

060010407-GUI Programming

Unit-2 C# Language Basics

Variables

- ▶ variables come in different flavours, known as types.
- ▶ To use variables, you have to declare them. This means that you have to assign them a name and a type.
- ▶ Syntax:

<type> <name>;

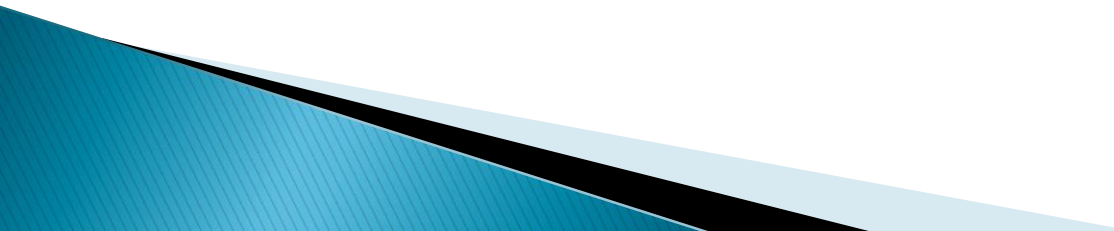
- ▶ **Variable naming:**
- ▶ The first character of a variable name must be either a letter, an underscore character (`_`), or the at symbol (`@`).
- ▶ Subsequent characters may be letters, underscore characters, or numbers.

Variables

- ▶ C# is case sensitive
- ▶ **Variable Declaration and Assignment**

```
int age;  
age=25;
```

Data Types

- ▶ The variables in C#, are categorized into the following types:
 - ▶ Value types
 - ▶ Reference types
 - ▶ Pointer types
 - ▶ **Value types:**
 - ▶ Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.
- 

Data Types

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28–29 significant digits	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / 10^0$ to 28	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D

Data Types

Type	Represents	Range	Default Value
float	32-bit single-precision floating point type	-3.4×10^{38} to $+3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0

Data Types

Type	Represents	Range	Default Value
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

Data Types

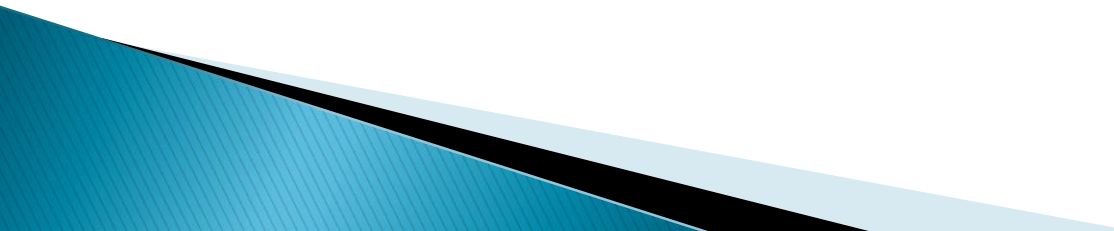
- ▶ To know size of a data type:
- ▶ Syntax:
- ▶ sizeof(type)
- ▶ Example:

```
using System;
namespace DataTypeApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Size of int: {0}", sizeof(int));
            Console.ReadLine();
        }
    }
}
```

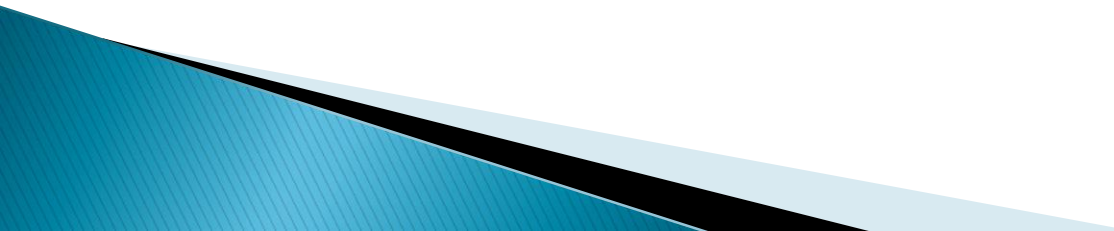

Data Types

- ▶ **Reference Type:**
 - ▶ do not contain the actual data stored in a variable, but they contain a reference to it.
 - ▶ Example of built-in reference types are: object, dynamic, and string.
- ▶ **Object type:**
 - ▶ The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS)
 - ▶ Object is an alias for System.Object class

