C#.NET Programming

Objectives

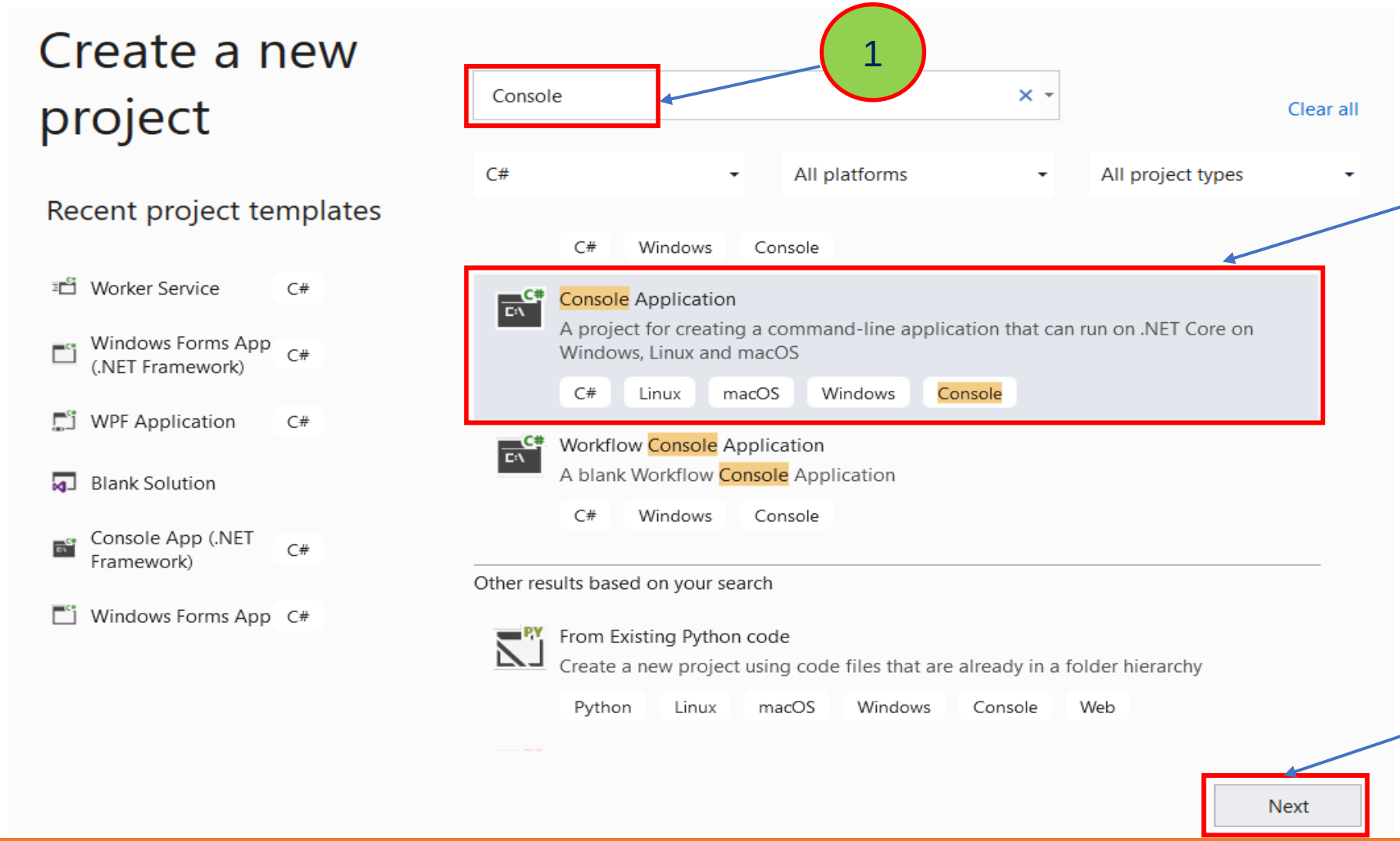
- Explain about input/output in C#
- Create C# Console Application using Visual Studio.NET
- Describe more new features of C# :
  - var and dynamic type
  - ref, out and params
  - Local Function and Static Local Function
  - String Interpolation
  - Null-Condition Operator
  - Ref locals and Ref returns
  - Discards and Pattern Matching
  - Numeric literal syntax
  - Tuples

