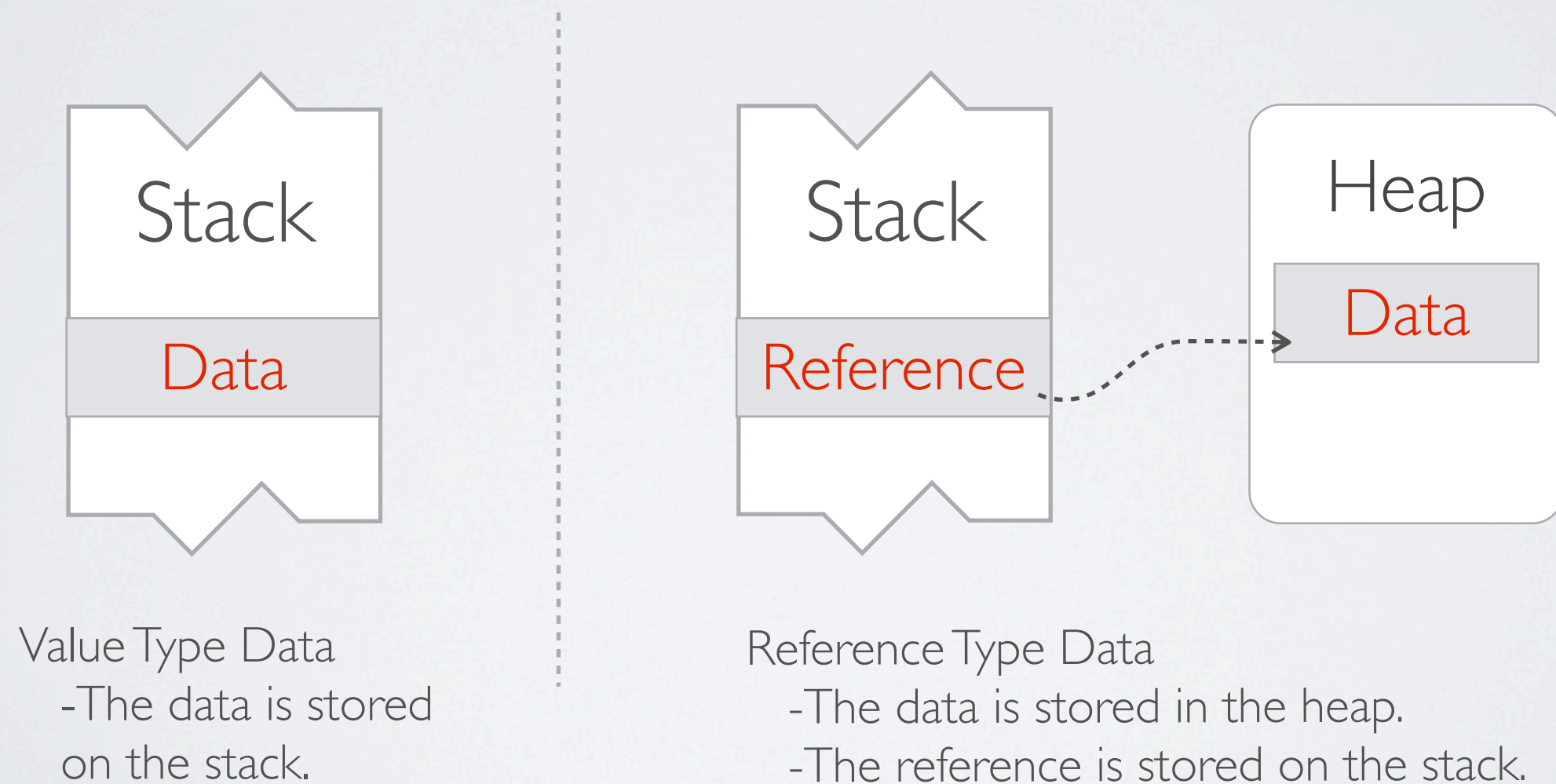
Value Types and Reference Types

The **type** of a data item defines **how much memory** is required to store it, the data members that comprise it, and the functions that it is able to execute. The **type** also determines **where** an object is stored in memory—the **stack** or the **heap**. Types are divided into two categories: **value** types and **reference** types. Objects of these types are stored differently in memory.

- **Value** types require only a single segment of memory—which stores the actual data.
- **Reference** types require two segments of memory:
 - The first contains the actual **data**—and is always located in the **heap**.
 - The second is a **reference** that points to where in the heap the data is stored.