Data Types

- ▶ When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.
 - ▶ **Dynamic types:**
 - ▶ can store any type of value in the dynamic data type variable.
 - ▶ Type checking for these types of variables takes place at run-time.
- 

Data Types

- ▶ Syntax:
 - ▶ dynamic <variable_name> = value;
 - ▶ Example,
 - ▶ dynamic d = 20;
 - ▶ **String Types:**
 - ▶ The **String Type** allows you to assign any string values to a variable.
 - ▶ an alias for the System.String class.
 - ▶ derived from object type.
- 

Data Types

- ▶ can be assigned using string literals in two forms:
quoted and @quoted.
- ▶ Example:
 1. String str = "Tutorials Point";
 2. @"Tutorials Point";
- ▶ The user-defined reference types are: class, interface, or delegate.

Data Types

- ▶ **Pointer types:**

- ▶ Pointer type variables store the memory address of another type.

- ▶ Syntax:

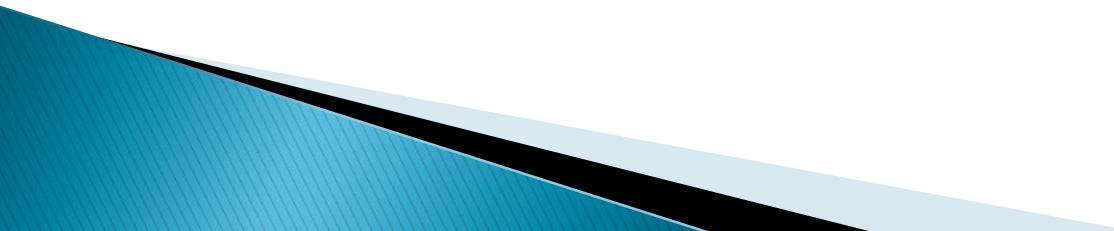
`type* identifier;`

- ▶ Example:

`char* cptr;`

`int* iptr;`

Operators

- ▶ Types of operators:
 - ▶ Arithmetic Operators
 - ▶ Relational Operators
 - ▶ Logical Operators
 - ▶ Bitwise Operators
 - ▶ Assignment Operators
 - ▶ Misc Operators
- 

Arithmetic operators

Operator	Description	Example
+	Adds two operands	$A + B = 30$
-	Subtracts second operand from the first	$A - B = -10$
*	Multiplies both operands	$A * B = 200$
/	Divides numerator by de-numerator	$B / A = 2$
%	Modulus Operator and remainder of after an integer division	$B \% A = 0$
++	Increment operator increases integer value by one	$A++ = 11$
--	Decrement operator decreases integer value by one	$A-- = 9$

Relational operators

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

Relational operators

Operator	Description	Example
\leq	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$(A \leq B)$ is true.

Logical operators

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

Bitwise operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand	A << 2 = 240, which is 1111 0000

Bitwise operators

Operator	Description	Example
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$A \gg 2 = 15$, which is 0000 1111

Assignment Operators

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

Assignment Operators

Operator	Description	Example
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<code><<=</code>	Left shift AND assignment operator	<code>C <<= 2</code> is same as <code>C = C << 2</code>
<code>>>=</code>	Right shift AND assignment operator	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	bitwise exclusive OR and assignment operator	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	bitwise inclusive OR and assignment operator	<code>C = 2</code> is same as <code>C = C 2</code>

Miscellaneous Operators

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), returns 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; returns actual address of the variable.
*	Pointer to a variable.	*a; creates pointer named 'a' to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
as	Cast without raising an exception if the cast fails.	Object obj=new StreamReader("Hello"); StreamReader r=obj as StreamReader;

Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right

Operator Precedence

Category	Operator	Associativity
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

Boxing and Unboxing

- ▶ **Boxing:**

- ▶ Implicit conversion of a value type (int, char etc.) to a reference type (object), is known as Boxing.
- ▶ In boxing process, a value type is being allocated on the heap rather than the stack.

- ▶ **Unboxing:**

- ▶ Explicit conversion of same reference type (which is being created by boxing process); back to a value type is known as unboxing.
- ▶ In unboxing process, boxed value type is unboxed from the heap and assigned to a value type which is being allocated on the stack.

