Chapter 1

Introduction to the C# Language

Contents

Ebook: from chapter 1 to 6 (Part I)

What is the .NET Framework?

- The .NET Framework is a platform created by Microsoft for developing applications
  - Currently in version 4
  - The .NET Framework has been designed to be used from any language: C#, Visual Basic, C++, JScript,…
- The .NET Framework includes a Common Type System (CTS) and Common Language Runtime (CLR)
  - CTS: contains data and some of the most fundamental of these
  - CLR: responsible for maintaining the execution of all applications

Execute applications in .NET Framework

- To execute an application, it must be converted into a language that the target operating system understands, known as *native code*
  - *This conversion is performed* by a *compiler*
- Under the .NET Framework, this is a two-stage process
  - First, applications are compiled into CIL (*Common Intermediate Language)*
  - Second, the JIT (*just-in-time*) compiles this CIL into native code

  → Program → CIL → native code
- Only at this point can the OS execute the application

Garbage Collection

- This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use.
- Garbage collection works by *periodically* inspecting the memory of your computer and removing anything from it that is no longer needed

What is C#?

- C# is one of the languages included in the .NET Framework
  - C# is an object-oriented programming language
- Applications you can write with C#
  - Windows applications
  - Web applications
  - Web services
  - ...
- Tools to write C# program
  - Visual Studio 2010 (Ultimate)
  - Visual C# Express

How to install?

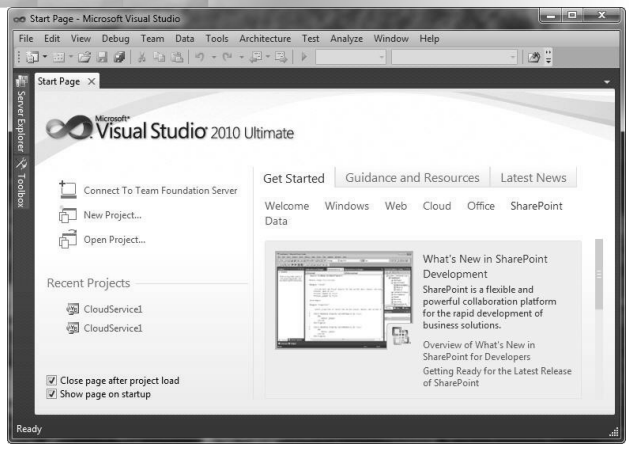
- System requirements

Operating System	Windows XP SP3 (All editions except Starter), Windows Vista SP2 (All editions except Starter), Windows 7
Processor	Computer with 1.6GHz or faster processor
RAM	At least 1GB (32 bit) or 2GB(64 bit)
Hard Disk Space	At least 3GB of hard disk space
Video	DirectX 9 capable video card 1024×768 resolution or higher

- Video to install

Contents

The Development Environment



Basic concepts in C#

- Project
  - Group of related files, images, and documentations
- Solution
  - Group of projects creating one or a group of applications
- Console Application
  - Application that runs in the DOS
- Windows Application
  - Application that runs in the Windows OS
    - Microsoft Word, Microsoft Internet Explorer,…

Basic concepts in C# (cont.)

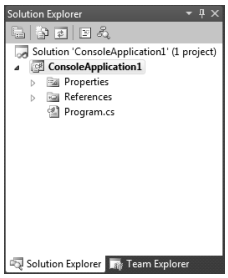
- Class
  - A class has properties, methods and events
- Method
  - The `Main` method
    - Each program must have exactly one
    - All programs start by executing the `Main` method
- Statement
  - Every statement must end in a semicolon (;)

Basic concepts in C# (cont.)