C#:TYPES

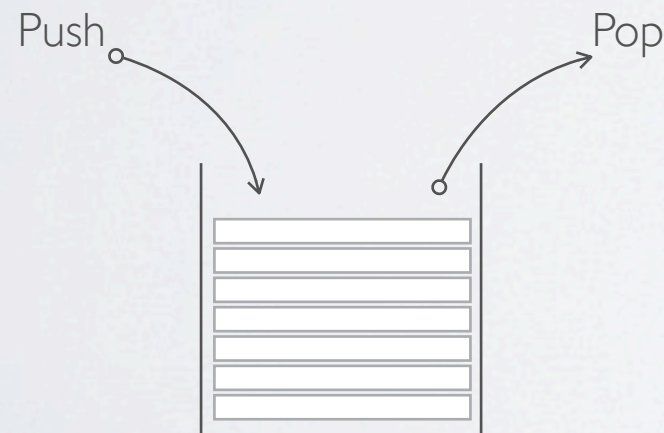
Value Types and Reference Types



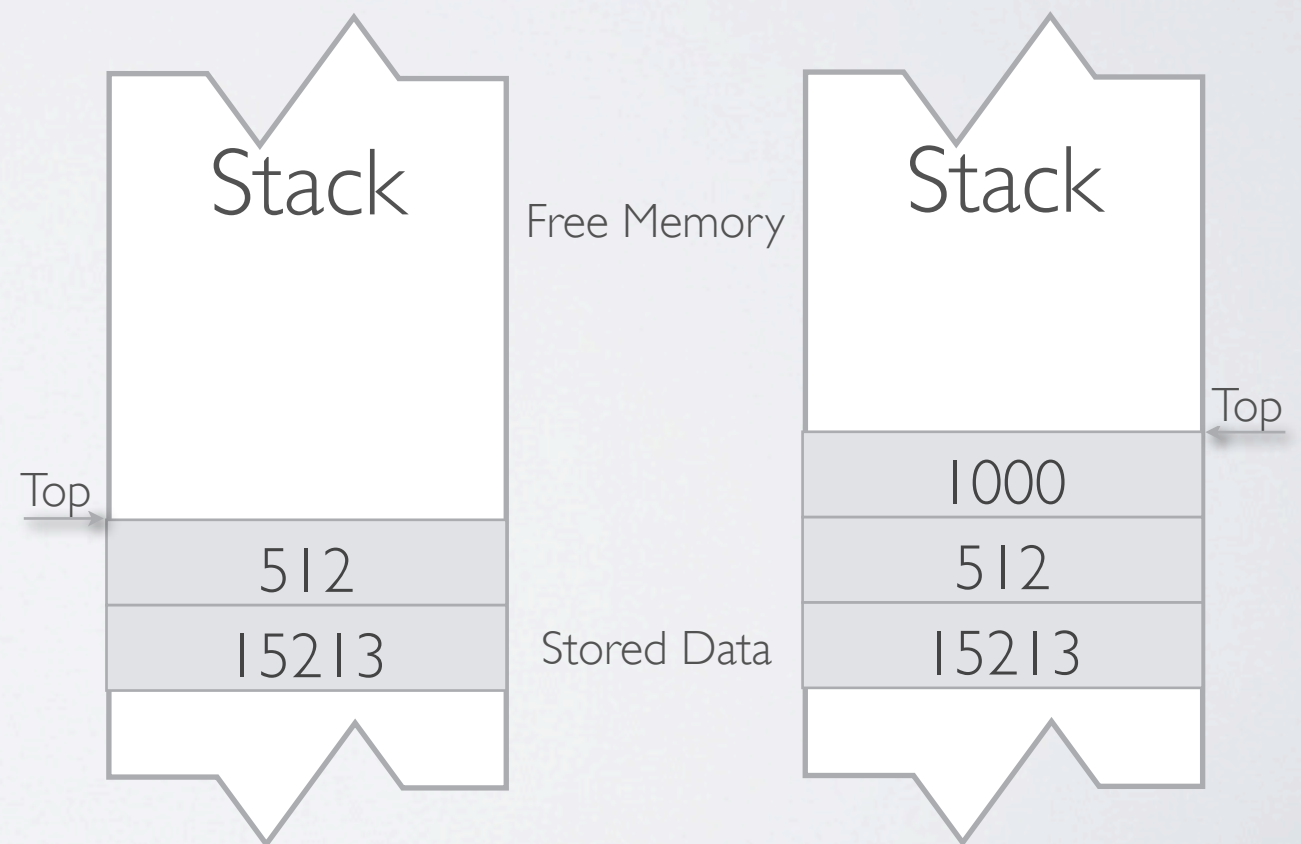
C#: STORAGE_STACK

The stack is an array of memory that acts as a last-in, first-out (LIFO) data structure. It stores several types of data:

