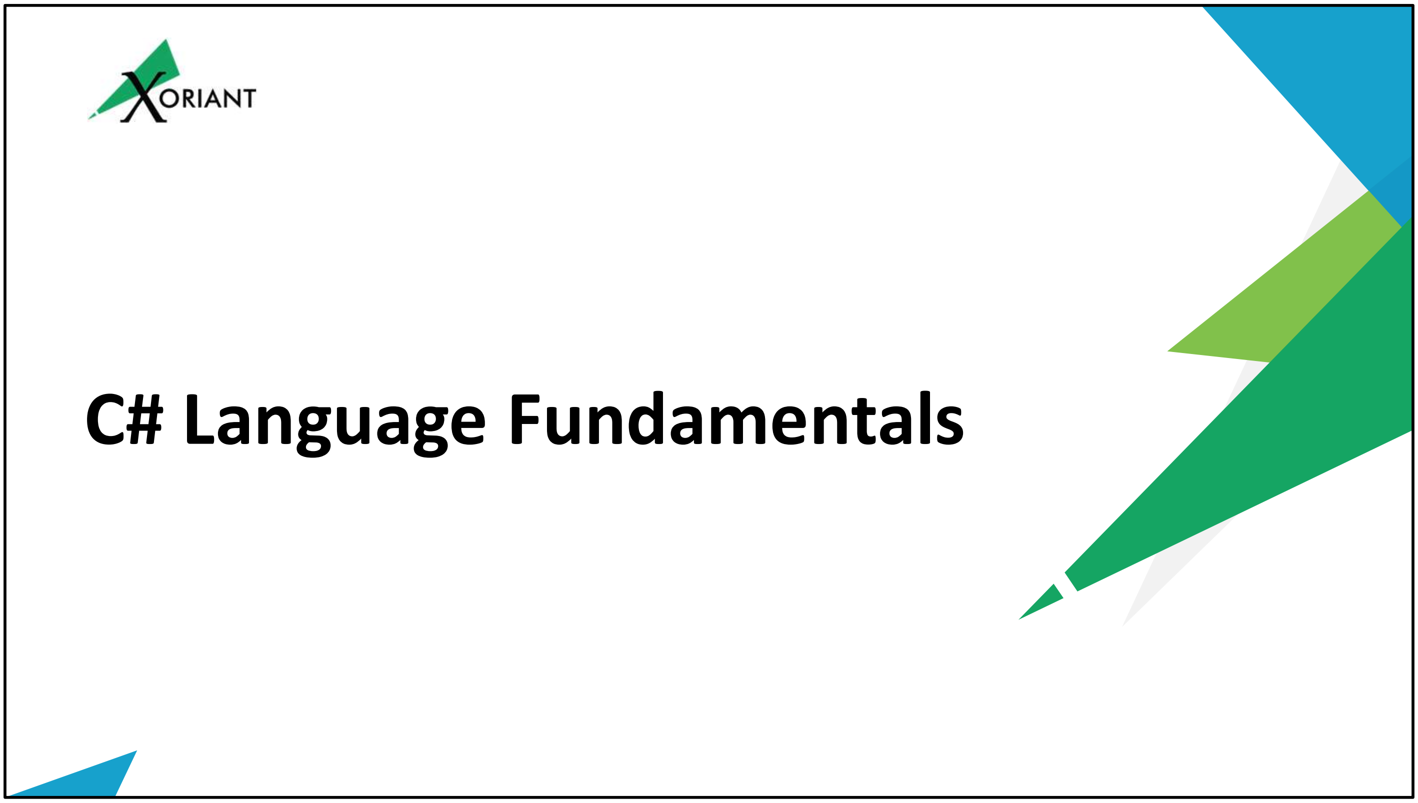
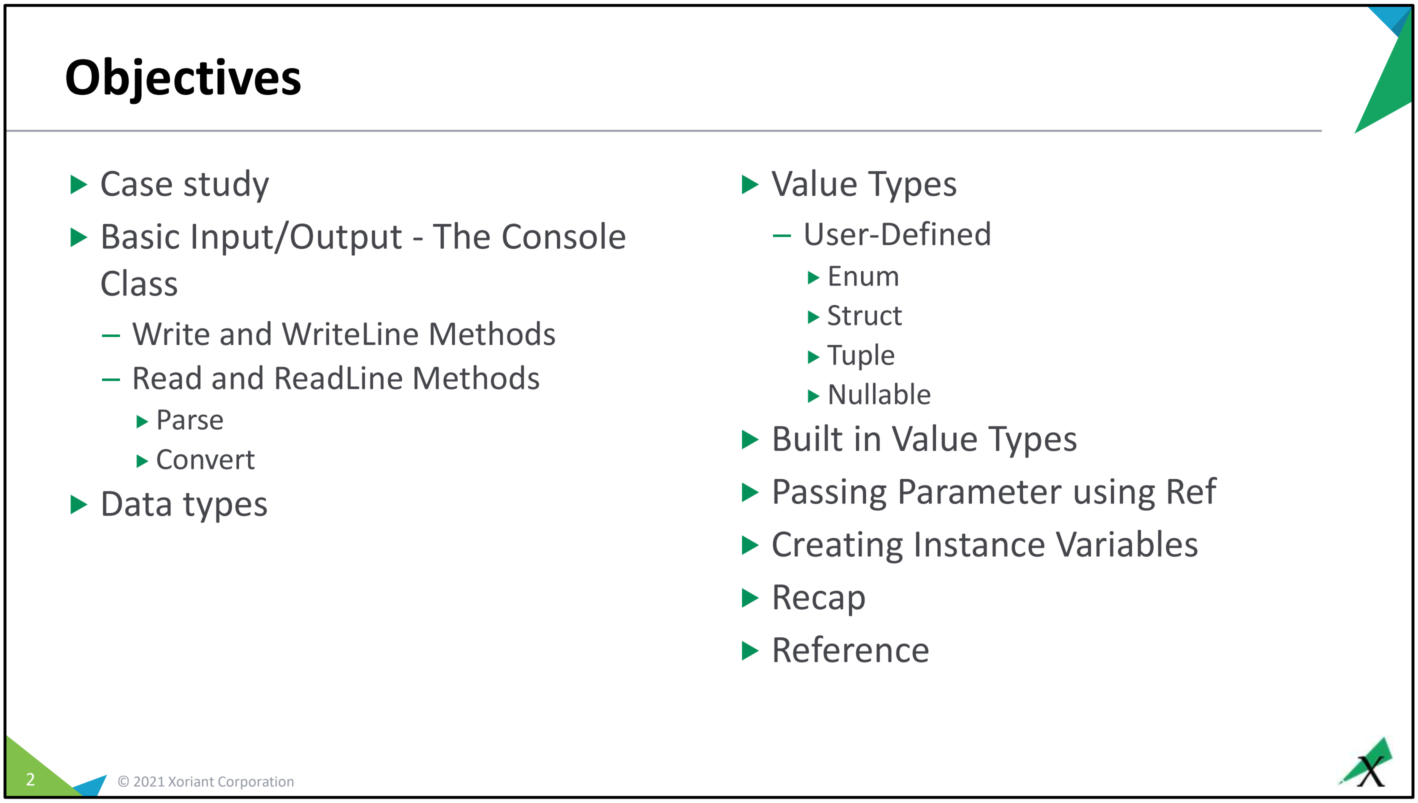


XORIANT

# C# Language Fundamentals



# Objectives

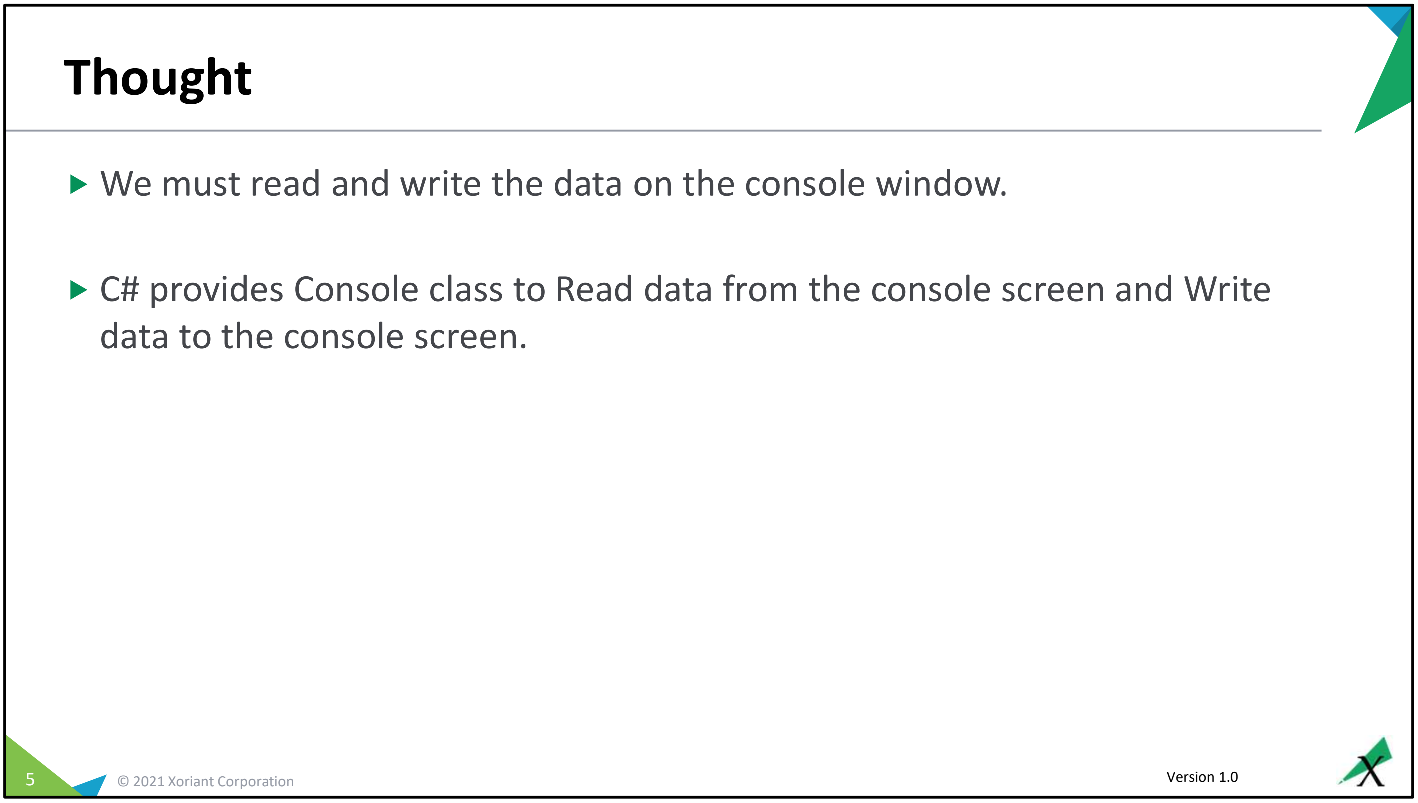
- Case study
- Basic Input/Output - The Console Class
  - Write and WriteLine Methods
  - Read and ReadLine Methods
    - Parse
    - Convert
- Data types
- Value Types
  - User-Defined
    - Enum
    - Struct
    - Tuple
    - Nullable
- Built in Value Types
- Passing Parameter using Ref
- Creating Instance Variables
- Recap
- Reference