Introduction to C#

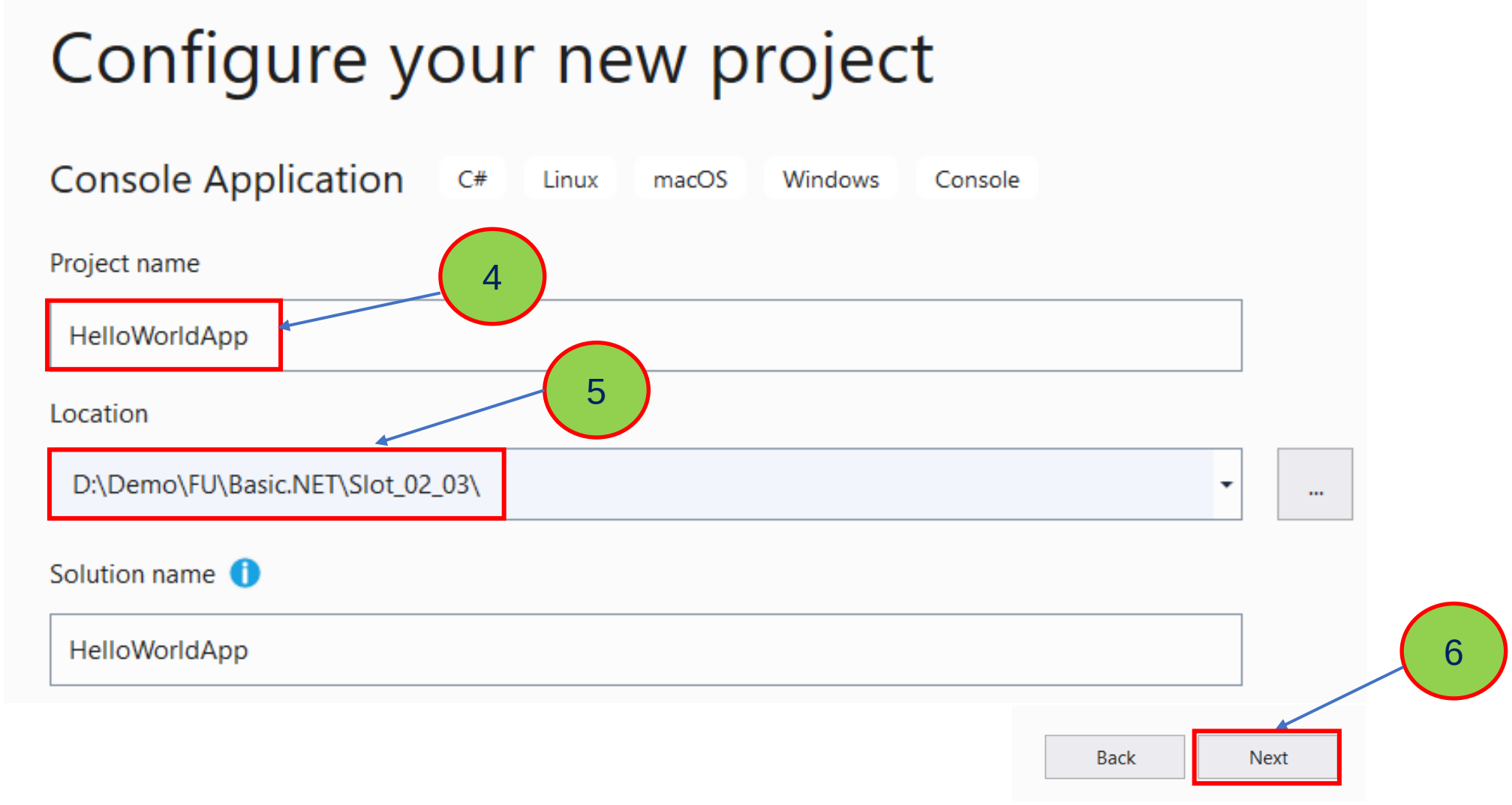
- C# is an object-oriented, component-oriented programming language
- C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components
- Several C# features help create robust and durable applications:
  - Garbage collection automatically reclaims memory occupied by unreachable unused objects
  - Exception handling provides a structured and extensible approach to error detection and recovery
  - Lambda expressions support functional programming techniques. Language Integrated Query (LINQ) syntax creates a common pattern for working with data from any source. Language support for asynchronous operations provides syntax for building distributed systems and so on

Create C# Console App using Visual Studio.NET Demonstration

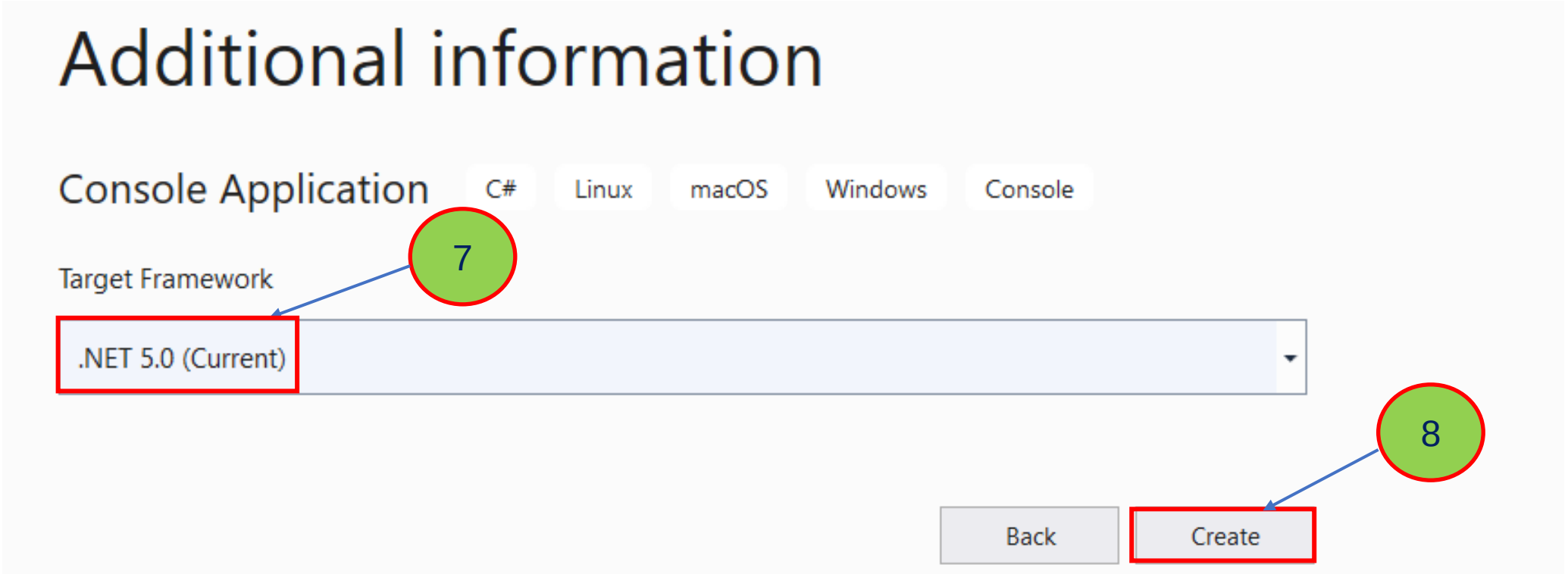
1. Open Visual Studio.NET , File | New | Project