- Namespace (p.51)
  - Namespaces are used as a means of categorizing items
  - Within a namespace, you can declare:
    - another namespace
    - class
    - interface
    - struct
    - enum
    - delegate
- C# is case sensitive

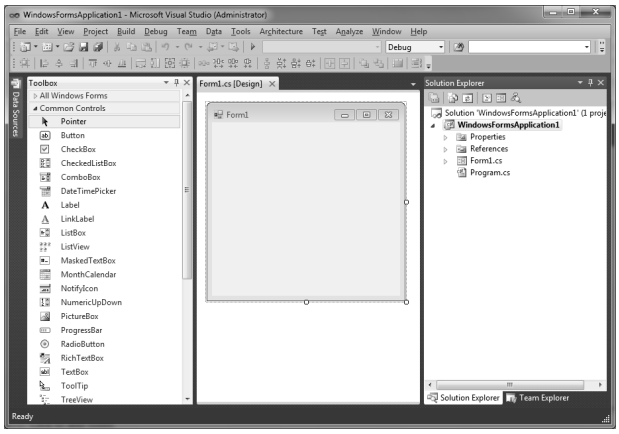
Console Application (try it out p.18)

- Basic Console Application structure




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Output text to the screen.
            Console.WriteLine("The first app in Beginning C# Programming!");
            Console.ReadKey();
        }
    }
}
```


Windows Forms Application (try it out p.25)



The Solution Explorer

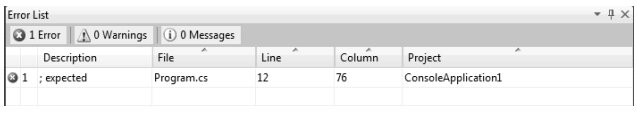
- A window to list all files in the solution
- Perform some operations on files: rename, delete
- The **start up project** is the project that runs when the program is executed
  - It appears in bold in the **`Solution Explorer`**
- **`Solution Explorer`** toolbar
  - **`Properties`** icon
  - **`Shows All Files`** icon
  - **`Refresh`** icon
  - **`View Code`** icon
- To display the **`Solution Explorer`**, select View\Solution Explorer or by pressing Ctrl+W,S

The Properties Window

- A window to show additional information about whatever you select
- Help the programmers alter controls visually without writing code
- **`Properties Window`** toolbar
  - **`Alphabetic`** icon: arranges the properties alphabetically
  - **`Categorized`** icon: arranges the properties by category
  - **`Event`** icon: allows reactions to user actions
- To display the **`Properties Window`**, select View\Properties Window or by pressing Ctrl+W,P

The Error List

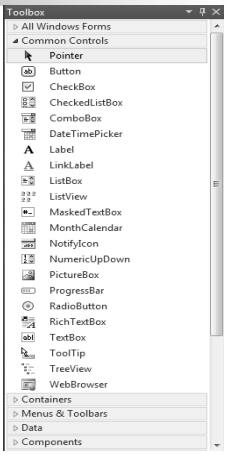
- Display the Errors, Warnings, and Messages produced as you edit and compile code



- Double-click any error message entry to open the file where the problem occurs, and move to the error location

The Toolbox

- Contains reusable controls
- Visual programming allows '**drag and drop**' of controls
- Activate the toolbox by selecting View\Toolbox or by pressing Ctrl+W,X



Contents

1. Introducing C#
2. Writing a C# Program
3. Variables, Constants and Expressions
4. Flow control
5. More about variables
6. Methods
7. Using some classes

Variables, Constants

- Declare variable:

```
DataType   name;
DataType   name = init_value;
```

  - Example: int x, y;
  - Note: Variables need to be initialized before it's used

- Declare constant: `const DataType CONST_NAME = value;`
  - Example: double const PI = 3.14;

- Data types that are built into C#
  - 14 primitive data types: string, int, double, char, long,…(see table)

Primitive Data Types

Type	Size in bits	Values	Standard
`bool`	8	`true` or `false`	
`char`	16	`'\u0000'` to `'\uFFFF'`	(Unicode character set)
`byte`	8	0 to 255	(unsigned)
`sbyte`	8	-128 to +127	
`short`	16	–32,768 to +32,767	
`ushort`	16	0 to 65,535	(unsigned)
`int`	32	–2,147,483,648 to 2,147,483,647	
`uint`	32	0 to 4,294,967,295	(unsigned)
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	
`ulong`	64	0 to 18,446,744,073,709,551,615	(unsigned)
`decimal`	128	1.0×10^{-28} to 7.9×10^{28}	
`float`	32	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	(IEEE 754 floating point)
`double`	64	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	(IEEE 754 floating point)
`string`			(Unicode character set)

The basic variable naming rules

- The first character of a variable name must be either a letter, _ or @
- Subsequent characters may be letters, underscore characters, or numbers
- Not use keywords
- Example:

Right variable names:	Wrong variable names:
myBigVar	99BottlesOfBeer
VAR1	namespace
_test	It's-All-Over

Example: Using variables