# TIME FOR CASE STUDY

# Case Study-1

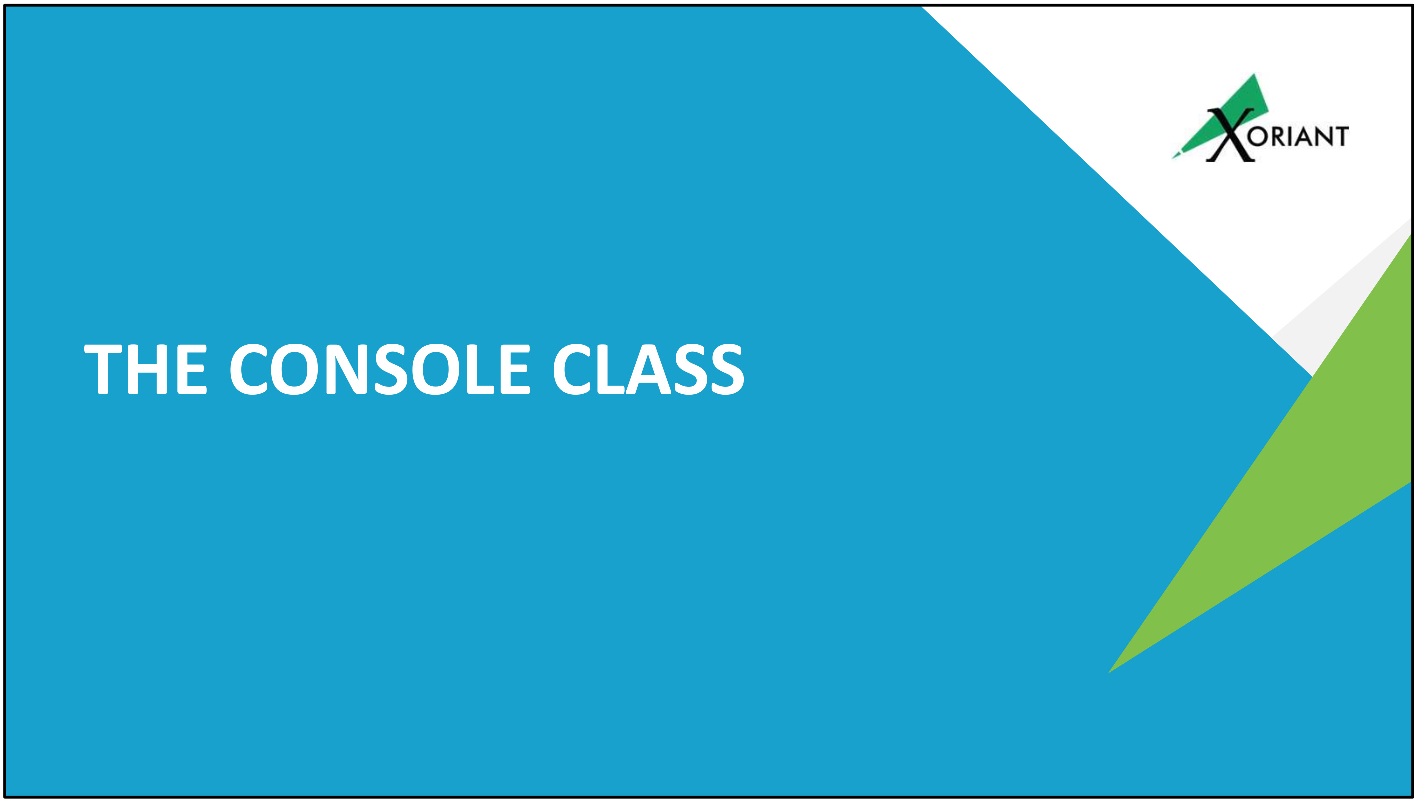
- Shopon is one stop mobile shopping portal which deals with mobiles and its accessories.
- As a portal **admin**, I should be allowed to add following information: pid, productName, price

Thought

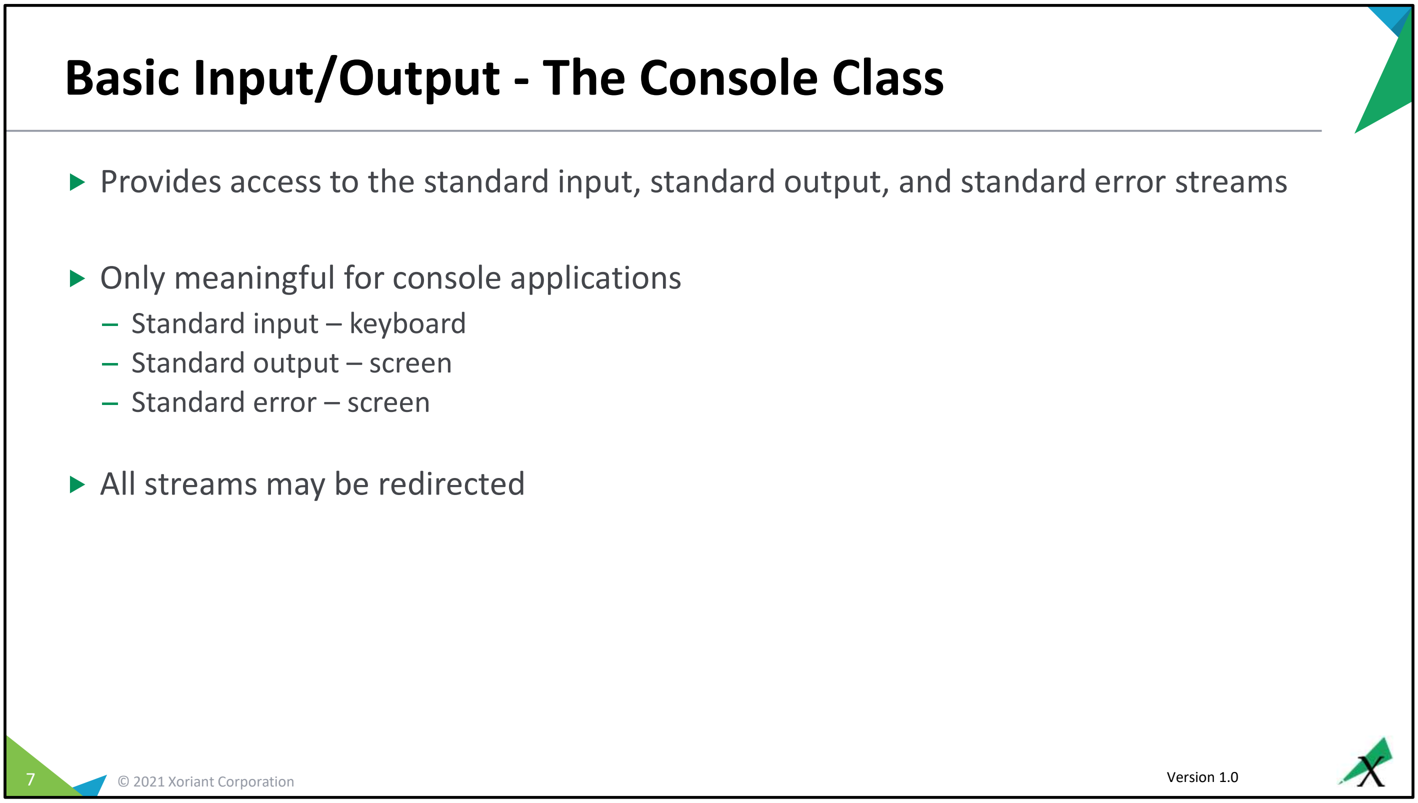


# Thought

- We must read and write the data on the console window.
- C# provides Console class to Read data from the console screen and Write data to the console screen.



# THE CONSOLE CLASS



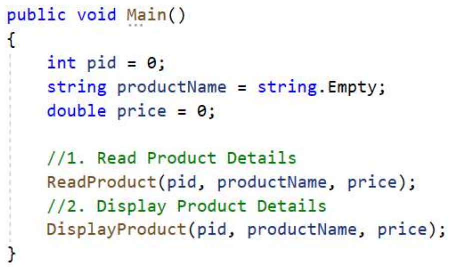
# Basic Input/Output - The Console Class

- Provides access to the standard input, standard output, and standard error streams
- Only meaningful for console applications
  - Standard input – keyboard
  - Standard output – screen
  - Standard error – screen
- All streams may be redirected

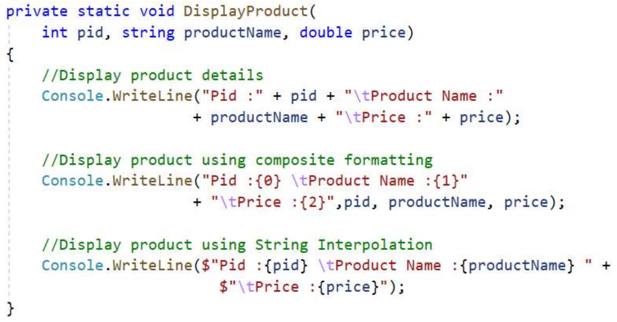
Version 1.0

https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0

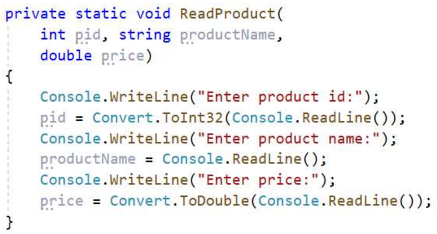
# Solution



```
public void Main()
{
    int pid = 0;
    string productName = string.Empty;
    double price = 0;

    //1. Read Product Details
    ReadProduct(pid, productName, price);
    //2. Display Product Details
    DisplayProduct(pid, productName, price);
}
```

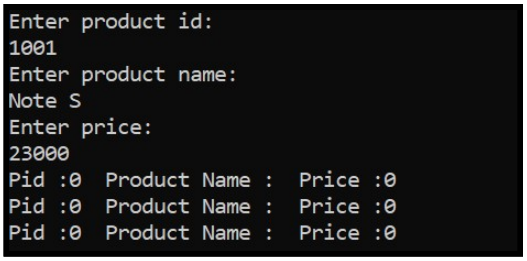


```
private static void DisplayProduct(
    int pid, string productName, double price)
{
    //Display product details
    Console.WriteLine("Pid :" + pid + "\tProduct Name :"
                    + productName + "\tPrice :" + price);

    //Display product using composite formatting
    Console.WriteLine("Pid :{0} \tProduct Name :{1}"
                    + "\tPrice :{2}",pid, productName, price);

    //Display product using String Interpolation
    Console.WriteLine($"Pid :{pid} \tProduct Name :{productName} " +
                      $"\tPrice :{price}");
}
```



```
private static void ReadProduct(
    int pid, string productName,
    double price)
{
    Console.WriteLine("Enter product id:");
    pid = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter product name:");
    productName = Console.ReadLine();
    Console.WriteLine("Enter price:");
    price = Convert.ToDouble(Console.ReadLine());
}
```

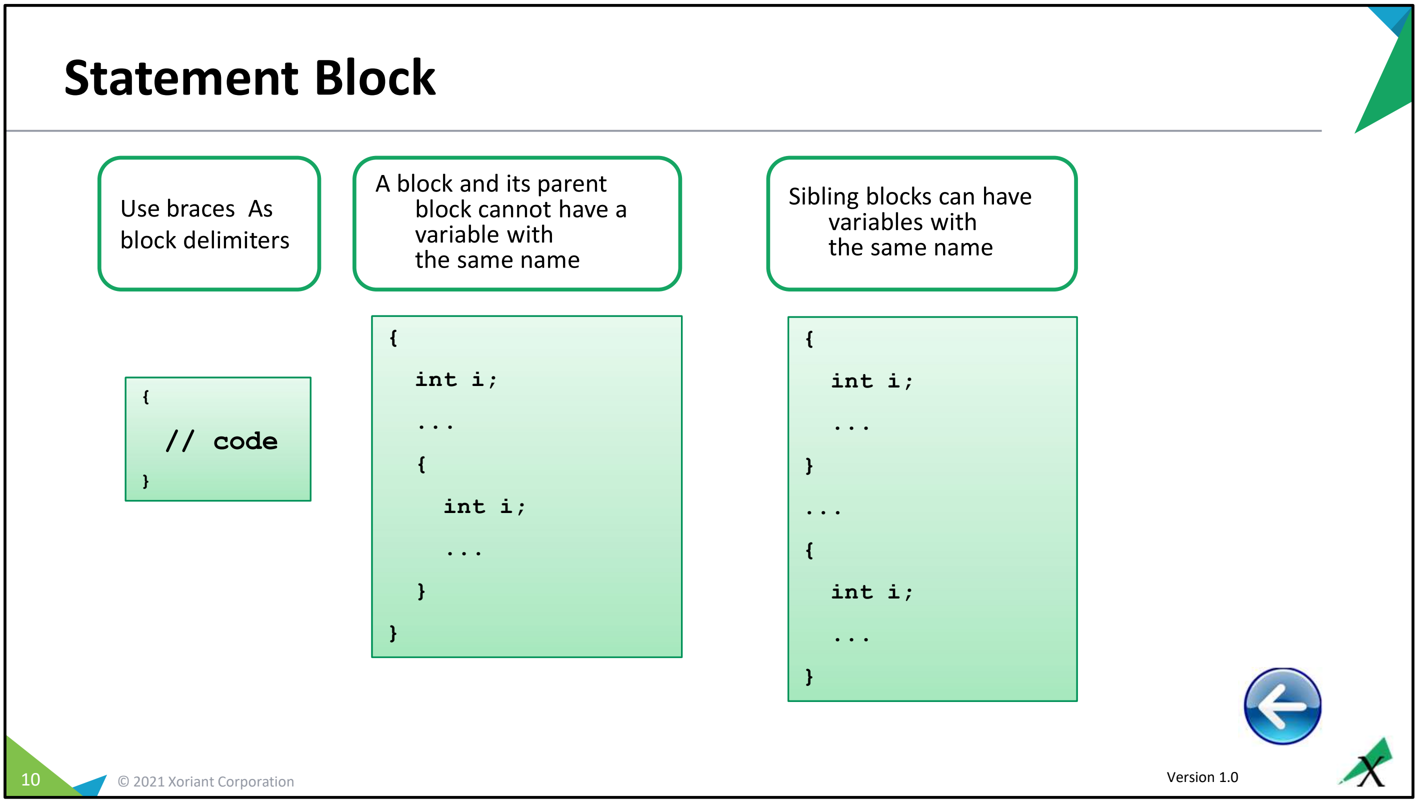
# Output of the Solution 1



```
Enter product id:
1001
Enter product name:
Note S
Enter price:
23000
Pid :0  Product Name :  Price :0
Pid :0  Product Name :  Price :0
Pid :0  Product Name :  Price :0
```

We are not getting the desired output, why?