2. Fill out **Project name**: HelloWorldApp and **Location** then click **Next**



3. Choose **Target Framework**: .NET 5.0(Current) then click **Create**



4. Write code for **Main** method then press **F5(**run Debugging**)** or **Ctrl+F5**(run without Debugging) to run application

```
1   using System;
2
3   namespace HelloWorldApp
4   {
5       class Program
6       {
7           static void Main(string[] args)
8           {
9               // Display a simple message to the user.
10              Console.WriteLine("***** My First C# App *****");
11              Console.WriteLine("Hello World!");
12              // Wait for Enter key to be pressed before shutting down.
13              Console.ReadLine();
14          }
15      }
16  }
17
```

Structure of a C# program

Referencing namespace

Namespace of the current class

Entry point: Main()

```
1   using System;
2
3   namespace HelloWorldApp
4   {
5       class Program
6       {
7           static void Main(string[] args)
8           {
9               // Display a simple message to the user.
10              Console.WriteLine("***** My First C# App *****");
11              Console.WriteLine("Hello World!");
12              // Wait for Enter key to be pressed before shutting down.
13              Console.ReadLine();
14          }
15      }
16  }
17
```

Namespaces in C#

- Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects
- The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features
- The members of a namespace can be namespaces, interfaces, structures, and delegates.

Namespaces in C#

- To define a namespace in C#, we will use the namespace keyword followed by the name of the namespace and curly braces containing the body of the namespace as follows:

```
namespace name_of_namespace
{
    // Namespace (Nested Namespaces)
    // Classes
    // Interfaces
    // Structures
    // Delegates
}
```

```
namespace MyNamespace
{
    // MyClass is the class in the namespace MyNamespace
    class MyClass
    {
        // class code
    }
}
```

Variations on the Main() Method

- By default, Visual Studio will generate a Main() method that has a void return value and an array of string types as the single input parameter
- To construct application's entry point using any of the following signatures:

```
static int Main(string[] args){
  // Must return a value before exiting!
  return 0;
}
// No return type, no parameters.
static void Main(){
}
// int return type, no parameters.
static int Main(){
  // Must return a value before exiting!
  return 0;
}
```