```
class Program
{
    static void Main(string[] args)
    {
        int myInteger;
        string myString;
        myInteger = 17;
        myString = "\"myInteger\" is";
        Console.WriteLine("{0}  {1}.", myString, myInteger);
        Console.ReadKey();
    }
}
```


Expression

- Expressions are built from *operators* and operands (variables or literal values)
  - Example: x = 2 * (a + b)

- Operators:
  - Mathematical operators: +, -, *, /, %
  - Increment and decrement operators: ++, --
  - Assignment operators: =, +=, -=, *=, /=, %=
  - Comparison operators: !, &&, ||
  - Conditional operators (p.70): Exp1 ? Exp2 : Exp3;

Operator Precedence

PRECEDENCE	OPERATORS
Highest	++, -- (used as prefixes); +, - (unary)
	*, /, %
	+, -
	=, *=, /=, %=, +=, -=
Lowest	++, -- (used as suffixes)

- Example:
  - var1 = var2 + var3 * var4;
  - int var1, var2 = 5, var3 = 6;
    var1 = var2++ * --var3;
- Try it out p.47

Contents

Flow control (chapter 4)

- Selection structure
  - The **`if`** and **`if/else`** statements (p.70)
  - The **`switch`** statement (p.74)
- Repetition structure
  - The **`do`** loops (p.78)
  - The **`while`** loops (p.80)
  - The **`for`** loops (p.83)
- Interrupting loops (p.87): break, continue, goto, return

Selection structure: *if – if/else*

- Syntax:

```
if (condition)
        statement;
```

```
if (condition)
        statement1;
else
        statement2;
```

- Example:

```
if (diem >= 5)
    Console.WriteLine( "DAU" );
else
    Console.WriteLine( "ROT" );
```


Selection structure: *switch*

- The **`switch`** statement
  - Expression
    - String
    - Integral
  - Cases
    - Case '*x*' :
      - Use of constant variable cases
    - Empty cases
    - The default case
  - The **`break`** statement
    - Exit the **`switch`** statement

```
switch (Expression)
{
        case const_1:
                statements;
                break;
        case const_2:
                statements;
                break;
        …
        default:
                statements;
}
```


Repetition structure: *for*

- Syntax:

```
for (Expression1; Expression2; Expression3)
        statement;
```

  - Expression1: names the control variable
  - Expression2: loop-continuation condition
  - Expression3: incrementing/decrementing
    - If Expression1 has several variables, Expression3 must have several variables accordingly
    - ++counter and counter++ are equivalent
- Example:

```
for (counter = 1; counter <= 5; counter++ )
   Console.WriteLine(counter*10);
```

Repetition structure: *while*

- Syntax:

```
while (condition)
    statement;
```

- Example:

```
int i=1;
while (i<=10)
{
    Console.Write("{0}    ", i);
    i++;
}
```


Repetition structure: *do*

- Syntax:

```
do {
    statement;
} while (condition);
```

- Example:

```
int i=1;
do {
    Console.Write("{0}    ", i);
    i++;
} while (i<=10);
```


`while` vs. `do/while`

- Using a **`while`** loop
  - Condition is tested
  - The action is performed
  - Loop could be skipped altogether
- Using a **`do/while`** loop
  - Action is performed
  - Then the loop condition is tested
  - Loop must be run though once
  - Always uses brackets ({) to prevent confusion

Statements `break` and `continue`

- Used to alter the flow of control
  - The **`break`** statement
    - Used to exit a loop early
  - The **`continue`** statement
    - Used to skip the rest of the statements and begin the loop at the first statement in the loop

Example: *break*

```
string output = "";
int count;

for ( count = 1; count <= 10; count++ )
{
    if ( count == 5 )
        break;
    output += count + " ";
}
output += "\nBroke out of loop at count = " + count;