- Local Variables: are those which will loses its value between calls.
- They can be accessed only with the method.
- They should follow **camelcase** while naming. They cannot have **Underscores**



# Statement Block

Use braces As block delimiters

```
{
  // code
}
```

A block and its parent block cannot have a variable with the same name

```
{
  int i;
  ...
  {
    int i;
    ...
  }
}
```

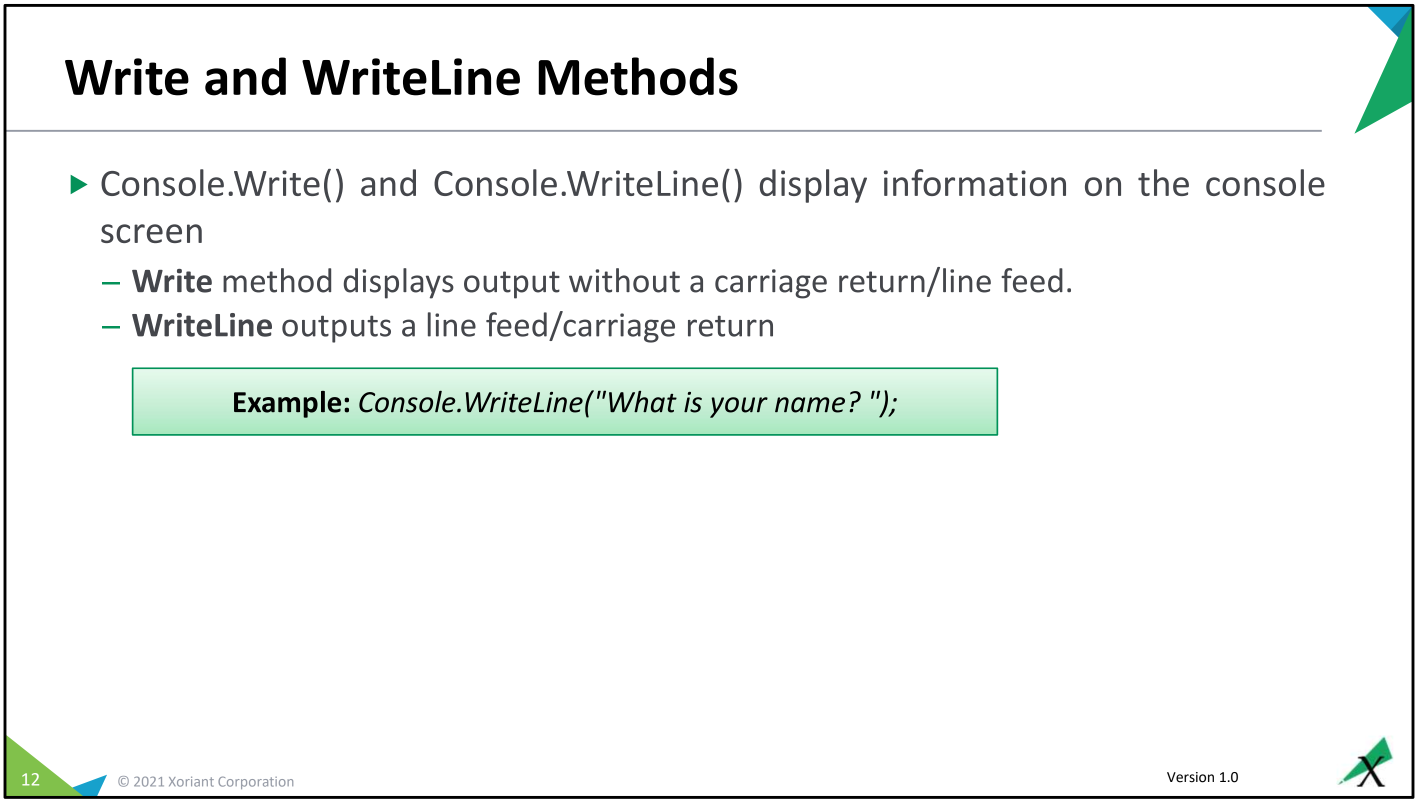
Sibling blocks can have variables with the same name

```
{
  int i;
  ...
}
...
{
  int i;
  ...
}
```

# Read and ReadLine Methods

- Console.Read() and Console.ReadLine() read user input
  - **Read** reads the next character
  - **ReadLine** reads the entire input line
  - Example: *string name = Console.ReadLine( );*

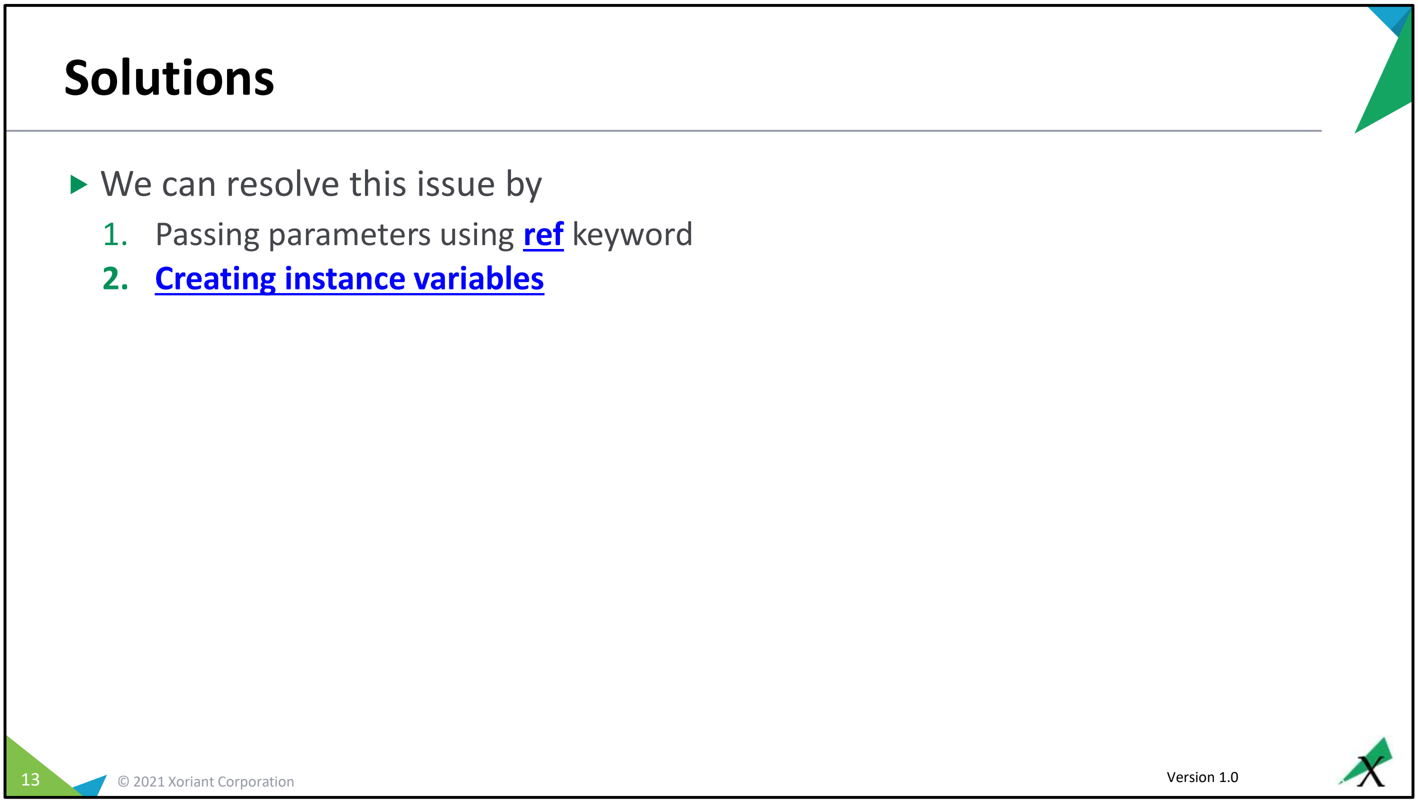
- To change the string data to other data types, we can use  or



# Write and WriteLine Methods

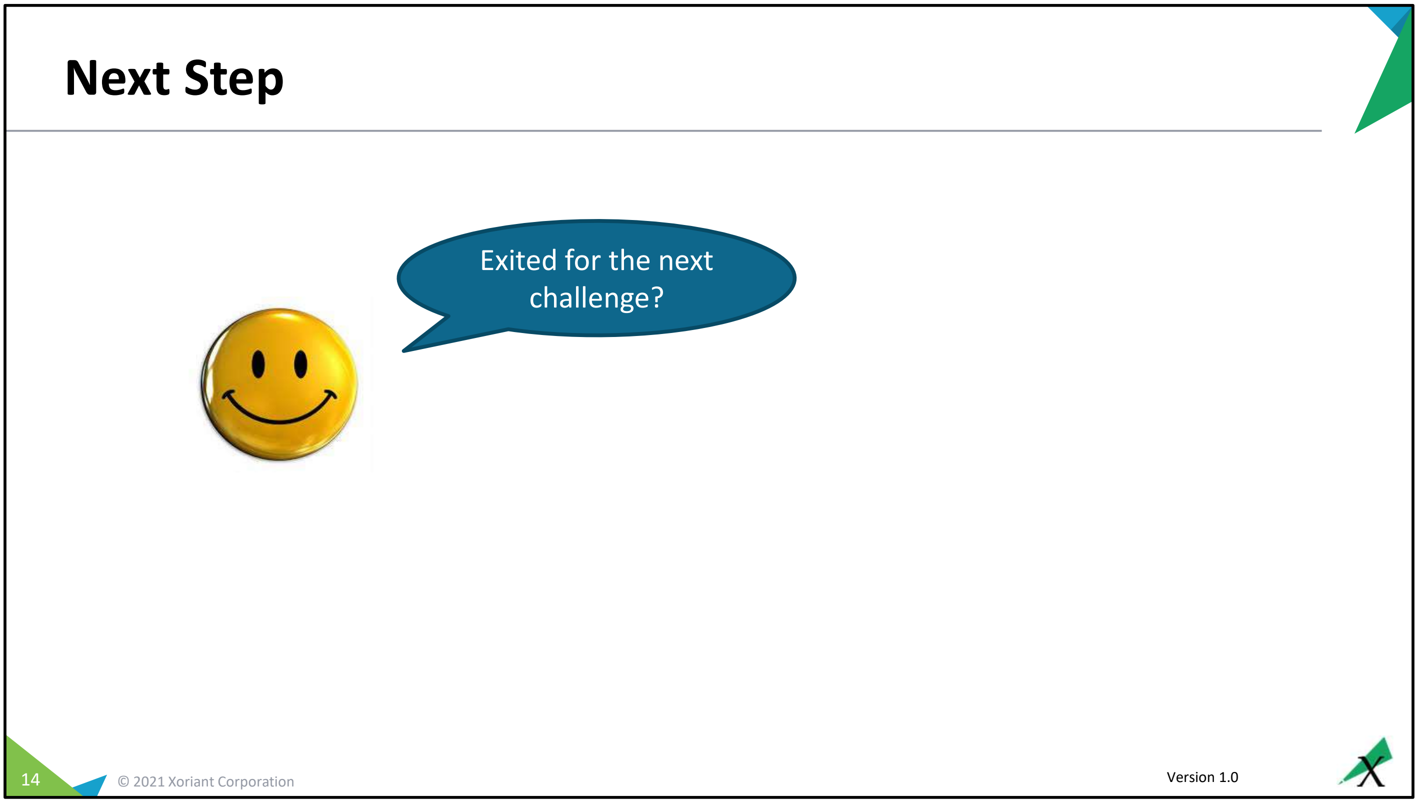
- Console.Write() and Console.WriteLine() display information on the console screen
  - **Write** method displays output without a carriage return/line feed.
  - **WriteLine** outputs a line feed/carriage return

**Example:** *Console.WriteLine("What is your name? ");*



## Solutions

- We can resolve this issue by
  1. Passing parameters using **ref** keyword
  2. **Creating instance variables**



# Next Step

Exited for the next challenge?