The Main() method can be **asynchronous**

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

Processing Command-Line Arguments

- Create C# Console App as the following and run it by **dotnet** CLI

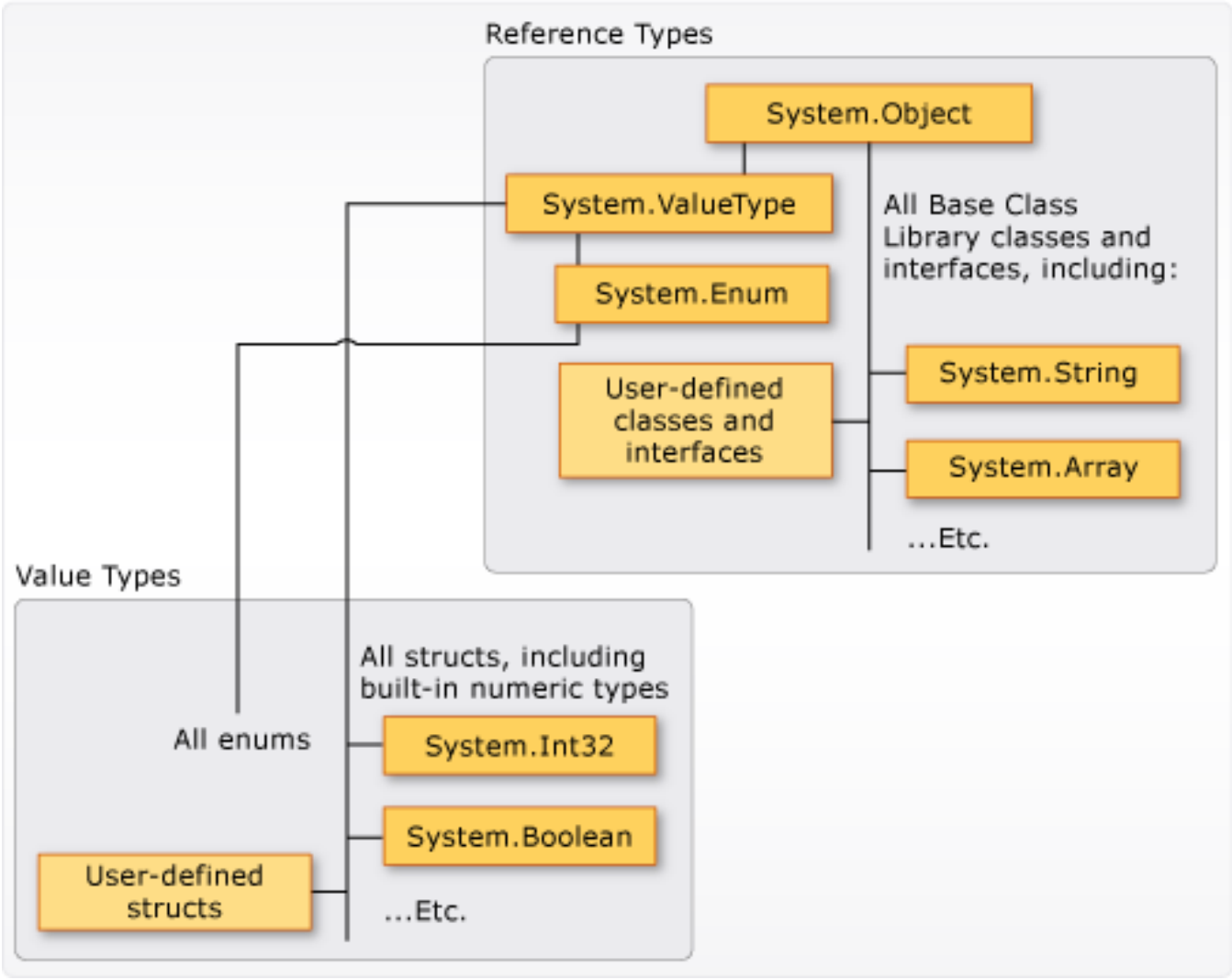
```
1  using System;
2  namespace Processing_Command_Line_Arguments {
3      class Program{
4          static void Main(string[] args){
5              string msg = "***** Welcome to C#.NET Programming *****";
6              Console.WriteLine("{0}",msg);
7              // Process any incoming args.
8              for (int i = 0; i < args.Length; i++){
9                  Console.WriteLine($"Arg: {args[i]}");
10             }
11             Console.ReadLine();
12         }
13     }
14 }
```

```
Command Prompt - dotnet  run Hello World !
D:\Demo\Processing_Command_Line_Arguments>dotnet run Hello World !
***** Welcome to C#.NET Programming *****
Arg: Hello
Arg: World
Arg: !
```

Value Types and Reference types

- **Value types** derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR(Common Language Runtime).
  - There are two categories of value types: **struct** and **enum**
- **Reference type**: A type that is defined as a class, delegate, array, or interface is a reference type.
  - At run time, when declare a variable of a reference type, the variable contains the value **null** until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using **new**

Value Types and Reference types

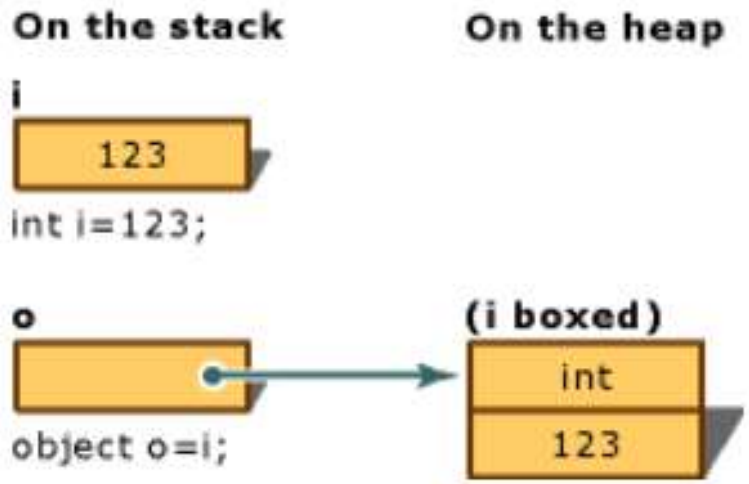


Boxing and Unboxing

- **Boxing** is the process of converting a value type to the type object or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap

```
static void Main(string[] args){
    int i = 123;
    // Boxing copies the value of i into object o.
    object o = i;
    // Change the value of i.
    i = 456;
    // The change in i doesn't affect the value stored in o.
    Console.WriteLine("The value-type value = {0}", i);
    Console.WriteLine("The object-type value = {0}", o);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoBoxing_UnBoxing
The value-type value = 456
The object-type value = 123
```