- The values of certain types of variables
- The program's current execution environment
- Parameters passed to methods



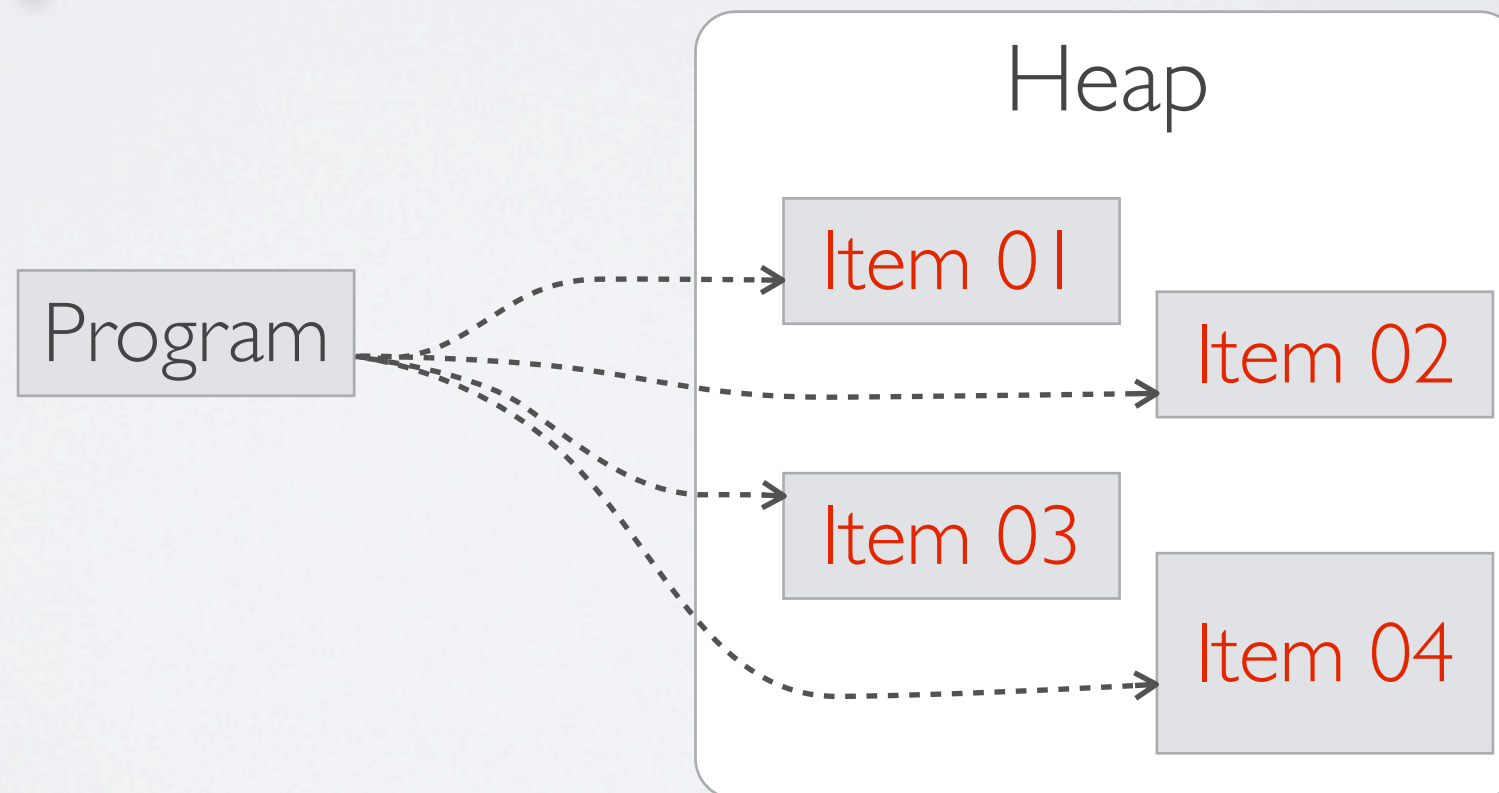
Data items are pushed onto the top of Stack and popped from the top of the stack.



Pushing an integer(e.g. 1000) onto the Stack

C#: STORAGE_HEAP

The heap is an area where chunks of memory can be allocated to store certain kinds of data. Unlike the stack, memory can be stored and removed from the heap in any order. Figure below shows a program that has stored four items in the heap.