# Parse

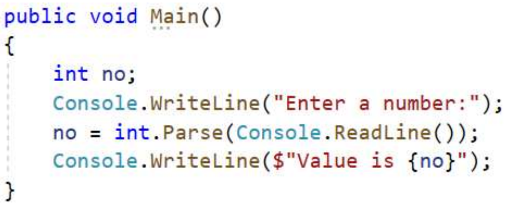
- We convert a string to a number by calling the **Parse** or **TryParse** method found on numeric types (int, long, double, and so on).

- The Parse method returns the converted number; the TryParse method returns a boolean value that indicates whether the conversion succeeded, and returns the converted number in an out parameter.

- If the string isn't in a valid format, *Parse throws an exception*, but *TryParse returns false*. When calling a Parse method, we should always use exception handling to catch a FormatException when the parse operation fails.

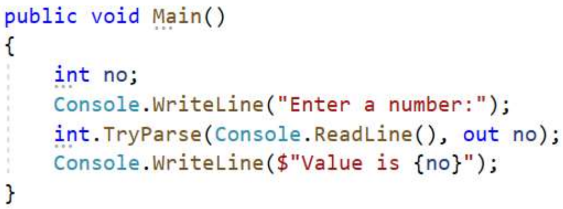
# Parse Example

Parse



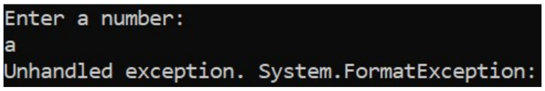
```
public void Main()
{
    int no;
    Console.WriteLine("Enter a number:");
    no = int.Parse(Console.ReadLine());
    Console.WriteLine($"Value is {no}");
}
```

TryParse



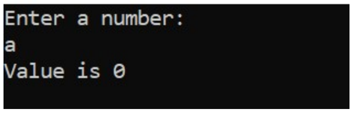
```
public void Main()
{
    int no;
    Console.WriteLine("Enter a number:");
    int.TryParse(Console.ReadLine(), out no);
    Console.WriteLine($"Value is {no}");
}
```

**Output with error**

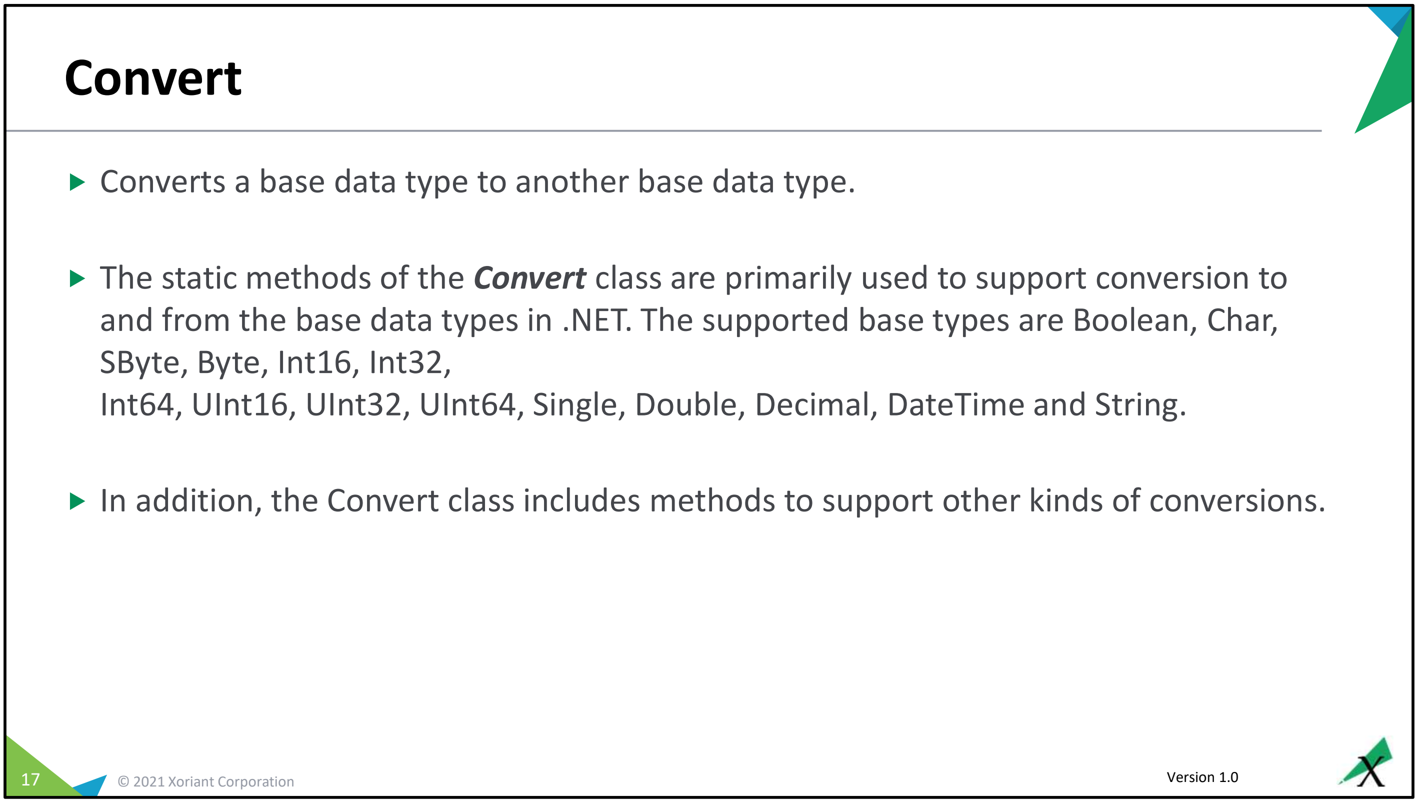


```
Enter a number:
a
Unhandled exception. System.FormatException:
```

**Output without error**



```
Enter a number:
a
Value is 0
```



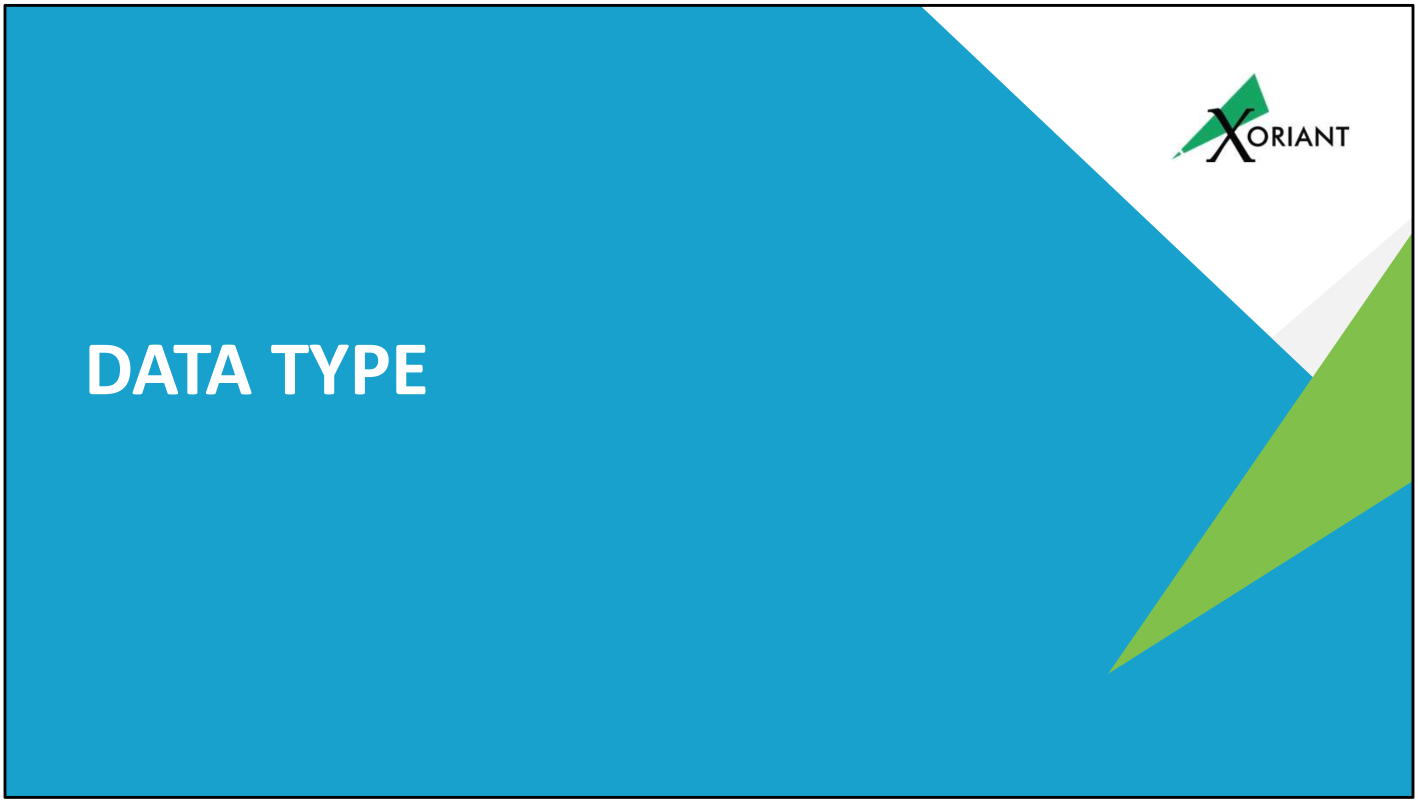
# Convert

- Converts a base data type to another base data type.
- The static methods of the ***Convert*** class are primarily used to support conversion to and from the base data types in .NET. The supported base types are Boolean, Char, SByte, Byte, Int16, Int32,
  Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime and String.
- In addition, the Convert class includes methods to support other kinds of conversions.

17

Version 1.0

For more details visit: https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0

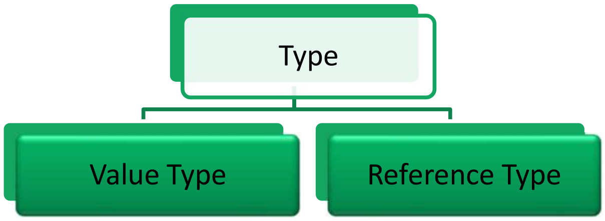


# DATA TYPE

# Overview of Data Type System

- There are two different categories of  data types.