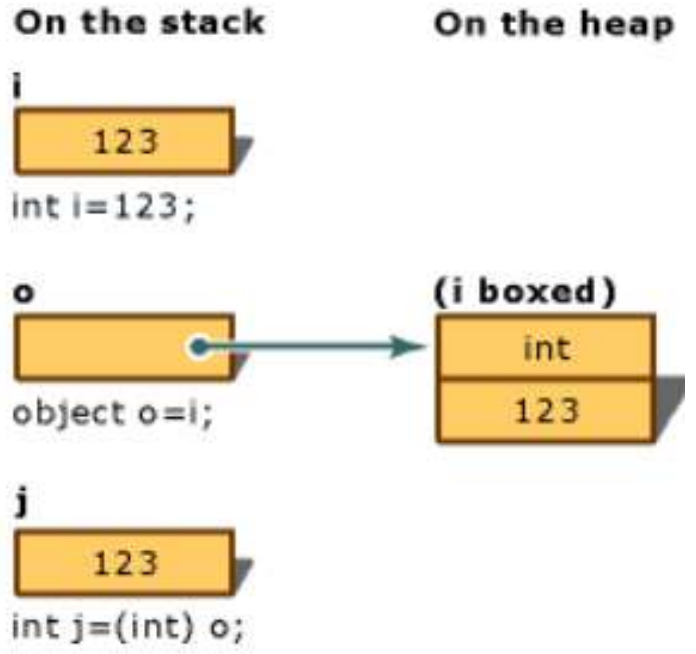


Boxing and Unboxing

- **Unboxing** is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface
- An unboxing operation consists of: **Checking** the object instance to make sure that it is a boxed value of the given value type and **Copying** the value from the instance into the value-type variable

```
static void Main(string[] args){
    int i = 123;      // a value type
    object o = i;     // boxing
    int j = (int)o;   // unboxing
    Console.WriteLine("i = {0}, o = {1}, j = {2}",i,o,j);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoBoxing_UnBoxing\bin\Debug\net5.0\
i = 123, o = 123, j = 123
```



If we change code line:

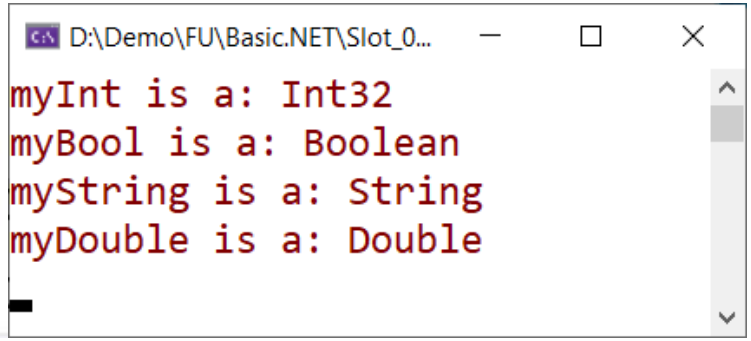
int j = (int)o to int j = (short)o

what happens?

var keyword

- The var keyword can be used in place of specifying a specific data type (such as int, bool, or string) and the compiler will automatically infer the underlying data type based on the initial value used to initialize the local data point.

```
static void Main(string[] args)
{
    var myInt = 0;
    var myBool = true;
    var myString = "Hello World !";
    var myDouble = 0.5;
    // Print out the underlying type.
    Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
    Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
    Console.WriteLine("myString is a: {0}", myString.GetType().Name);
    Console.WriteLine("myDouble is a: {0}", myDouble.GetType().Name);
    Console.ReadLine();
}
```



var keyword

- The following restrictions apply to implicitly-typed variable declarations:
  - **var** can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function
  - **var** cannot be used on fields at class scope
  - Variables declared by using var cannot be used in the initialization expression
  - Multiple implicitly-typed variables cannot be initialized in the same statement

dynamic type

- The dynamic type is a static type, the compiler does not check the type of the dynamic type variable at compile time, instead of this, the compiler gets the type at the run time
- In most of the cases, the dynamic type behaves like object types
- The dynamic type changes its type at the run time based on the value present on the right-hand side
- To get the actual type of the dynamic variable at runtime by using GetType() method
- Can pass a dynamic type parameter in the method so that the method can accept any type of parameter at run time

dynamic type

```
static void Main(string[] args)
{
    dynamic myValue = 0;
    Console.WriteLine("myInt is a: {0}", myValue.GetType().Name);
    myValue = true;
    Console.WriteLine("myBool is a: {0}", myValue.GetType().Name);
    myValue = "Hello World !";
    Console.WriteLine("myString is a: {0}", myValue.GetType().Name);
    myValue = 0.5;
    Console.WriteLine("myDouble is a: {0}", myValue.GetType().Name);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\...
myInt is a: Int32
myBool is a: Boolean
myString is a: String
myDouble is a: Double
```

String Interpolation

- The string interpolation feature is built on top of the composite formatting feature and provides a more readable and convenient syntax to include formatted expression results in a result string.
- To identify a string literal as an interpolated string, prepend it with the $ symbol

$ " <text> { <interpolation-expression> <optional-comma-field-width> <optional-colon-format> } <text> {... } "