C#: TYPES SUMMARY

Value Types and Reference Types in C#

	Value Types				Reference Types
Predefined Types	sbyte	byte	float	short	object
	ushort	double	int	uint	string
	char	long	ulong	decimal	
	bool				
User-Defined Types	struct				class
	enum				interface
					delegate
					array

C#: VARIABLES

A general-purpose programming language must allow a program to **store** and **retrieve** data. A variable is a name that represents data stored in memory during program execution.

Variable Declarations

A variable must be declared before it can be used. The variable declaration defines the variable, and accomplishes two things:

- It gives the variable a **name** and associates a **type** with it.
- It allows the compiler to **allocate memory** for it.

C#: VARIABLES

example:

```
class Program
```

```
{
```

```
    int num = 5;
```

```
    .....
```

```
}
```

int : Type

num : Variable name

Multiple-Variable declaration:

```
// Variable declarations--some with initializers,
```

```
//some without
```

```
int var3 = 7, var4, var5 = 3;
```

```
double var6, var7 = 6.52;
```


C#: OPERATORS

Assignment Operator(=)

Mathematical Operators(+, -, *, /, %)

Increment/Decrement Operators (+=, -=, *=, /=(?))

Prefix/Postfix Operators (++ , --)

Rational Operators(==, !=, >, >=, <, <=)

Logical Operators(&&, ||, !)

C#: OPERATORS

The Conditional Operator (? :)

The conditional operator is a powerful and succinct way of returning one of two values, based on the result of a condition.

< Syntax > **Condition ? value01 : value02**

if..else operator

```
int intVar;  
  
if ( x < y )  
    intVar = 5;  
else  
    intVar = 10;
```

consitional operator

```
int intVar;  
  
intVar = x < y ? 5 : 10;
```


C#: OPERATOR_IS

The **is** Operator

The **is** operator can be used to check whether a conversion would complete successfully. The syntax of the **is** operator is the following, where **Expr** is the source expression:

Expr is TargetType

Returns a bool



```
class Employee : Person { }
class Person
{
    public string Name = "Anonymous";
    public int Age    = 25;
}
class Program
{
    static void Main()
    {
        Employee bill = new Employee();
        Person p;
        if( bill is Person ) // Check if variable bill can be converted to type Person
        {
            p = bill;
            Console.WriteLine("Person Info: {0}, {1}", p.Name, p.Age);
        }
    }
}
```

C#: OPERATOR_AS

The **as** Operator

The **as** operator is like the cast operator, except that it does not raise an exception. If the conversion fails, rather than raising an exception, it sets the target reference to **null**. The syntax of the **as** operator is the following, where

- Expr is the source expression.
- TargetType is the target type, which must be a reference type.

Expr **as** TargetType

Returns a Object

```
class Employee : Person { }
class Person
{
    public string Name = "Anonymous";
    public int Age    = 25;
}
class Program
{
    static void Main()
    {
        Employee bill = new Employee();
        Person p;
        p = bill as Person;
        if( p != null )
        {
            Console.WriteLine("Person Info: {0}, {1}", p.Name, p.Age);
        }
    }
}
```


C#: STATEMENTS

conditional branching statements

if

if else

switch

while

do...while

for

if (TestExpr)
 Statement

```
// With a block - Multiple statements
if( x >= 20 )
{
    // Block--braces needed
    x = x - 5;
    y = x + z;
}
```

If (TestExpr)
 Statement1
else
 Statement2

```
If( x <= 10 )
    z = x - 1; // Single statement
else
{
    // Multiple statements--block
    x = x - 5;
    y = x + z;
}
```

C#: STATEMENTS

conditional branching statements

if
if else
switch
while
do...while
for

while(TestExpr)
Statement

```
int x = 3;  
while( x > 0 )  
{  
    Console.WriteLine("x: {0}", x);  
    x--;  
}  
Console.WriteLine("Out of loop");
```

x: 3
x: 2
x: 1
Out of loop

Output

do
Statement
while(TestExpr);

```
int x = 3;  
do  
{  
    Console.WriteLine("x: {0}", x++);  
}  
while (x < 6);  
Console.WriteLine("Out of loop");
```

x: 3
x: 4
x: 5
Out of loop

Output

C#: STATEMENTS

conditional branching statements

if
if else
switch
while
do...while
for

for(Initializer ; TestExpr ; IterationExpr)
Statement

```
for( int x=1; x<6; x++ )
{
    switch( x )    // Evaluate the value of variable x.
    {
        case 2:                // If x equals 2
            Console.WriteLine("x is {0} -- In Case 2", x);
            break;              // Go to end of switch.

        case 5:                // If x equals 5
            Console.WriteLine("x is {0} -- In Case 5", x);
            break;              // Go to end of switch.

        default:                // If x is neither 2 nor 5
            Console.WriteLine("x is {0} -- In Default case", x);
            break;
    }
}
```

C#: CLASS

- **class** types
- struct types
- array types
- enum types
- delegate types
- interface types

A class is a data structure that can store data and execute code. It contains the following:

- Data members, which store data associated with the class or an instance of the class. Data members generally model the attributes of the real-world object the class represents.
- Function members, which execute code. Function members generally model the functions and actions of the real-world object the class represents.