1. Value types
2. Reference types



# Value Types

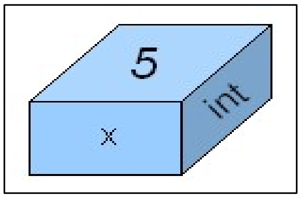
- In a variable that holds a **value type**, the data itself is directly contained within the memory allotted to the variable

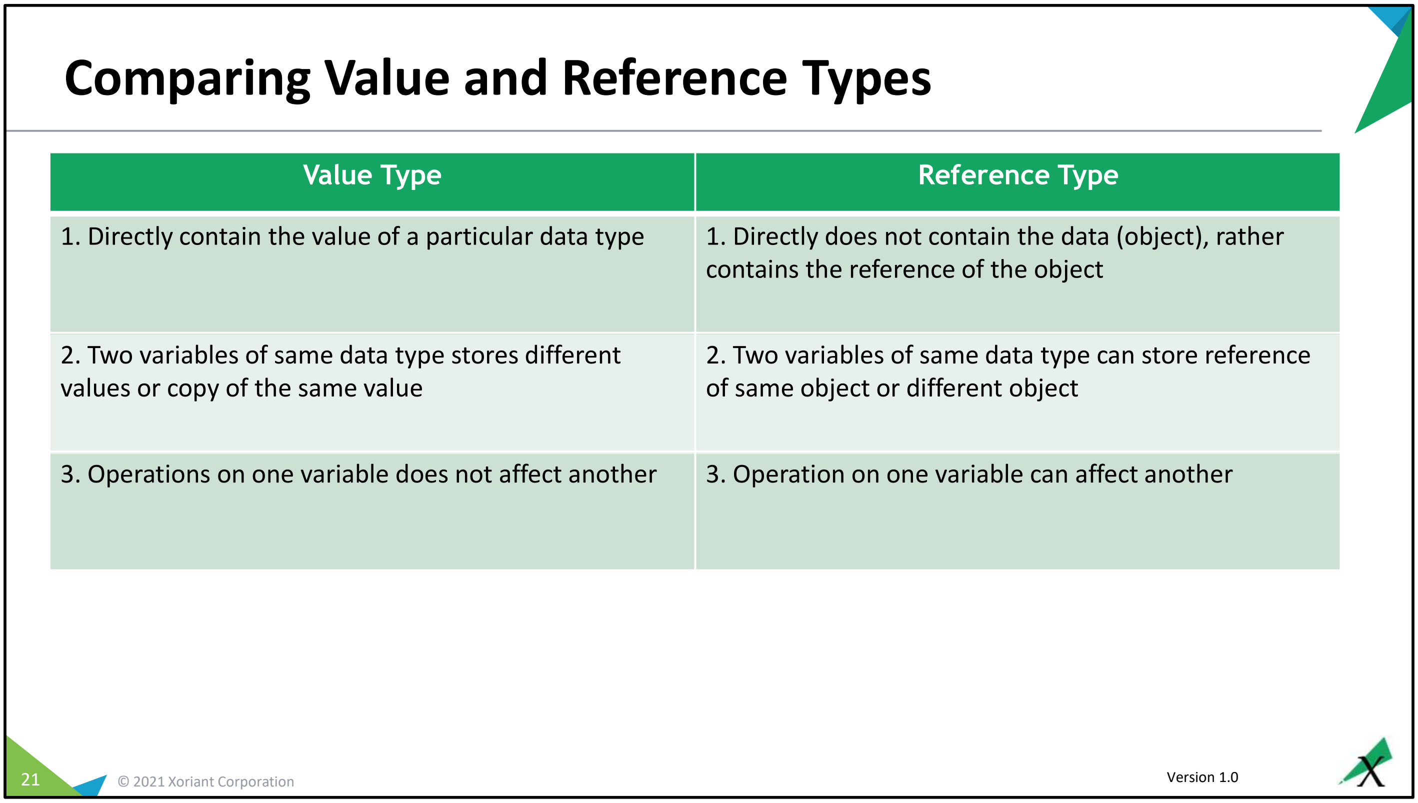
**Example:**

*int x = 5;*

The above code declares an 32-bit signed integer variable, called x, initialized with a value of 5. The following figure represents the corresponding variable diagram:

The following figure represents the corresponding variable diagram:





# Comparing Value and Reference Types

| Value Type | Reference Type |
|---|---|
| 1. Directly contain the value of a particular data type | 1. Directly does not contain the data (object), rather contains the reference of the object |
| 2. Two variables of same data type stores different values or copy of the same value | 2. Two variables of same data type can store reference of same object or different object |
| 3. Operations on one variable does not affect another | 3. Operation on one variable can affect another |

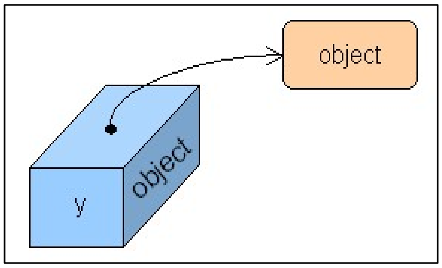
# Reference Type

- A variable that holds a **reference type** contains the address of an object stored in the heap.

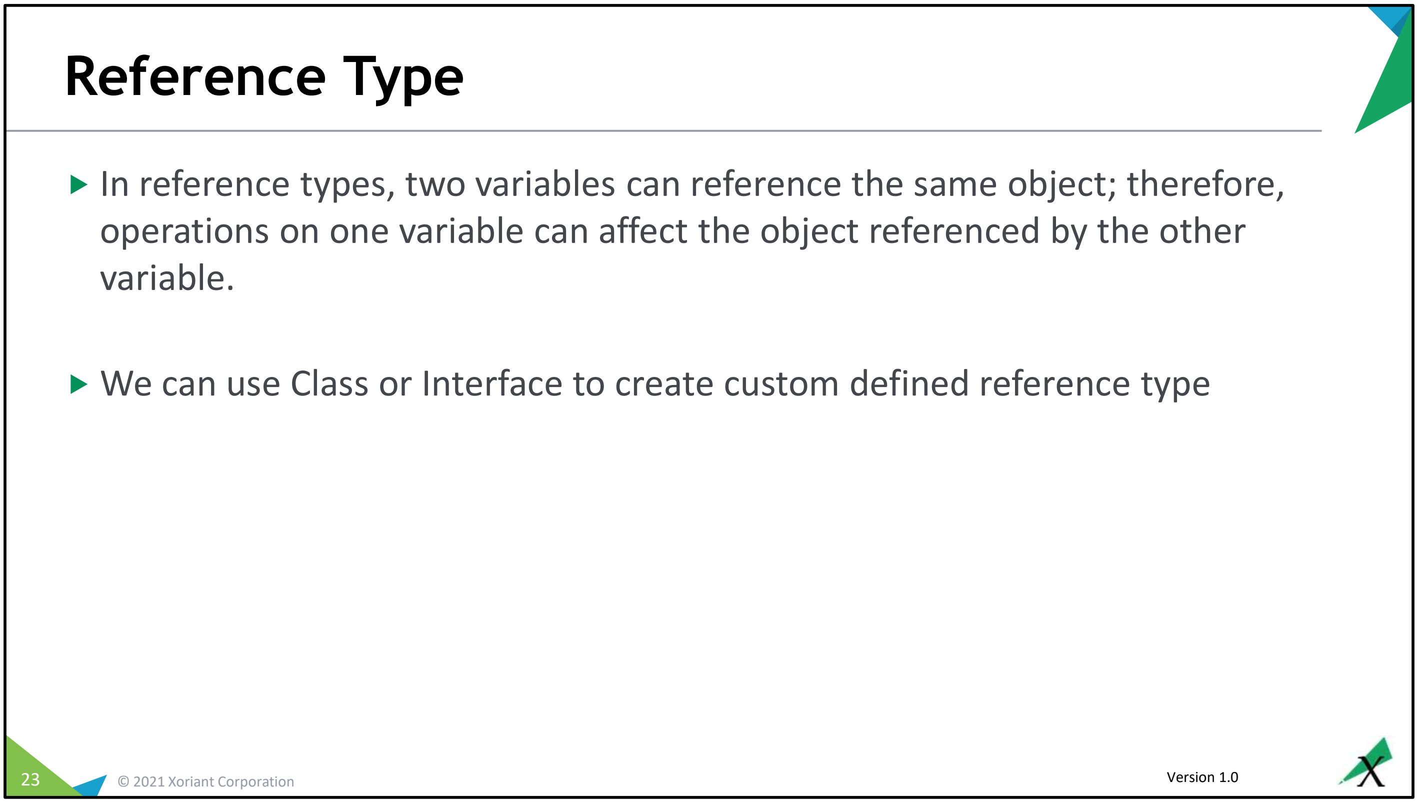
**Example:**

*object y = new object();*

The above code declares a variable called y of type object which gets initialized, thanks to the new operator, so that it refers to a new heap allocated object instance (object is the base class of all C# types, but more of this latter).



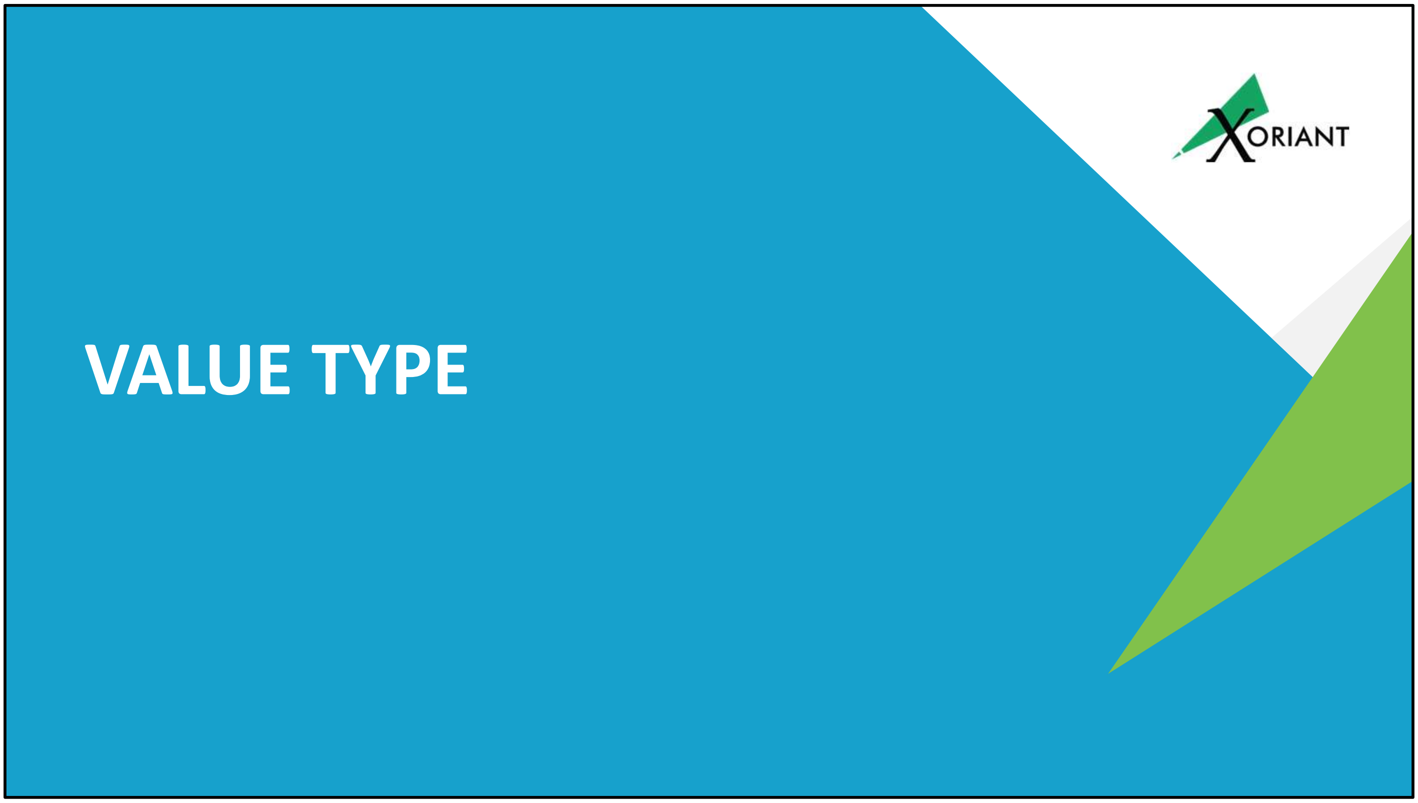
The following figure represents the corresponding variable diagram:



# Reference Type

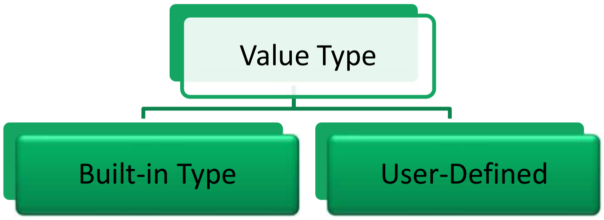
- In reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable.