Boxing and Unboxing

On the Stack

stackVar

12

```
int stackVar = 12
```

boxedVar



```
object boxedVar = stackVar
```

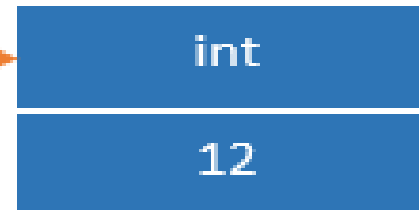
unBoxed

12

```
int unBoxed = (int)boxedVar
```

On the Heap

(stackVar boxed)



@dotnet-tricks.com

Boxing and unboxing

Boxing and Unboxing

▶ Example:1

```
int stackVar = 12;
```

```
object boxedVar = stackVar; //boxing
```

```
int unBoxed = (int)boxedVar;//unboxing
```

▶ Example:2

```
int i = 10;
```

```
ArrayList arrlst = new ArrayList();
```

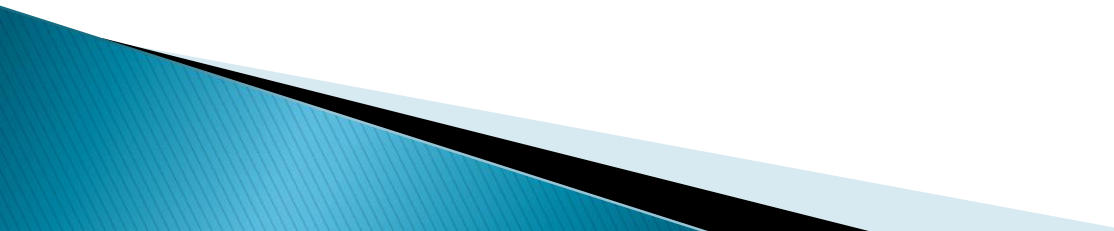
```
arrlst.Add(i); // Boxing occurs automatically
```

```
int j = (int)arrlst[0]; // Unboxing occurs
```

Boxing and Unboxing

- ▶ Sometimes boxing is necessary, but you should avoid it if possible, since it will slow down the performance and increase memory requirements.

Flow Control

- ▶ There are three types of statements available to manage smooth execution of a program.
 - ▶ **Selection statement**
 - ▶ **Iteration statement**
 - ▶ **Jump statement**
-
- ▶ **Selection Statement:**
 - ▶ A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is **true** or not.
- 

Flow Control

- ▶ If_else statement and Switch_case statement are two types of selection statements.
- ▶ if statement selects a statement for execution based on the value of a Boolean expression.
- ▶ if-statement:
 - ▶ if (boolean-expression) embedded-statement
 - ▶ if (boolean-expression) embedded-statement else embedded-statement
- ▶ An else part is associated with the lexically nearest preceding if that is allowed by the syntax

Flow Control

▶ if (x) if (y) F(); else G();

Is equivalent to:

```
if (x)
```

```
{
```

```
    if (y)
```

```
    {
```

```
        F();
```

```
    }
```

```
else
```

```
{
```

```
    G();
```

```
}
```

```
}
```


Flow Control

- ▶ The **switch** statement is a control statement that selects a *switch section* to execute from a list of candidates.

- ▶ Example:

```
int caseSwitch = 1;
switch (caseSwitch)
{
case 1: Console.WriteLine("Case 1");
    break;
case 2: Console.WriteLine("Case 2");
    break;
default: Console.WriteLine("Default case");
    break;
}
```

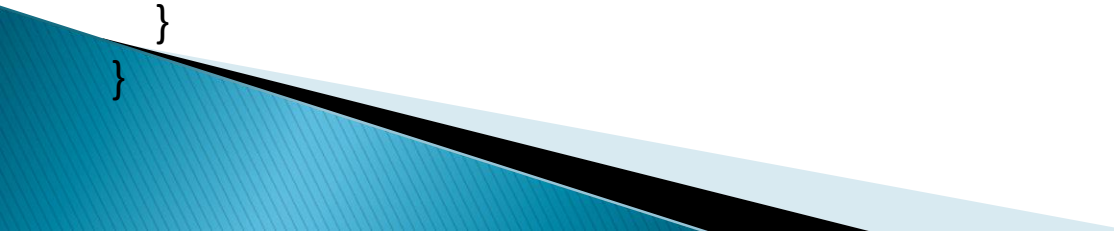
Flow Control

- ▶ **Jump statement:(break,continue,return,throw)**
- ▶ The goto statement transfers the program control directly to a labeled statement.
- ▶ using System;

```
class Program
{
    static void Main()
    {
        Console.WriteLine(M());
    }
    static int M()
    {
        int dummy = 0;
        for (int a = 0; a < 10; a++)
```