Types of Class Members:

Data Members–Store Data		Function Members–Execute Code	
Fields		Methods	Operators
Constants		Properties	Indexers
		Constructors	Events
		De-constructors	

C#: CLASS

Class Declaration

A class declaration defines the characteristics and members of a new class. It does not create an instance of the class, but creates the template from which class instances will be created. The class declaration provides the following:

- The class name
- The members of the class
- The characteristics of the class

The diagram shows a C# class declaration with two annotations above it. An arrow labeled 'Type' points to the keyword 'class'. Another arrow labeled 'Class Name' points to the identifier 'MyExcellentClass'. The class body is enclosed in curly braces, containing the text 'MemberDeclarations' followed by an ellipsis '.....' and a closing brace '}'.

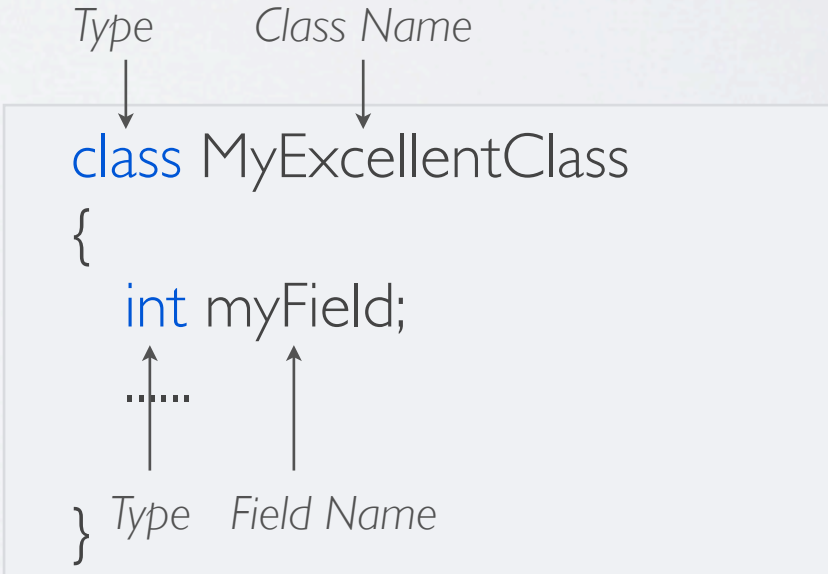
```
      Type      Class Name  
      ↓        ↓  
class MyExcellentClass  
{  
    MemberDeclarations  
    .....  
}
```

C#: CLASS

Class Declaration: **Fields**

A field is a variable that belongs to a class. It can be of any type, either predefined or user-defined. Like all variables, fields store data, and have the following characteristics:

- They can be written to.
- They can be read from.



The diagram shows a C# class declaration with annotations. The code is: `class MyExcellentClass { int myField; }`. Annotations include: 'Type' pointing to 'class', 'Class Name' pointing to 'MyExcellentClass', 'int' pointing to 'int' (with a dotted line from 'myField' above it), and 'Field Name' pointing to 'myField'. A closing brace '}' is also present at the end of the class body.

```
class MyExcellentClass
{
    int myField;
}
```

Type Class Name

int myField;

Type Field Name

C#: CLASS

Class Declaration: **Methods**

A method is a named block of executable code that can be executed from many different parts of the program, and even from other programs. The minimum syntax for declaring a method includes the following components:

- **Return type**: This states the type of value the method returns. If a method does not return a value, the return type is specified as *void*.
- **Name**: This is the name of the method.
- **Parameter list**: This consists of at least an empty set of matching parentheses. If there are parameters, they are listed between the parentheses.
- **Method body**: This consists of a matching set of curly braces, containing the executable code.

C#: CLASS

Class Declaration: **Methods**

Return type

Name

Parameter list

Method body

```

Type      Class Name
↓         ↓
class MyExcellentClass
{
    int myField;

    return type      name      Parameter list
    ↓               ↓         ↓
    void PrintNums (int n1, int n2)
    {
        Console.WriteLine("{0}", n1);
        Console.WriteLine("{0}", n2);
    }
}
} method body
```


C#: CLASS

Class Declaration: **Constructor**

An instance constructor is a special method that is executed whenever a new instance of a class is created.