Console.WriteLine( output );
```


Example: *continue*

```
string output = "";
for ( int count = 1; count <= 10; count++ )
{
    if ( count == 5 )
        continue;
    output += count + " ";
}
output += "\nUsed continue to skip printing 5";

Console.WriteLine( output );
```

Contents

1. Introducing C#
2. Writing a C# Program
3. Variables, Constants and Expressions
4. Flow control
5. More about variables
6. Methods
7. Using some classes

- ❖ Type Conversion
- ❖ Enumeration
- ❖ Struct

Type Conversion

- Implicit conversion (table p.95)
  - Example:

```
ushort i;
char c = 'a';
i = c;
Console.WriteLine("value of c: {0}", c);
Console.WriteLine("value of i: {0}", i);
```

Output:
value of c: a
value of i: 97

- Explicit conversion (p.96)
  - Syntax: (DataType)variable
    - Example:

```
float x = (float)7/2;
```

  - Explicit conversions using the Convert commands

Explicit Conversions using the Convert commands

COMMAND	RESULT
Convert.ToBoolean(val)	val converted to bool
Convert.ToByte(val)	val converted to byte
Convert.ToChar(val)	val converted to char
Convert.ToDecimal(val)	val converted to decimal
Convert.ToDouble(val)	val converted to double
Convert.ToInt16(val)	val converted to short
Convert.ToInt32(val)	val converted to int
Convert.ToInt64(val)	val converted to long
Convert.ToSByte(val)	val converted to sbyte
Convert.ToSingle(val)	val converted to float
Convert.ToString(val)	val converted to string
Convert.ToUInt16(val)	val converted to ushort
Convert.ToUInt32(val)	val converted to uint
Convert.ToUInt64(val)	val converted to ulong

Enumeration (p.102)

- *An enumeration is a user-defined integer type*
- Defining Enumeration

```
enum <typeName> {
        <value1>,
        <value2>,
        ...
        <valueN>
}
```

By default, each value is assigned a corresponding value automatically according to the order in which it is defined, starting from zero. This means that *<value1> gets the value 0, <value2>* gets 1 *and so on.*

- Declare variables of enum: ***<typeName> <varName>;***
- Access value of the enum: ***<typeName>.<value>***

Enumeration: Example

```
public enum TimeOfDay
{
    Morning,
    Afternoon,
    Evening
}
class EnumExample
{
    public static void Main()
    {
        TimeOfDay time;
        time = TimeOfDay.Morning;
        switch( time )
        {
            case TimeOfDay.Morning:
                Console.WriteLine("Good morning!");
                break;
            case TimeOfDay.Afternoon:
                Console.WriteLine("Good afternoon!");
                break;
            case TimeOfDay.Evening:
                Console.WriteLine("Good evening!");
                break;
            default:
                Console.WriteLine("Hello!");
                break;
        }
    }
}
```


Struct (p.107)

- Defining Struct

```
struct <typeName>
{
        <accessibility>  <type>  <name>;
}
```

- Declare variable of the struct: ***<typeName> <varName>;***
- Access value of the struct: ***<varName>.<name>***
- Example:

```
struct Dimensions
{
    public double Length;
    public double Width;
}
```

```
public static void Main()
{
        Dimensions d;
        d.Length = 10;
        d.Width = 5;
        //...
}
```

Contents

1. Introducing C#
2. Writing a C# Program
3. Variables, Constants and Expressions
4. Flow Control
5. More About Variables
6. Methods
7. Using some classes

- ❖ Writting and using methods
- ❖ Value and Reference parameters
- ❖ Variable scope
- ❖ Supply methods as members of struct types
- ❖ Overloading method
- ❖ Delegate

Defining Method

```
<scope> <static> <returnType> MethodName(paramType> <paramName>,…)
{
        …
        return <returnValue>;
}
```

- Header
  - <scope>: public, private, protected, <blank>
  - <returnType>: *void* if need't return value
- Body
  - Contains the code of what the method does
  - Contains the return value if necessary (at most **one value**)
- All methods must be defined inside of a class

Calling methods

- Three ways to call a method:
  - Calling method in the same class: using a method name
  - Calling method in difference classes: using an object followed by the dot (**.**) operator and the method name
    - Example: Random r = new Random();
      int k = r.Next(100);
  - Calling the ***static*** *methods*: using a class name followed by a method name
    - Example:
      - **Console.Write("Hello");**
      - **Convert.ToInt32("10");**

Example 1: Method