- We can use Class or Interface to create custom defined reference type



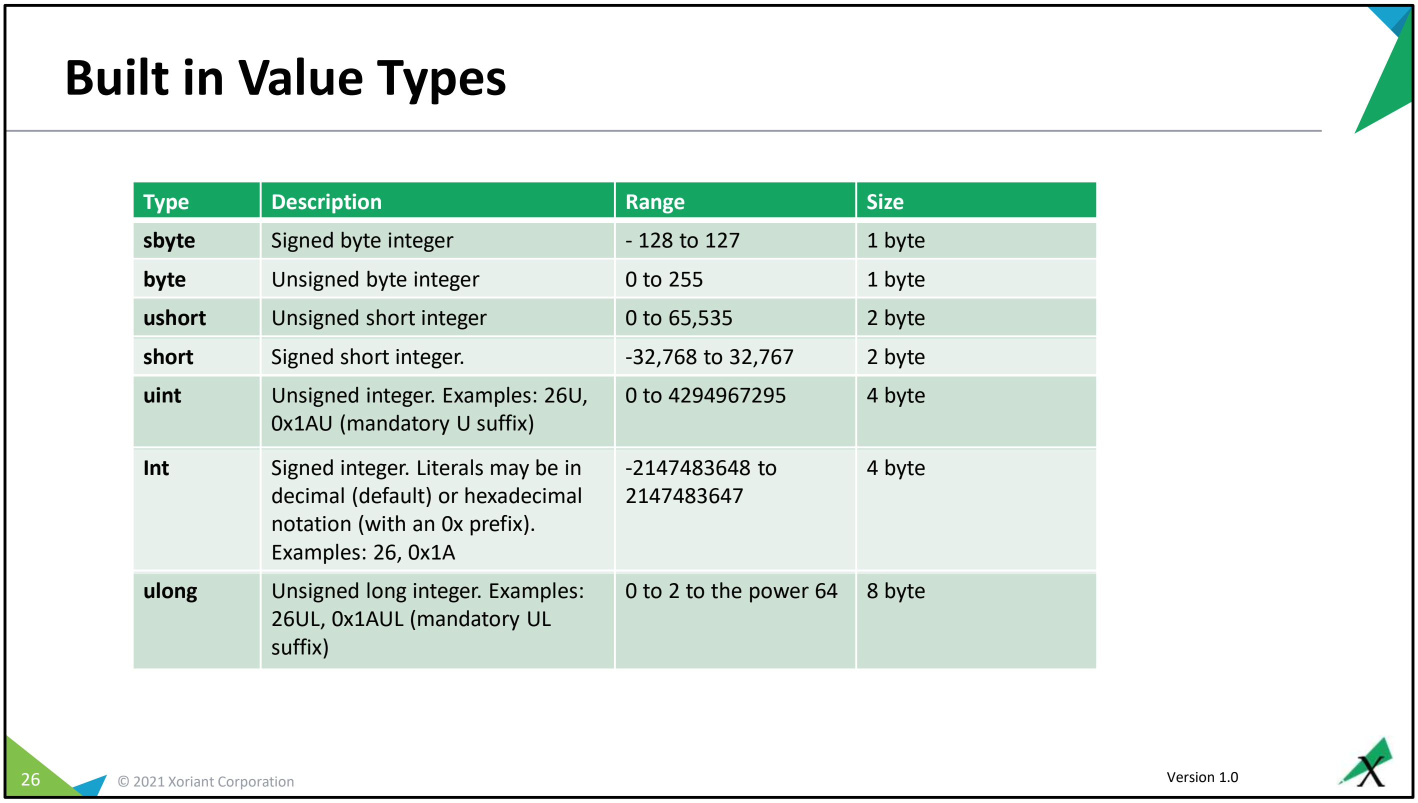
# VALUE TYPE

# Value Types



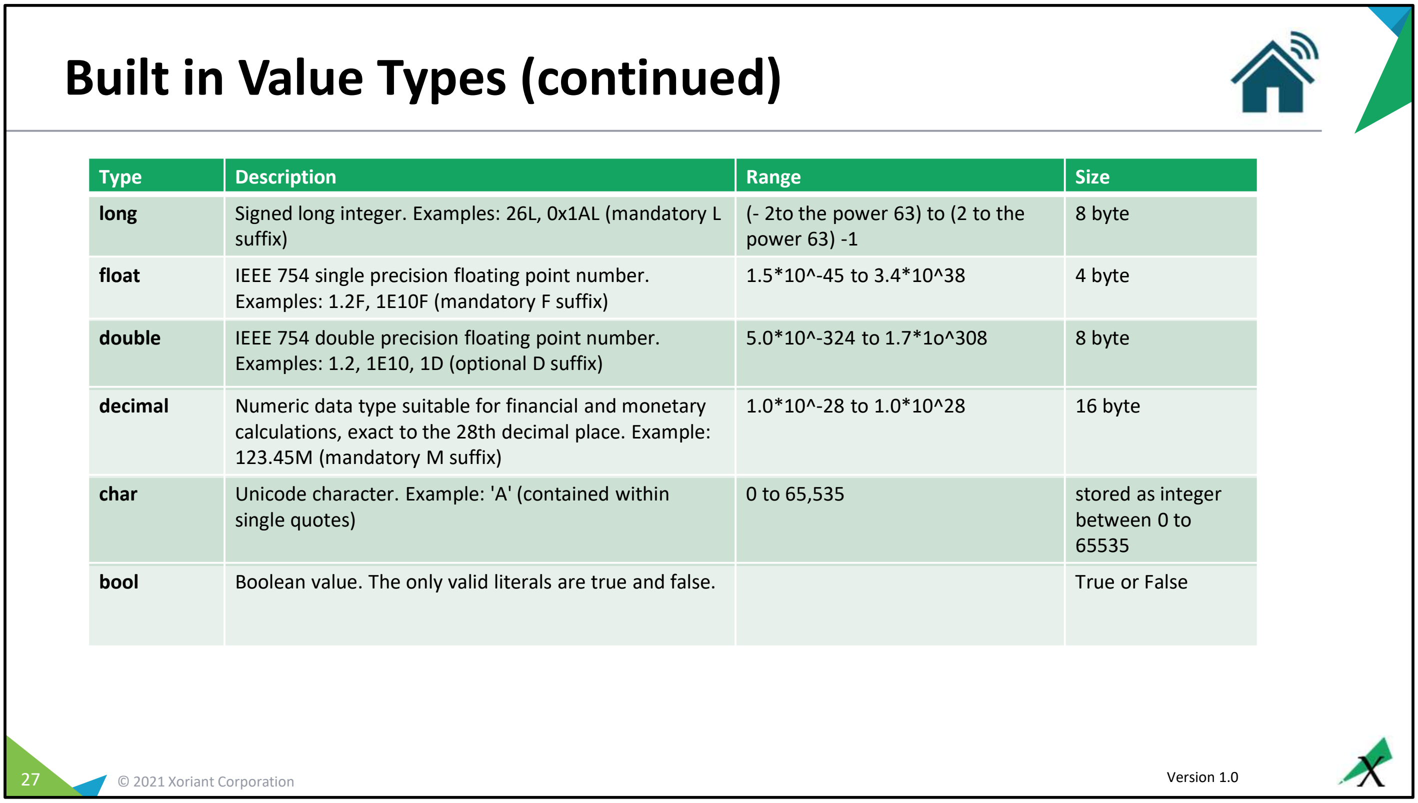
- Examples of built-in value types:
  - int
  - float

- **Examples of user-defined value types:**
  - **Enum**
  - **struct**
  - **Tuple**
  - **Nullable**



# Built in Value Types

| Type | Description | Range | Size |
|---|---|---|---|
| **sbyte** | Signed byte integer | - 128 to 127 | 1 byte |
| **byte** | Unsigned byte integer | 0 to 255 | 1 byte |
| **ushort** | Unsigned short integer | 0 to 65,535 | 2 byte |
| **short** | Signed short integer. | -32,768 to 32,767 | 2 byte |
| **uint** | Unsigned integer. Examples: 26U, 0x1AU (mandatory U suffix) | 0 to 4294967295 | 4 byte |
| **Int** | Signed integer. Literals may be in decimal (default) or hexadecimal notation (with an 0x prefix). Examples: 26, 0x1A | -2147483648 to 2147483647 | 4 byte |
| **ulong** | Unsigned long integer. Examples: 26UL, 0x1AUL (mandatory UL suffix) | 0 to 2 to the power 64 | 8 byte |



# Built in Value Types (continued)

| Type | Description | Range | Size |
|---|---|---|---|
| **long** | Signed long integer. Examples: 26L, 0x1AL (mandatory L suffix) | (- 2to the power 63) to (2 to the power 63) -1 | 8 byte |
| **float** | IEEE 754 single precision floating point number. Examples: 1.2F, 1E10F (mandatory F suffix) | 1.5*10^-45 to 3.4*10^38 | 4 byte |
| **double** | IEEE 754 double precision floating point number. Examples: 1.2, 1E10, 1D (optional D suffix) | 5.0*10^-324 to 1.7*1o^308 | 8 byte |
| **decimal** | Numeric data type suitable for financial and monetary calculations, exact to the 28th decimal place. Example: 123.45M (mandatory M suffix) | 1.0*10^-28 to 1.0*10^28 | 16 byte |
| **char** | Unicode character. Example: 'A' (contained within single quotes) | 0 to 65,535 | stored as integer between 0 to 65535 |
| **bool** | Boolean value. The only valid literals are true and false. | | True or False |

27

Version 1.0

# User Defined Types

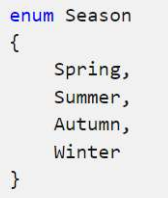
- User defined types are also know as custom types.
- These types are developer defined types.
- To create these types we can use
  - Structure
  - Enum
  - Tuple
  - Nullable
  - Class
  - Interface

Value Type

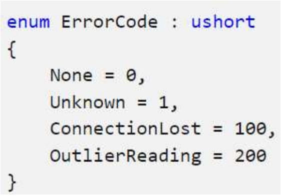
Reference Type

# User Defined Types - Enumeration

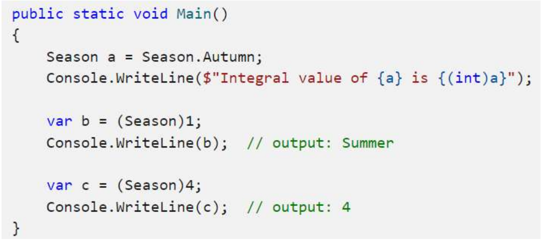
- An **Enumeration type** (or enum type) is a value type defined by a set of named constants of the underlying integral numeric type.
- To define an enumeration type, use the enum keyword and specify the names of enum members:



```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

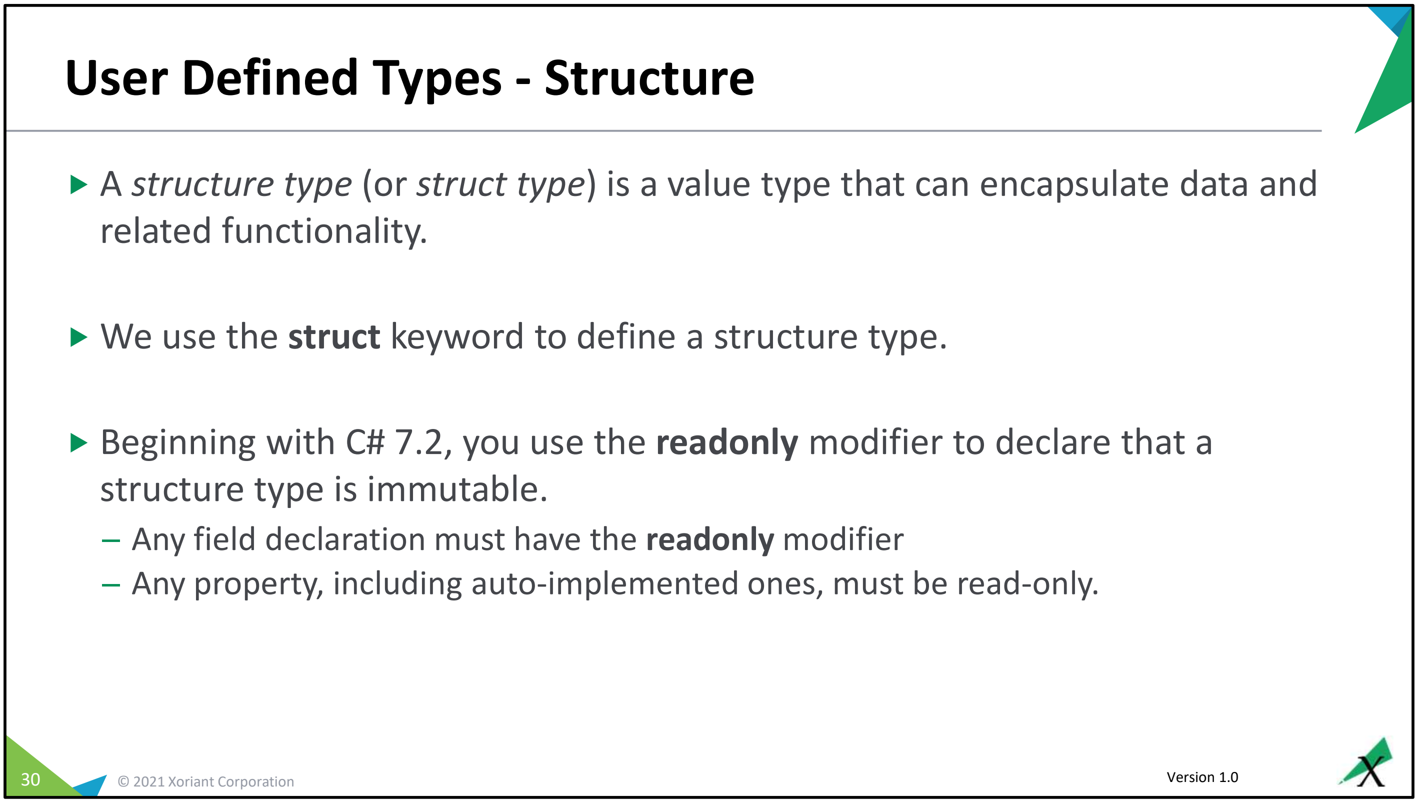


```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```



```
public static void Main()
{
    Season a = Season.Autumn;
    Console.WriteLine($"Integral value of {a} is {(int)a}");

    var b = (Season)1;
    Console.WriteLine(b);  // output: Summer

    var c = (Season)4;
    Console.WriteLine(c);  // output: 4
}
```

https://docs.microsoft.com/en-us/dotnet/api/system.enum?view=net-6.0
The *integral numeric types* represent integer numbers. All integral numeric types are value types. They are also simple types and can be initialized with literals. All integral numeric types support arithmetic, bitwise logical, comparison, and equality operators.