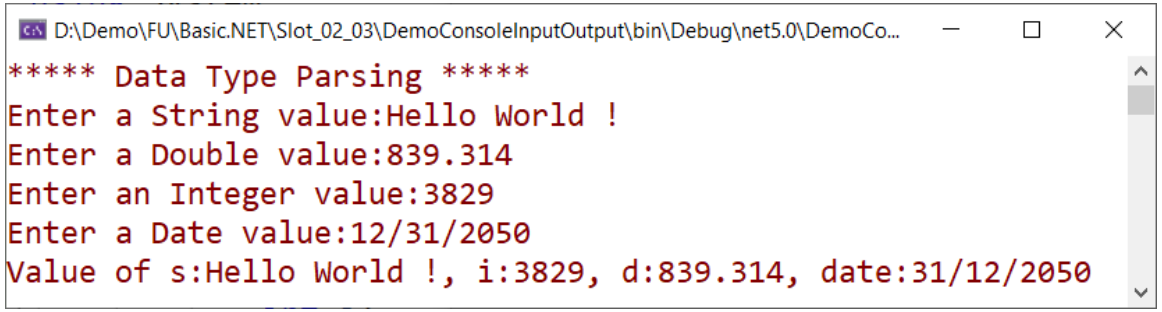
```
static void Main(string[] args) {
    double salary = 200.234;
    string name = "Soren";
    // Using curly-bracket syntax.
    string str1 = string.Format("Name{0,6}, Salary{1,7:N2}", name,salary);
    Console.WriteLine(str1);
    // Using string interpolation
    string str2 = $"Name{name,7},Salary{salary,8:N2}";
    Console.WriteLine(str2);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoStringIn
Name Soren, Salary 200.23
Name   Soren,Salary   200.23
```

The Console Class

- The Console type defines a set of methods to capture input and output, all of which are static, therefore, called by prefixing the name of the class **Console** to the method name

```
static void Main(string[] args){
    double d;
    int i;
    string s;
    DateTime date;
    Console.WriteLine("***** Data Type Parsing *****");
    Console.Write("Enter a String value:");
    s = Console.ReadLine();
    Console.Write("Enter a Double value:");
    d = double.Parse(Console.ReadLine());
    Console.Write("Enter an Integer value:");
    i = int.Parse(Console.ReadLine());
    Console.Write("Enter a Date value:");
    date = DateTime.Parse(Console.ReadLine());
    Console.WriteLine($"Value of s:{s}, i:{i}, d:{d},date:{date:dd/MM/yyyy}");
    Console.ReadLine();
}
```



Numeric Literal Syntax

- When assigning large numbers to a numeric variable, there are more digits we can use underscore (_) as a digit separator (for integer, long, decimal, double data, or hex types)
- C# provides also a new literal for binary values allows for binary numbers to start with an underscore

```
static void Main(string[] args){
    Console.WriteLine("**** Use Digit Separators *****");
    Console.Write("Integer:");
    Console.WriteLine(123_456);
    Console.Write("Double:");
    Console.WriteLine(123_456.12);
    Console.Write("Hex:");
    Console.WriteLine(0x_00_00_FF);
    Console.WriteLine("***** Use Binary Literals *****");
    Console.WriteLine("Sixteen: {0}", 0b_0001_0000);
    Console.WriteLine("Thirty Two: {0}", 0b_0010_0000);
    Console.WriteLine("Sixty Four: {0}", 0b_0100_0000);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\...
**** Use Digit Separators *****
Integer:123456
Double:123456.12
Hex:255
***** Use Binary Literals *****
Sixteen: 16
Thirty Two: 32
Sixty Four: 64
```

Passing Parameters with ref, out and params

- 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** with the intent of changing the value, use the **ref**, or **out** keyword
- The **ref** keyword makes the formal parameter an alias for the argument, which must be a variable
- An argument that is passed to a **ref** parameter **must be initialized** before it is passed

Passing Parameters with ref, out and params

- Variables passed as **out** arguments **do not have to be initialized** before being passed in a method call. However, the called method is required to assign a value before the method returns

```
static void MyMethod(int a, ref int b, out int c) {
    a = 3;
    b = 4;
    c = 5;
}
static void Main(string[] args)
{
    int x = 1, y = 2, z;
    MyMethod(x, ref y, out z);
    Console.WriteLine($"x:{x}, y:{y}, z:{z}");
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoPassing Parameters\
x:1, y:4, z:5
```

Passing Parameters with ref, out and params

- The params keyword allows you to pass into a method a variable number of identically typed parameters (or classes related by inheritance) as a single logical parameter
- The arguments marked with the **params** keyword can be processed if the caller sends in a **strongly typed array** or **a comma-delimited list of items**
- The parameter type must be a **single-dimensional** array
- No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration

Passing Parameters with ref, out and params

```
static void SumArray(out int s,params int[] list){
    int i;
    s = 0;
    for (i = 0; i < list.Length; i++){
        s += list[i];
    }
}
static void Main(string[] args){
    int s;
    SumArray(out s,1, 2, 3, 4);
    Console.WriteLine($"s={s}");
    int[] myIntArray = { 5, 6, 7, 8, 9 };
    SumArray(out s,myIntArray);
    Console.WriteLine($"s={s}");
    SumArray(out s);
    Console.WriteLine($"s={s}");
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoUseParams\
s=10
s=35
s=0
```