- A constructor is used to **initialize** the state of the class instance.
- If you want to be able to create instances of your class from outside the class, you need to declare the constructor public.
- The **name** of the constructor is the same as the name of the class.
- A constructor **cannot** have a return value.

```
class MyExcellentClass
{
    DateTime TimeOfInstantiation;           // Field
    ...
    public MyClass()                       // Constructor
    {
        TimeOfInstantiation = DateTime.Now; // Initialize field
    }
    ...
}
```

C#: CLASS

Encapsulating Data with Properties: **get** and **set** Accessors

The **set** and **get** accessors have predefined syntax and semantics. You can think of the set accessor as a method with a single parameter that “sets” the value of the property. The get accessor has no parameters and returns a value from the property.

```
class CI
{
    private int TheRealValue;    // Field: memory allocated

    public int MyValue           // Property: no memory allocated
    {
        set
        {
            TheRealValue = value;
        }

        get
        {
            return TheRealValue;
        }
    }
}
```


C#: STRUCT

Structs are programmer-defined data types, very similar to classes. They have **data members** and **function members**. Although similar to classes, there are a number of important differences. The most important ones are the following:

A variable of a struct type cannot be null.
Two structs variables cannot refer to the same object.

- Classes are reference types and structs are **value** types.
- Structs are implicitly sealed, which means that they cannot be derived from.

The syntax for declaring a struct is similar to that of declaring a class.

```
struct StructName
{
    MemberDeclarations
}
```

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

class Program
{
    static void Main()
    {
        Point first, second, third;

        first.X = 10; first.Y = 10;
        second.X = 20; second.Y = 20;

        Console.WriteLine("first: {0}, {1}", first.X, first.Y);
        Console.WriteLine("second: {0}, {1}", second.X, second.Y);
    }
}
```

C#: ARRAY

An array is a set of **uniform** data elements, represented by a single variable name. The individual elements are accessed using the variable name together with one or more **indexes** between square brackets, for example:

```
int[] intArr1 = new int[15];    // Declare 1-D array.  
intArr1[2] = 10;                // Write to element 2 of the array.  
int var1 = intArr1[2];          // Read from element 2 of the array.
```

```
int[,] intArr2 = new int[5,10]; // Declare 2-D array.  
intArr2[2,3] = 7;               // Write to the array.  
int var2 = intArr2[2,3];        // Read from the array.
```


C#: ENUMERATIONS

An enumeration, or **enum**, is a programmer-defined type, like a class or a struct.

- Like structs, **enums** are value types, and therefore store their data directly, rather than separately, with a reference and data.
- **Enums** have only one type of member: named constants with **integral** values.

```
enum TrafficLight {  
    Green, ← Comma separated—no semicolons  
    Yellow, ← Comma separated—no semicolons  
    Red  
}
```

C#: ENUMERATIONS

```
TrafficLight t1 = TrafficLight.Green;  
TrafficLight t2 = TrafficLight.Yellow;  
TrafficLight t3 = TrafficLight.Red;
```

```
Console.WriteLine("{0},\t{1}", t1, (int) t1);  
Console.WriteLine("{0},\t{1}", t2, (int) t2);  
Console.WriteLine("{0},\t{1}\n", t3, (int) t3);
```

Output

```
Green, 0  
Yellow, 1  
Red, 2
```


C#: ENUMERATIONS

```
enum CardSuit
```

```
{
```

```
    Hearts,
```

```
    Clubs,
```

```
    Diamonds,
```

```
    Spades,
```

```
    MaxSuits
```

```
}
```

```
// 0 - Since this is first
```

```
// 1 - One more than the previous one
```

```
// 2 - One more than the previous one
```

```
// 3 - A common way to assign a constant
```

```
//    to the number of listed items.
```

```
enum FaceCards
```

```
{ // Member
```

```
    Jack
```

```
    = 11,
```

```
// Value assigned
```

```
// 11 - Explicitly set
```

```
    Queen,
```

```
// 12 - One more than the previous one
```

```
    King,
```

```
// 13 - One more than the previous one
```

```
    Ace,
```

```
// 14 - One more than the previous one
```

```
    NumberOfFaceCards = 4,
```

```
// 4 - Explicitly set
```

```
    SomeOtherValue,
```

```
// 5 - One more than the previous one
```

```
    HighestFaceCard
```

```
    = Ace
```

```
// 14 - Ace is defined above
```

```
}
```

DEBUG

How:

```
Debug.WriteLine();  
Debug.Print();  
.....
```

Where:

Intermediate Window

> Under Tool->Options->Debugging->General->

CHOOSE REDIRECT ALL OUTPUT WINDOW TEXT TO THE IMMEDIATE WINDOW

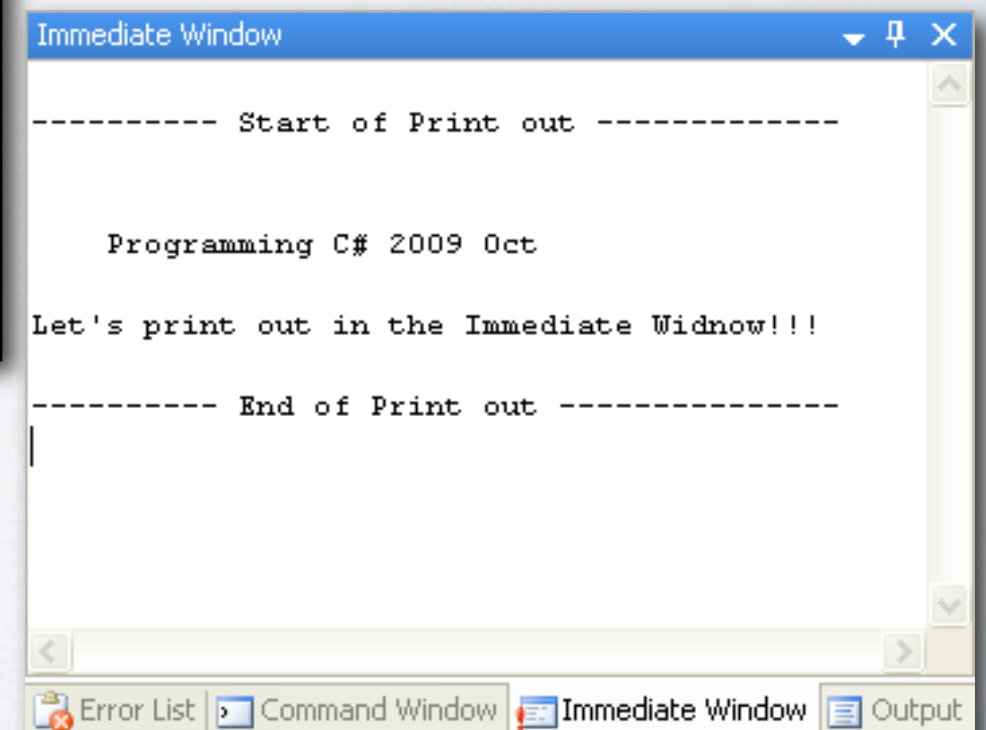
> To clean the outputs in the Immediate Window:

- ✿ type “>cls” in the Intermediate Window, or
- ✿ Right click on the window and choose “**Clear All**”

C#:DEBUG

```
using System;
using System.Diagnostics;

namespace ConsoleApplication4
{
    class Program
    {
        static void Main(string[] args)
        {
            Debug.WriteLine("\n----- Start of Print out -----");
            Debug.WriteLine("\n");
            Debug.Indent();
            Debug.WriteLine("Programming C# 2009 Oct");
            Debug.Print("\nLet's print out in the Immediate Widnow!!!\n");
            Debug.Unindent();
            Debug.WriteLine("----- End of Print out -----");
        }
    }
}
```



C#:DEBUG

How to use immediate windows to work with values :

When you enter commands in the Immediate window, they're executed **in the same context (or scope)** as the application that's running. That means that you can't display the value of a variable that's out of scope.

The commands that you enter into the Immediate window remain there until you exit from Visual Studio or explicitly delete them using the Clear All command in the shortcut menu for the window.

To retrieve/change values or using function

> Enter a **question mark (?)** followed by the expression whose value you want to display.

> To assign a different value to a variable, property, or object, enter an assignment statement in the Immediate window. Then, press the Enter key.

> To execute a user-defined method from the Immediate window, enter its name and any arguments it requires. Then, press the Enter key. If you want to display the value that's returned by a method, precede the method call with a question mark.

C#:THE FORMAT STRING

- The general form of the Write() and WriteLine() statements takes more than a single parameter.
 - >> If there is more than a single parameter, the parameters are separated by commas.
 - >>The first parameter must always be a **string**, and is called the format string.
 - >>The format string can contain **substitution markers**. A substitution marker marks the **position** in the format string where a value should be substituted in the output string.It consists of **an integer enclosed in a set of matching curly braces**.The integer is the numeric position of the substitution value to be used.
 - >>The parameters following the format string are called substitution values. These substitution values are numbered, starting at 0.

C#:THE FORMAT STRING

The syntax is as follows:

```
Console.WriteLine( FormatString, SubVal0, SubVal1, SubVal2, ... );
```

example:

```
Console.WriteLine("Two sample integers are {0} and {1}.", 3, 6);
```



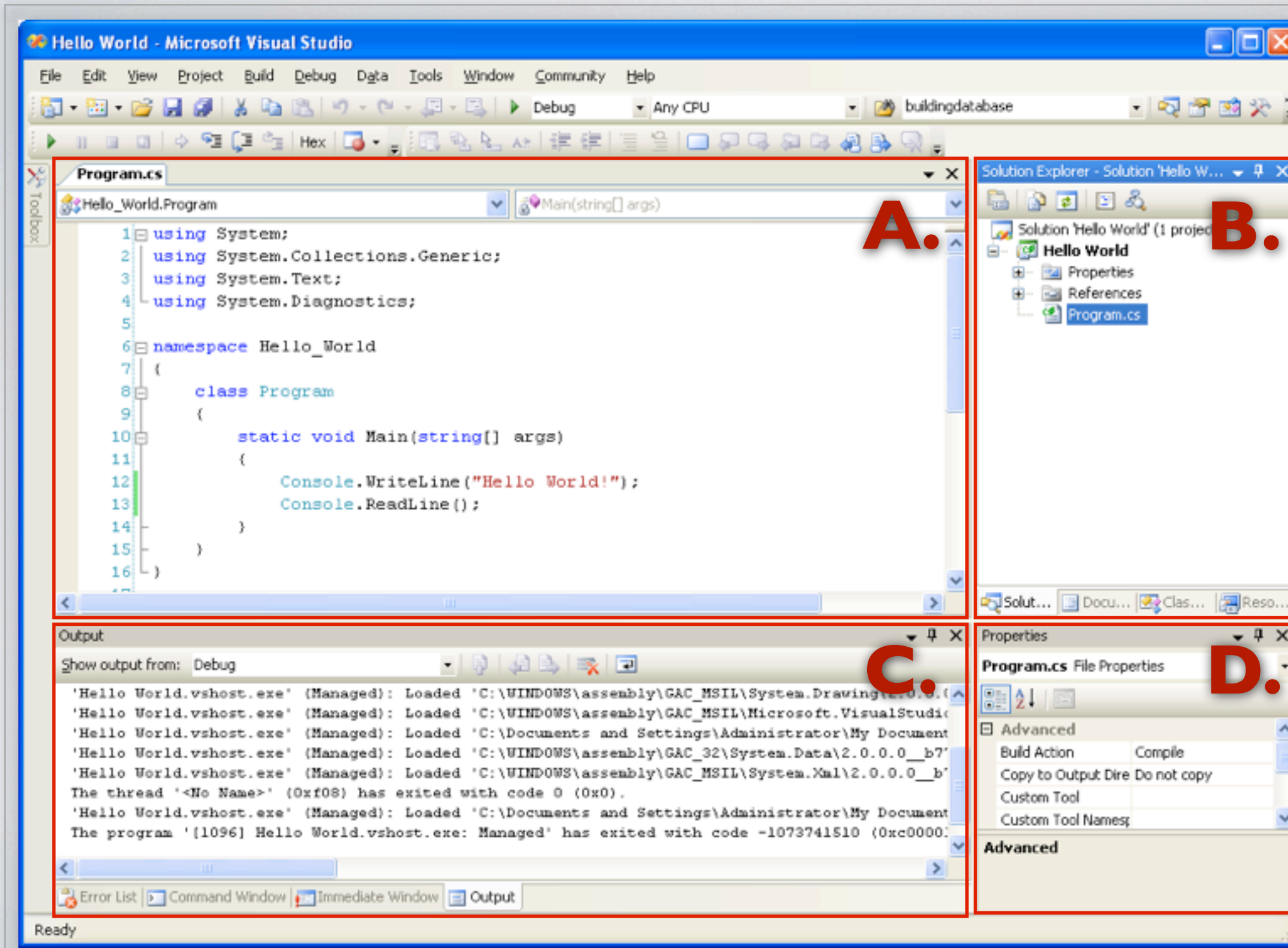
↑
Format string

↑
Substitute values

Output

Two sample integers are 3 and 6.

VISUAL STUDIO UI



- A. Coding Environment
- B. Object Navigation Panel
- C. Transaction
- D. Object Properties