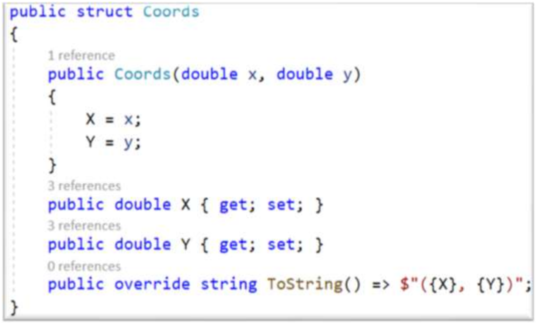
## User Defined Types - Structure

- A *structure type* (or *struct type*) is a value type that can encapsulate data and related functionality.
- We use the **struct** keyword to define a structure type.
- Beginning with C# 7.2, you use the **readonly** modifier to declare that a structure type is immutable.
  - Any field declaration must have the **readonly** modifier
  - Any property, including auto-implemented ones, must be read-only.

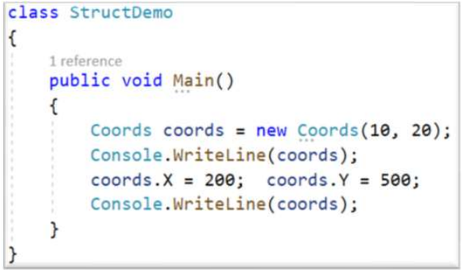
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. In the case of a structure-type variable, an instance of the type is copied.

# User Defined Types - Structure

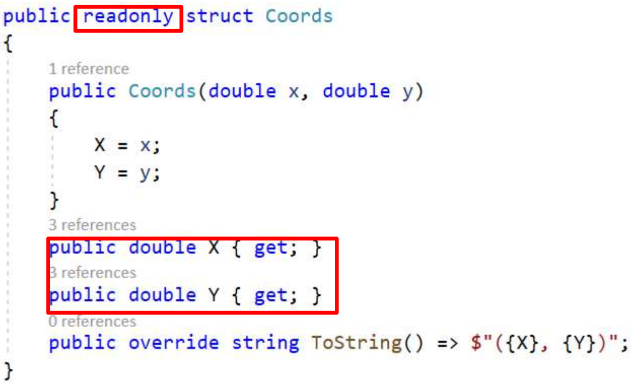


```
public struct Coords
{
    1 reference
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }
    3 references
    public double X { get; set; }
    3 references
    public double Y { get; set; }
    0 references
    public override string ToString() => $"({X}, {Y})";
}
```

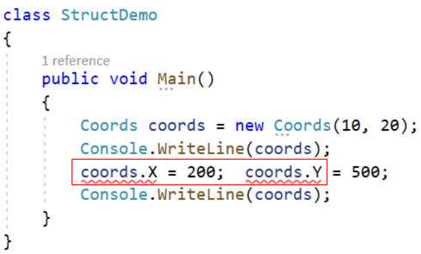


```
class StructDemo
{
    1 reference
    public void Main()
    {
        Coords coords = new Coords(10, 20);
        Console.WriteLine(coords);
        coords.X = 200;  coords.Y = 500;
        Console.WriteLine(coords);
    }
}
```

# User Defined Types – Readonly Structure

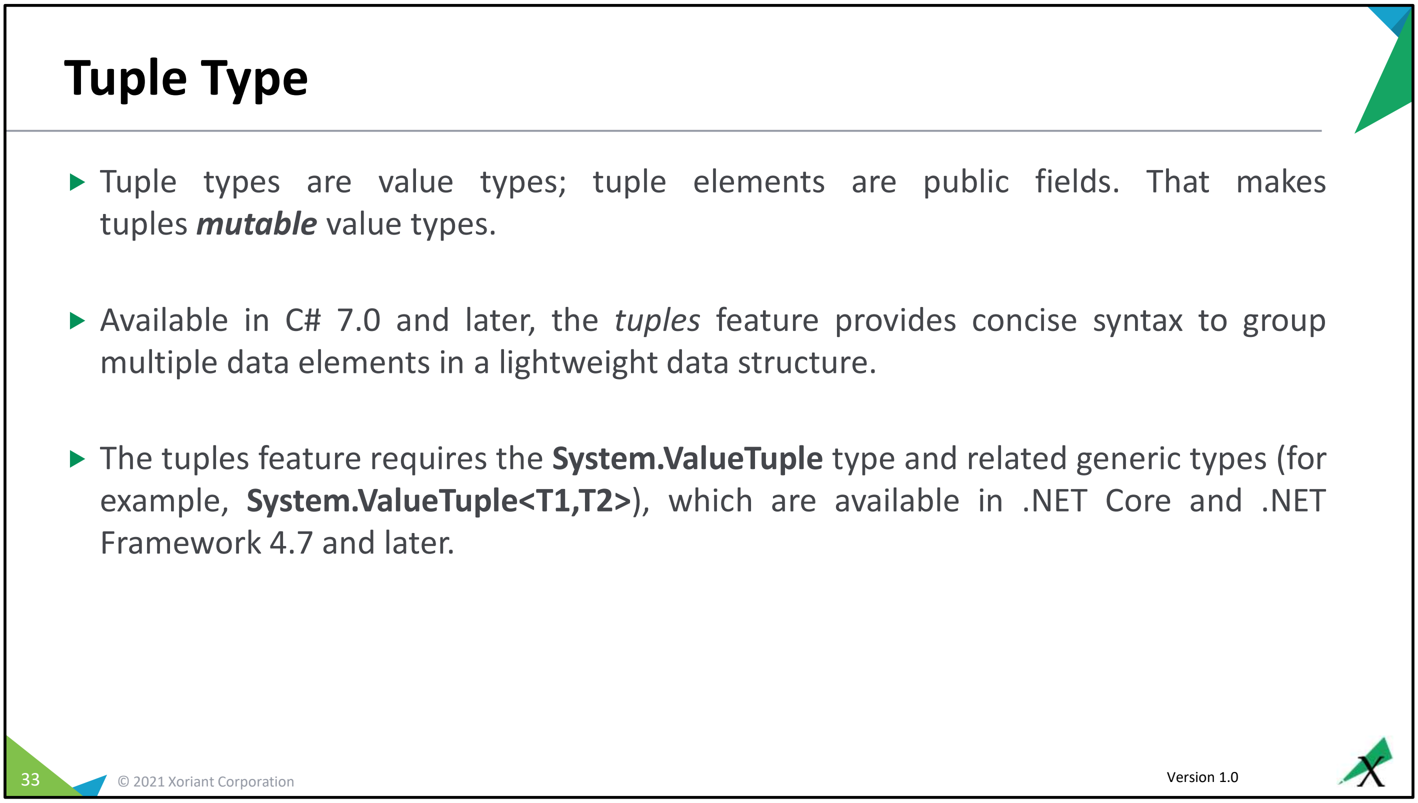


```
public readonly struct Coords
{
    1 reference
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }
    3 references
    public double X { get; }
    3 references
    public double Y { get; }
    0 references
    public override string ToString() => $"({X}, {Y})";
}
```



```
class StructDemo
{
    1 reference
    public void Main()
    {
        Coords coords = new Coords(10, 20);
        Console.WriteLine(coords);
        coords.X = 200;  coords.Y = 500;
        Console.WriteLine(coords);
    }
}
```

Note: Here all the properties will be readonly. Thus we get compile time error while initialization.



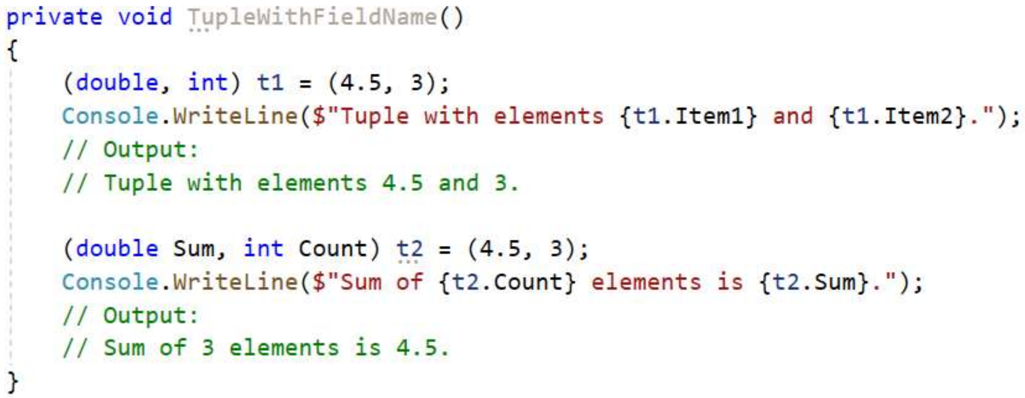
## Tuple Type

- Tuple types are value types; tuple elements are public fields. That makes tuples ***mutable*** value types.
- Available in C# 7.0 and later, the *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure.
- The tuples feature requires the **System.ValueTuple** type and related generic types (for example, **System.ValueTuple<T1,T2>**), which are available in .NET Core and .NET Framework 4.7 and later.

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

## Tuple Type – Example 1

//1.Tuple with field name

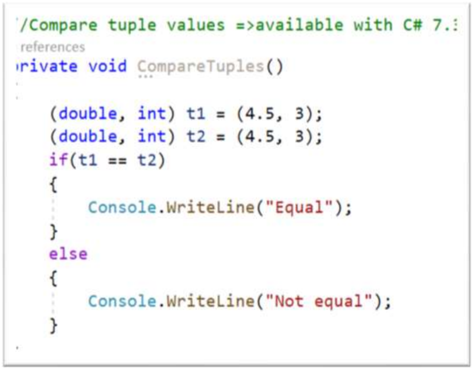


```
private void TupleWithFieldName()
{
    (double, int) t1 = (4.5, 3);
    Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
    // Output:
    // Tuple with elements 4.5 and 3.

    (double Sum, int Count) t2 = (4.5, 3);
    Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
    // Output:
    // Sum of 3 elements is 4.5.
}
```

We cannot define methods in a tuple type, but you can use the methods provided by .NET, as follows
(double, int) t = (4.5, 3); Console.WriteLine(t.ToString()); Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}."); // Output: // (4.5, 3) // Hash code of (4.5, 3) is 718460086.