Ref locals and Ref returns

- A **reference return value(ref returns)** allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a **ref local**.
- A method that returns a reference return value must satisfy the following two conditions:
  - The method signature includes the ref keyword in front of the return type
  - Each return statement in the method body includes the ref keyword in front of the name of the returned instance

Ref locals and Ref returns

```
class Program
{
    static int[] numbers = { 1, 2, 3, 4, 5 };
    //-----------------------------------------
    static ref int FindNumber(int target){
        bool flag = true;
        ref int value = ref numbers[0];
        for (int ctr = 0; ctr < numbers.Length && flag == true; ctr++) {
            if (numbers[ctr] == target) {
                value = ref numbers[ctr];
                flag = false;
            }
        }
        return ref value;
    }
    //-----------------------------------------
    static void Main(string[] args){
        Console.Write("Original sequence:");
        Console.WriteLine(string.Join(" ", numbers));
        ref int value = ref FindNumber(3);
        value *= 2;
        Console.Write("New sequence:     ");
        Console.WriteLine(string.Join(" ", numbers));
        Console.ReadLine();
    }//end Main
}//end Program
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoRefReturn_And_RefLocal\
Original sequence:1 2 3 4 5
New sequence:     1 2 6 4 5
```

Local Function and Static Local Function

- The local function allows declaring a method inside the body of an already defined method. Or in other words, we can say that a local function is a private function of a function whose scope is limited to that function in which it is created
- The type of local function is similar to the type of function in which it is defined. We can only call the local function from their container members
- Local functions can be defined anywhere inside a method: the top, the bottom, or middle
- Local function can access the local variables that are defined inside the container method including method parameters

Local Function and Static Local Function

- Local function allows to pass out/ref parameters, using generic and using params keyword
- Cannot declare a local function in the expression-bodied member
- Not allowed to use any member access modifiers in the local function definition, including private keyword because they are by default private
- Overloading is not allowed for local functions
- Using the **static** modifier to declare a **local function** as **a static local function**, ensure the **local function** doesn't reference any variables from the enclosing scope

Local Function and Static Local Function

```
static void Main(string[] args)
{
    // Variables of main method
    int x = 1;
    int y = 2;
    // Local Function
    void AddValue(int a, int b)
    {
        Console.WriteLine("Value of a is: " + a);
        Console.WriteLine("Value of b is: " + b);
        Console.WriteLine("Value of x is: " + x);
        Console.WriteLine("Value of y is: " + y);
        Console.WriteLine("Sum: {0}", a + b + x + y);
        Console.WriteLine();
    }
    // Calling Local function
    AddValue(3, 4);
    AddValue(5, 6);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoLocalFunction\
Value of a is: 3
Value of b is: 4
Value of x is: 1
Value of y is: 2
Sum: 10

Value of a is: 5
Value of b is: 6
Value of x is: 1
Value of y is: 2
Sum: 14
```

Local Function and Static Local Function

```
static void Main(string[] args){
    // AreaofCircle is the local function of the main function
    void AreaOfCircle(double a) {
        double ar;
        Console.WriteLine("Radius of the circle: " + a);
        ar = 3.14 * a * a;
        Console.WriteLine("Area of circle: " + ar);
        // Calling static local function
        Circumference(a);
        // Circumference is the Static local function
        static void Circumference(double radii){
            double cr;
            cr = 2 * 3.14 * radii;
            Console.WriteLine($"Circumference of the circle is:{cr:N2}");
        }
    }//end AreaOfCircle
    // Calling function
    AreaOfCircle(10);
    Console.ReadLine();
}//end Main
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoStaticLocalFunction\bin\[
Radius of the circle: 10
Area of circle: 314
Circumference of the circle is:62.80
```

Tuples

- The tuples feature provides concise syntax to group multiple data elements in a lightweight data structure which gives us the easiest way to represent a data set that has multiple values that may/may not be related to each other
- Each property in a tuple can be assigned a specific name (just like variables), greatly enhancing the usability
- There are two important considerations for tuples: the fields are not validated and cannot define our methods

```
static void Main(string[] args){
    //Create tuples
    (string MyString, int MyNumber) MyValues= ("Hello world !", 2050);
    Console.WriteLine($"MyString: {MyValues.MyString}");
    Console.WriteLine($"MyNumber: {MyValues.MyNumber}");
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoTuple
MyString: Hello world !
MyNumber: 2050
```

Tuples

```
static (int sum, int count) MyMethod(int[] values){
    //declare a tuple
    var r = (sum: 0, count: 0);
    for(int i=0;i<values.Length;i++){
        if(IsEvenNumber(values[i])){
            r.sum +=values[i];
            r.count++;
        }
    }
    return r;
    bool IsEvenNumber(int n){
        return n % 2 == 0;
    }
}
static void Main(string[] args){
    int[] numbers = { 2, 1, 5, 6, 3, 4, 7, 8, 10, 9};
    var (sum, count) = MyMethod(numbers);
    Console.WriteLine($"Sum: {sum}, Count: {count}");
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoTuple\bin\Debug\net5.0\DemoTuple.exe
Sum: 30, Count: 5
```

Discards

- C# allows discard the returned value which is not required. Underscore (_) character is used for discarding the parameter
- Discards are equivalent to unassigned variables, they don't have a value.
- Discards can reduce memory allocations.
- Discards make the intent of our code clear. They enhance its readability and maintainability.