Flow Control

```
{
    for (int y = 0; y < 10; y++) // Run until condition.
    {
        for (int x = 0; x < 10; x++) // Run until condition.
        {
            if (x == 5 &&
                y == 5)
            {
                goto Outer;
            }
        }
        dummy++;
    }
Outer:
    continue;
}
return dummy;
}
```



Flow Control

- ▶ The break statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement.

- ▶ Example:

```
class BreakTest
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        for (int i = 1; i <= 100; i++)
```

```
        {
```

```
            if (i == 5)
```

```
            {
```

```
                break;
```

```
            }
```

```
            Console.WriteLine(i);
```

```
        }
```

```
        // Keep the console open in debug mode.
```

```
        Console.WriteLine("Press any key to exit.");
```

```
        Console.ReadKey();
```

```
    }
```

```
}
```

Flow Control

- ▶ The continue statement passes control to the next iteration of the enclosing while, do, for, or foreach statement in which it appears.
- ▶ Example:

class ContinueTest

```
{  
    static void Main()  
    {  
        for (int i = 1; i <= 10; i++)  
        {  
            if (i < 9)  
            {  
                continue;  
            }  
            Console.WriteLine(i);  
        }  
        // Keep the console open in debug mode.  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}
```

Flow Control

- ▶ The return statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value.
- ▶ Example:

class ReturnTest

```
{  
    static double CalculateArea(int r)  
    {  
        double area = r * r * Math.PI;  
        return area;  
    }  
    static void Main()  
    {  
        int radius = 5;  
        double result = CalculateArea(radius);  
        Console.WriteLine("The area is {0:0.00}", result);  
        // Keep the console open in debug mode.  
        Console.WriteLine("Press any key to exit.");  
        Console.ReadKey();  
    }  
}
```

Flow Control

- ▶ The throw statement is used to signal the occurrence of an anomalous situation (exception) during the program execution.
- ▶ Example:

```
public class ThrowTest2
{

    static int GetNumber(int index)
    {
        int[] nums = { 300, 600, 900 };
        if (index > nums.Length)
        {
            throw new IndexOutOfRangeException();
        }
        return nums[index];
    }

    static void Main()
    {
        int result = GetNumber(3);
    }
}
```

Function (Method)

- ▶ A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

- ▶ To use a method, you need to:

Define the method

Call the method

- ▶ Defining a method:

<Access Specifier> <Return Type> <Method Name>(Parameter List)

```
{  
    Method Body
```

```
}
```

- ▶ Example:

```
class NumberManipulator
```

```
{  
    public int FindMax(int num1, int num2)  
    {  
        /* local variable declaration */  
        int result;  
        if (num1 > num2)  
            result = num1;  
        else  
            result = num2;  
        return result;  
    }  
}
```


Function (Method)

- ▶ Calling a method:
- ▶ You can call a method using the name of the method.

Function Argument passing mechanism

- ▶ In C#, arguments can be passed to parameters either by value or by reference.
- ▶ Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment.
- ▶ To pass a parameter by reference, use the ref or out keyword.
- ▶ Example:

```
class Program
```

```
{
```

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

```
{
```

```
    int arg;
```

```
    // Passing by value.
```

```
    // The value of arg in Main is not changed.
```

```
    arg = 4;
```

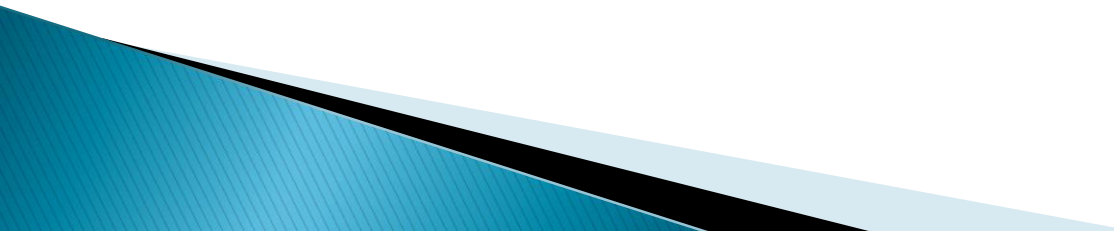
```
    squareVal(arg);
```

```
    Console.WriteLine(arg);
```