```
namespace vd1 {
    class Program
    {
        static void Write()
        {
            Console.WriteLine("Text output from function.");
        }
        static void Main(string[] args)
        {
            Write();
            Console.ReadKey();
        }
    }
}
```


Example 2: Method

```
namespace vd2 {
  class CBinhPhuong {
   static void Main(string[] args ) {
      int result;
      for ( int counter = 1; counter <= 10; counter++ )
      {
        result = BinhPhuong( counter );
        Console.WriteLine("Binh phuong cua " + counter + " la " + result);
      }
   }
   static int BinhPhuong( int y ) {
        return y * y;
   }
  }
}
```


Value and Reference parameters (p.134)

- Value parameters (default)
  - Passed a value into a variable used by the method
  - *Any changes made to this variable in the function have no effect on the parameter specified in the function call*
- Reference parameters
  - Send a method the actual reference point
  - *Any changes made to this variable will be reflected in the value of the variable used as a parameter*
    - Use the **`ref`** keyword: means the argument must be initialized before calling the method
    - Use the **`out`** keyword: means the argument can be initialized in the method

Value parameters: Example

Output to the console is as follows:
myNumber = 5
val in method = 10
myNumber = 5

```
class ParameterTest
{
    static void ShowValue( int val )
    {
        val *= 2;
        Console.WriteLine("val in method = {0}", val);
    }
    static void Main( string[] args )
    {
        int myNumber = 5;
        Console.WriteLine("myNumber = {0}", myNumber);
        ShowValue(myNumber);
        Console.WriteLine("myNumber = {0}", myNumber);
    }
}
```


Reference parameters: Example

Output to the console is as follows:
myNumber = 5
val in method = 10
myNumber = 10

```
class ParameterTest
{
    static void ShowValue( ref int val )
    {
        val *= 2;
        Console.WriteLine("val in method = {0}", val);
    }
    static void Main( string[] args )
    {
        int myNumber = 5;
        Console.WriteLine("myNumber = {0}", myNumber);
        ShowValue(ref myNumber);
        Console.WriteLine("myNumber = {0}", myNumber);
    }
}
```


Reference parameters: Example (cont.)

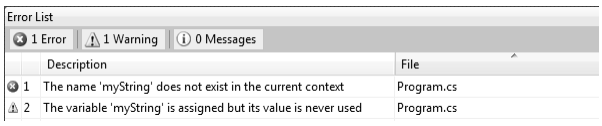
```
class ParameterTest
{
    static void SquareOut ( out int x )
    {
        x = 6;
        x = x * x;
        Console.WriteLine ("In method, x = {0}", x);
    }

    static void Main( string[] args )
    {
        int z;
        SquareOut(out z);
        Console.WriteLine ("After calling method, z = {0}", z);
    }
}
```


Read more...

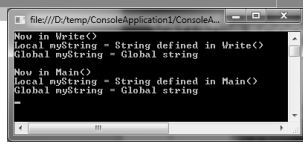
- **Variable scope: local variables and global variables, p.137**
  - **Note:**
    - Local variable should be initialized
    - Global variables needn't be initialized, if not initialized
      - All `int`, `float`, `double`… are set to `0`
      - All `bool` variables are set to `false`
      - All reference variables are set to `null`
- **The Main() function: p.143**
  - Some versions of Main method

Variable scope: Example 1

```
class Program
{
    static void Write()
    {
        Console.WriteLine("myString = {0}", myString);
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        Console.ReadKey();
    }
}
```



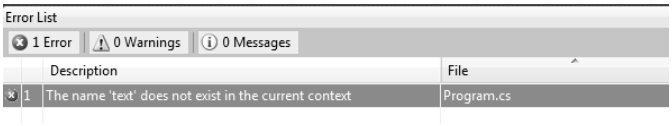
Variable scope: Example 2




```
class Program
{
    static string myString;        // global variables
    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Program.myString);
    }
    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Program.myString = "Global string";
        Write();
        Console.WriteLine("\nNow in Main()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Program.myString);
        Console.ReadKey();
    }
}
```

Variable scope: Example 3