Discards

- Using discard with **out** parameter

```
static void Main(string[] args)
{
    string[] stringArray = {"12","Hello","3.14","20"};
    for(int i = 0;i< stringArray.Length; i++)
    {
        if (int.TryParse(stringArray[i], out _))
            Console.WriteLine($"{stringArray[i]}: valid");
        else
            Console.WriteLine($"{stringArray[i]}: invalid");
    }
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoDiscard\
12: valid
Hello: invalid
3.14: invalid
20: valid
```

Discards

- Using discard with **Tuples**

```
static (string first, string middle, string last) SplitNames(string fullName)
{
    string[] strArray = fullName.Split(' ');
    return (strArray[0], strArray[1], strArray[2]);
}
static void Main(string[] args)
{
    var (first, _, last) = SplitNames("Philip F Japikse");
    Console.WriteLine($"{first}:{last}");
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoDiscard
Philip:Japikse
```

Pattern Matching

- Pattern matching allows the developer to match a value (or an object) against some patterns to select a branch/block of the code through the use of **is patterns** and **case patterns**
  - The **is** pattern
    - The is pattern allows we to check whether an *input* variable is of a certain type, and then assign it to a new variable named *count*.

```
if (input is int count && count > 0)
```

    - This pattern can also be used to check if an *input* variable is null:

```
if (input is null)
```

Pattern Matching

- The **case** pattern
  - The switch statement cases also support patterns. These patterns can include a type check, plus additional conditions:

```
switch (i)
{
    case int n when n > 100:
        ...
    case Car c: //c is instance of Car
        ...
    case null:
        ...
   case var j when (j.Equals(10)):
        ...
    default:
        ...
}
```

Pattern Matching

```
static void Main(string[] args)
{
    Console.Write("Input data:");
    int.TryParse(Console.ReadLine(), out int n);
    if (n is int count && count > 0)
    {
        Console.WriteLine(new string('*', count));
    }
    else
    {
        Console.WriteLine("Data invalid.");
    }
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoPatternMatching\
Input data:10
**********
```

```
static void Main(string[] args)
{
    Console.Write("Input data:");
    int.TryParse(Console.ReadLine(), out int n);
    switch(n)
    {
        case int count when count > 0:
            Console.WriteLine(new string('*', count));
            break;
        default:
            Console.WriteLine("Data invalid.");
            break;
    }
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\Slot_02_03\DemoPatternMatching\
Input data:
Data invalid.
```

Null-Condition Operator

- Used to test for null before performing **a** member access (?.) or index (?[]) operation. These operators help we write less code to handle null checks
  - If **a** evaluates to null, the result of **a**?.x or **a**?[x] is null.
  - If a evaluates to non-null, the result of **a**?.x or **a**?[x] is the same as the result of **a**.x or **a**[x], respectively.

```
static void Main(string[] args){
    int[] array1 = null;
    Console.WriteLine($"{ array1 ?.Length.ToString()?? "Array is empty."}");
    array1 = new int[] { 2, 1, 3 };
    dynamic[] array2 = { array1, null };
    var s = array2? [0] ?.Length?? "Array is empty.";
    Console.WriteLine($"{s}");
    s = array2? [1] ?.Length?? "Array is empty.";
    Console.WriteLine($"{s}");
}
```

Microsoft Visual Studio Debug Console

```
Array is empty.
3
Array is empty.
```

Nullable value types

- A nullable value type T? represents all values of its underlying value type T and an additional null value

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

```
static void Main(string[] args){
    int? a = null;
    if (a is null){
        Console.WriteLine("a does not have a value");
        a = 2050;
    }
    if(a is int valueOfA){
        Console.WriteLine($"a is {valueOfA}");
    }
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic...
a does not have a value
a is 2050
```

Nullable reference types

- Nullable reference types follow many of the same rules as nullable value types. Nullable reference types can be null, but still must be assigned something before first use
- Nullable reference types use the same symbol (?) to indicate that they are nullable

```
static void PrintFullName(string first, string? middle, string last){
    Console.WriteLine(middle?.Length);
    Console.WriteLine(first + middle + last);
}
static void Main(string[] args) {
    string firstName = "Mike ";
    string? middleName = null;
    string lastName = " John";
    PrintFullName(firstName, middleName, lastName);
    Console.ReadLine();
}
```

```
D:\Demo\FU\Basic.NET\
Mike  John
```

Summary

- Explain about input/output in C#
- Demo create C# Console Application using Visual Studio.NET
- Describe more new features of C# :
  - var and dynamic type
  - ref, out and params
  - Local Function and Static Local Function
  - String Interpolation and Namespaces
  - Null-Condition Operator and Nullable reference types
  - Ref locals and Ref returns
  - Discards and Pattern Matching
  - Numeric literal syntax
  - Tuples