Function Argument passing mechanism

```
// Output: 4
//Passing by reference.
// The value of arg in Main is changed.
arg = 4;
squareRef(ref arg);
Console.WriteLine(arg);
// Output: 16
}
static void squareVal(int valParameter)
{
    valParameter *= valParameter;
}
// Passing by reference
static void squareRef(ref int refParameter)
{
    refParameter *= refParameter;
}
}
```

Out parameter

- ▶ you can specify that a given parameter is an out parameter by using the out keyword, which is used in the same way as the ref keyword .
 - ▶ In effect, this gives you almost exactly the same behaviour that the value of the parameter at the end of the function execution is returned to the variable used in the function call.
 - ▶ **Two differences between out and ref parameter:**
 - ▶ it is illegal to use an unassigned variable as a ref parameter, you can use an unassigned variable as an out parameter.
 - ▶ An out parameter must be treated as an unassigned value by the function that uses it.
- 

Out parameter

```
public class Example
{
    public static void Main() //calling method
    {
        int val1 = 0; //must be initialized
        int val2; //optional
        Example1(ref val1);
        Console.WriteLine(val1); // val1=1
        Example2(out val2);
        Console.WriteLine(val2); // val2=2
    }
    static void Example1(ref int value) //called method
    {
        value = 1;
    }
    static void Example2(out int value) //called method
    {
        value = 2; //must be initialized
    }
}
/* Output
1
2
*/
```

Variable scope

```
class program
```

```
{
```

```
static void Write()
```

```
{
```

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

```
}
```

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

```
{
```

```
string myString = "String defined in Main()";
```

```
Write();
```

```
Console.ReadKey();
```

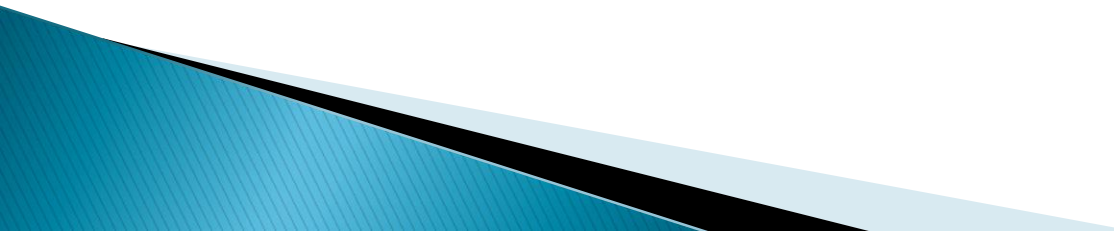
```
}
```

```
}
```

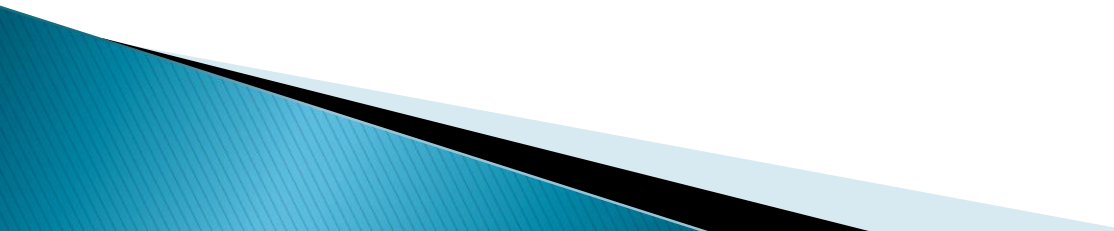
- ▶ This code will give u error that variable myString is not defined in write method.