```
int i;
for (i = 0; i < 10; i++)
{
        string text = "Line " + Convert.ToString(i);
        Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```



Struct method (p.146)

- Struct can contain functions as well as data
- Structs are value types, not reference types
  - But you can use the keyword `new` to declare
- You can define constructors for structs, but not allow to define a constructor with no parameters

Structs: Example

```
struct Dimensions {
    public double Length;
    public double Width;
    public Dimensions(double length, double width) {
        Length = length;
        Width = width;
    }
    public double Diagonal()  {
        return Math.Sqrt(Length*Length + Width*Width);
    }
}
static void Main(string[] args)       {
        Dimensions d;
        d = new Dimensions(5, 7);
        Console.WriteLine( d.Diagonal() );
        Console.ReadKey();
}
```


Overloading method (p.147)

- Overloading methods are multiple methods (in the same class) with the same name, but each having a different parameter list
- Example:

```
class Program  {
    static int MaxValue(int a, int b) {
        //...
    }
    static int MaxValue(int a, int b, int c) {
        //...
    }
    //...
}
```


Delegate (p.149)

- A *delegate* is a type that enables you to store references to functions
- Steps to working with delegate:
  - **Defining the delegate**
    - Like functions, but with no function body and using the *delegate* keyword
    - Example: public delegate void myDelegate(string mess);
  - **Writting methods which has <u>the same return type</u>** and **<u>parameter list</u> as the delegate**
  - **Creating the delegate objects** and **assigning methods to those delegate objects**
  - **Calling the methods via delegate objects**

Delegate: Example 1

- A simple to create and using delegate

```
// Defining the delegate
public delegate void SimpleDelegate();
class TestDelegate    {
    // Writing method
    public static void MyFunc()       {
        Console.WriteLine("I was called by delegate ...");
    }
    public static void Main()       {
        // Creating the delegate object
        SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);
        // Calling the methods
        simpleDelegate();
    }
}
```

Delegate: Example 2

- Write a program to perform some calculations using delegate

```
public class DelegateExample {
    public delegate int Calculate( int value1, int value2 );
    public static int add( int value1, int value2 ) {
        return value1 + value2;
    }
    public static int sub( int value1, int value2 ) {
        return value1 - value2;
    }
    static void Main( string[] args ) {
        Calculate cal;
        cal = new Calculate(add);
        Console.WriteLine("Adding two values: " + cal(10, 6));
        cal = new Calculate(sub);
        Console.WriteLine("Subtracting two values: " + sub(10,4) );
    }
}
```


Contents

Console class

- Writing to the console
  - Console.Write(…)
  - Console.WriteLine(…)
- To read a line of text from the console
  - Console.ReadLine(): return the input string
- Example:

```
string s = Console.ReadLine();
Console.WriteLine(s);
```


Math class

- Allows the user to perform common math calculations
- Some methods
  - **`Math.Abs(x)`**: absolute value of x
  - **`Math.Pow(x, y)`**: x raised to power y
  - **`Math.Sqrt(x)`**: square root of x
  - ...
- Example:

```
Console.WriteLine(Math.Sqrt(900.0));
```

- Constants
  - **`Math.PI`**
  - **`Math.E`**

Random class

- Within namespace **`System`**
- Methods
  - **`randomObject.Next`**()
    - Returns a number from 0 to **`Int32.MaxValue`**
  - **`randomObject.Next`** (int *x*)
    - Returns a value from 0 up to but not including x
  - **`randomObject.Next`** (*x*, *y*)
    - Returns a number between *x* and up to y, not including y
- Example:

```
Random rand = new Random();
int value;
value = rand.Next();            // random a number in [0; 2,147,483,647]
value = rand.Next( 6 );         // random a number in [0; 5]
value = rand.Next( 1, 7 );      // random a number in [1,6]
```


Anwser the questions

- How to create a Console project, Windows Form project?
- The structure of a project?
- The content of the Program.cs file?
- How to open Solution Explorer, Property Window, Toolbox?
- How to change properties for controls or forms?
- Which class is used for type conversion?