Global Data

```
class Program
{
    static int val;
    static void ShowDouble()
    {
        val *= 2;
        Console.WriteLine("val doubled = {0}", val);
    }
    static void Main(string[] args)
    {
        val = 5;
        Console.WriteLine("val = {0}", val);
        ShowDouble();
        Console.WriteLine("val = {0}", val);
    }
}
```



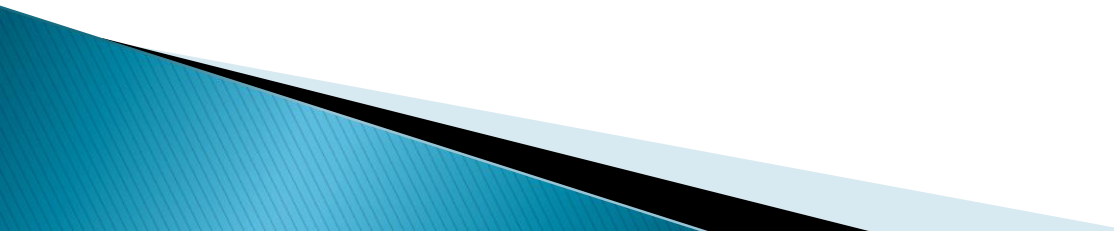
Global Data

- ▶ This limits the versatility of the function slightly and means that you must continuously copy the global variable value into other variables if you intend to store the results.
 - ▶ global data might be modified by code elsewhere in your application, which could cause unpredictable results.
 - ▶ Sometimes you only want to use a function for one purpose, and using a global data store reduces the possibility that you will make an error in a function call, perhaps passing it the wrong variable.
- 

Overloading Function

- ▶ Function overloading provides you with the capability to create multiple functions with the same name, but each working with different parameter types.

```
class Program
{
    static int MaxValue(int[] intArray)
    {
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
                maxVal = intArray[i];
        }
        return maxVal;
    }
}
```



Overloading Function

```
static void Main(string[] args)
{
    int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
    int maxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    Console.ReadKey();
}
```

- ▶ This function can only be used with arrays of int values. You could provide different named functions for different parameter types, perhaps renaming the preceding function as `IntArrayMaxValue()` and adding functions such as `DoubleArrayMaxValue()` to work with other types.

Delegates

- ▶ A delegate is a type that enables you to store references to functions.
- ▶ declared much like functions, but with no function body and using the delegate keyword.
- ▶ declaration specifies a return type and parameter list.
- ▶ you can pass a delegate variable to a function as a parameter, and then that function can use the delegate to call whatever function it refers to, without knowing what function will be called until runtime.

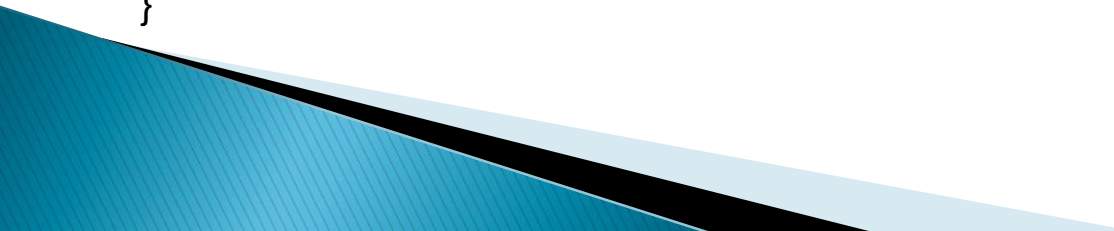
Example:

Class program

```
{  
delegate double ProcessDelegate(double param1, double param2);  
static double Multiply(double param1, double param2)  
{  
return param1 * param2;  
}  
static double Divide(double param1, double param2)  
{  
return param1 / param2;  
}
```

Delegates

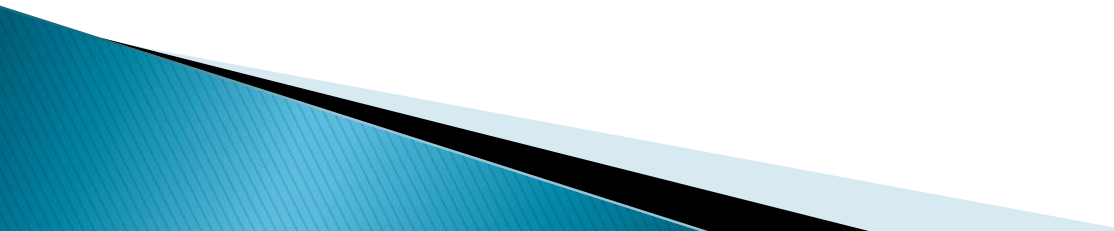
```
static void Main(string[] args)
{
    ProcessDelegate process;
    Console.WriteLine("Enter 2 numbers separated with a comma:");
    string input = Console.ReadLine();
    int commaPos = input.IndexOf(',');
    double param1 = Convert.ToDouble(input.Substring(0, commaPos));
    double param2 = Convert.ToDouble(input.Substring(commaPos + 1, input.Length -
        commaPos - 1));
    Console.WriteLine("Enter M to multiply or D to divide:");
    input = Console.ReadLine();
    if (input == "M")
        process = new ProcessDelegate(Multiply);
    else
        process = new ProcessDelegate(Divide);
    Console.WriteLine("Result: {0}", process(param1, param2));
    Console.ReadKey();
}
```



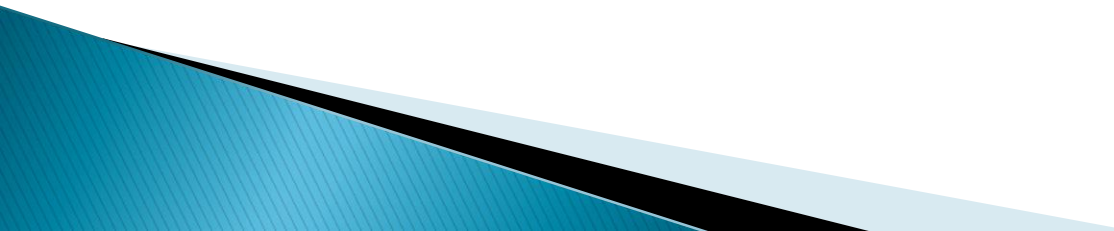
Error handling

- ▶ Errors in application's logic is known as semantic error.
- ▶ Fatal errors include simple errors in code that prevent compilation or more serious problems that occur only at runtime called syntax errors.

Structured exception handling

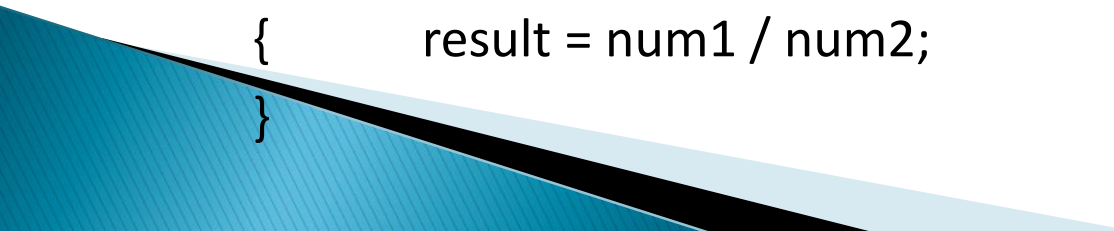
- ▶ try . . . catch . . . Finally
 - ▶ try— Contains code that might throw exceptions.
 - ▶ catch—Contains code to execute when exceptions are thrown. catch blocks may be set to respond only to specific exception types.
 - ▶ finally— Contains code that is always executed, either after the try block if no exception occurs, after a catch block if an exception is handled.
- 

Structured exception handling

- ▶ The try block terminates at the point where the exception occurred.
 - ▶ If a catch block exists, then a check is made to determine whether the block matches the type of exception that was thrown.
 - ▶ If no catch block exists, then the finally block (which must be present if there are no catch blocks) executes.
 - ▶ If a catch block exists but there is no match, then a check is made for other catch blocks.
 - ▶ If a catch block matches the exception type, then the code it contains executes, and then the finally block executes if it is present.
 - ▶ If no catch blocks match the exception type, then the finally block of code executes if it is present.
- 

Try...catch...finally

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
        int result;
        DivNumbers()
        {
            result = 0;
        }
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
        }
    }
}
```



Try...catch...finally

```
catch (DivideByZeroException e)
{
    Console.WriteLine("Exception caught: {0}", e);
}
finally
{
    .WriteLine("Result: {0}", result);
}
}
static void Main(string[] args)
{
    DivNumbers d = new DivNumbers();
    d.division(25, 0);
    Console.ReadKey();
}
}
```