# Tuple Type – Example 2

//2.Compare tuple values

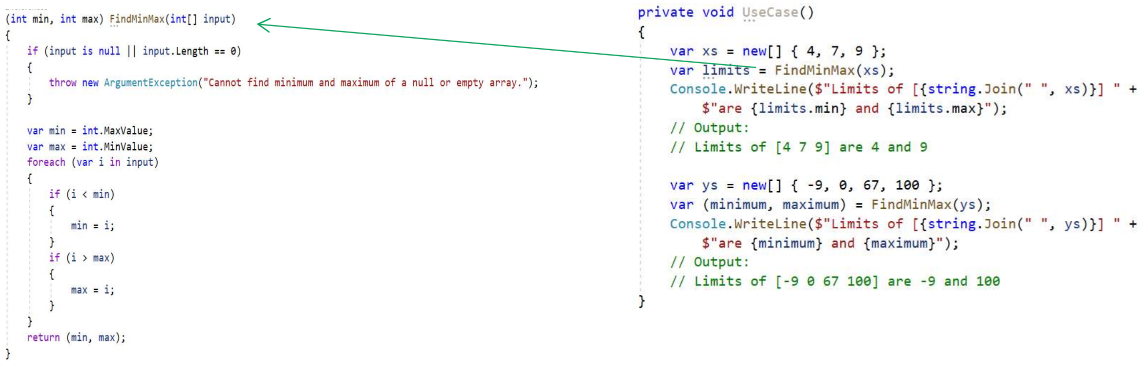


```
'/Compare tuple values =>available with C# 7.:
 references
rivate void CompareTuples()

    (double, int) t1 = (4.5, 3);
    (double, int) t2 = (4.5, 3);
    if(t1 == t2)
    {
        Console.WriteLine("Equal");
    }
    else
    {
        Console.WriteLine("Not equal");
    }
```

We cannot define methods in a tuple type, but you can use the methods provided by .NET, as follows
(double, int) t = (4.5, 3); Console.WriteLine(t.ToString()); Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}."); // Output: // (4.5, 3) // Hash code of (4.5, 3) is 718460086.

# Tuple Type – Example 3

- //3. Tuple as output parameter



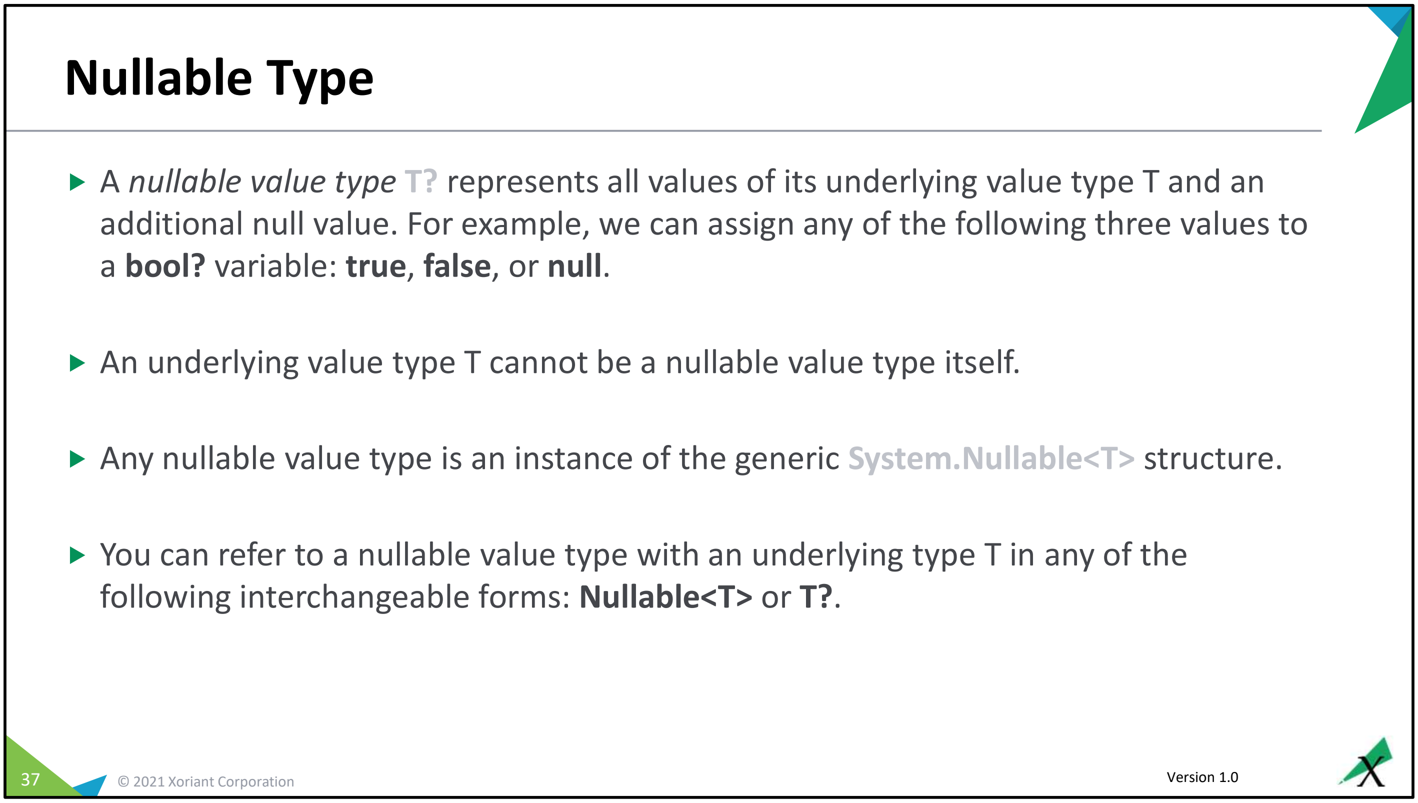
```
(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

```
private void UseCase()
{
    var xs = new[] { 4, 7, 9 };
    var limits = FindMinMax(xs);
    Console.WriteLine($"Limits of [{string.Join(" ", xs)}] " +
        $"are {limits.min} and {limits.max}");
    // Output:
    // Limits of [4 7 9] are 4 and 9

    var ys = new[] { -9, 0, 67, 100 };
    var (minimum, maximum) = FindMinMax(ys);
    Console.WriteLine($"Limits of [{string.Join(" ", ys)}] " +
        $"are {minimum} and {maximum}");
    // Output:
    // Limits of [-9 0 67 100] are -9 and 100
}
```

Version 1.0

We cannot define methods in a tuple type, but you can use the methods provided by .NET, as follows

(double, int) t = (4.5, 3); Console.WriteLine(t.ToString()); Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}."); // Output: // (4.5, 3) // Hash code of (4.5, 3) is 718460086.



# Nullable Type

- A *nullable value type* T? represents all values of its underlying value type T and an additional null value. For example, we can assign any of the following three values to a **bool?** variable: **true**, **false**, or **null**.
- An underlying value type T cannot be a nullable value type itself.
- Any nullable value type is an instance of the generic System.Nullable<T> structure.
- You can refer to a nullable value type with an underlying type T in any of the following interchangeable forms: **Nullable<T>** or **T?**.

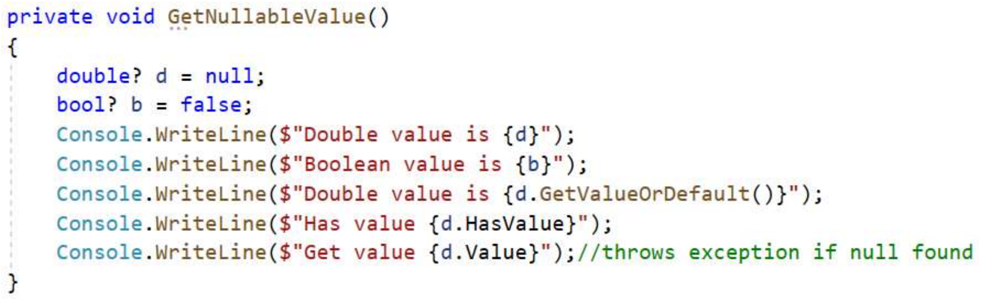
Version 1.0

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types

C# 8.0 introduces the nullable reference types feature.

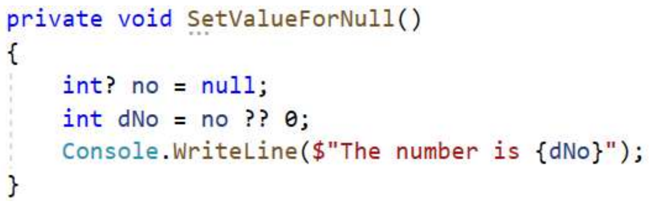
# Nullable Type - Example

//1. Get data from nullable



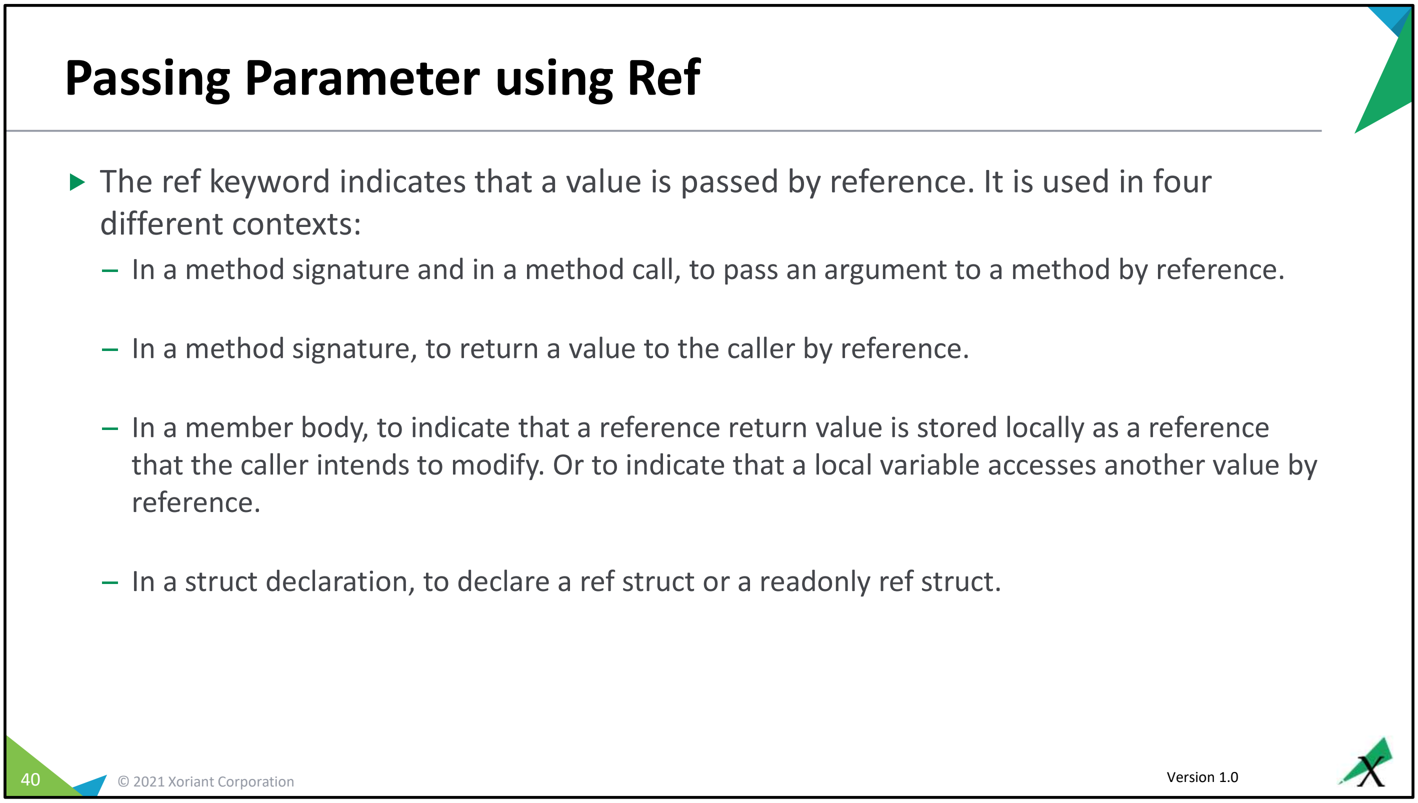
```
private void GetNullableValue()
{
    double? d = null;
    bool? b = false;
    Console.WriteLine($"Double value is {d}");
    Console.WriteLine($"Boolean value is {b}");
    Console.WriteLine($"Double value is {d.GetValueOrDefault()}");
    Console.WriteLine($"Has value {d.HasValue}");
    Console.WriteLine($"Get value {d.Value}");//throws exception if null found
}
```

//2. Set default value if null



```
private void SetValueForNull()
{
    int? no = null;
    int dNo = no ?? 0;
    Console.WriteLine($"The number is {dNo}");
}
```

# PASSING PARAMETER USING REF



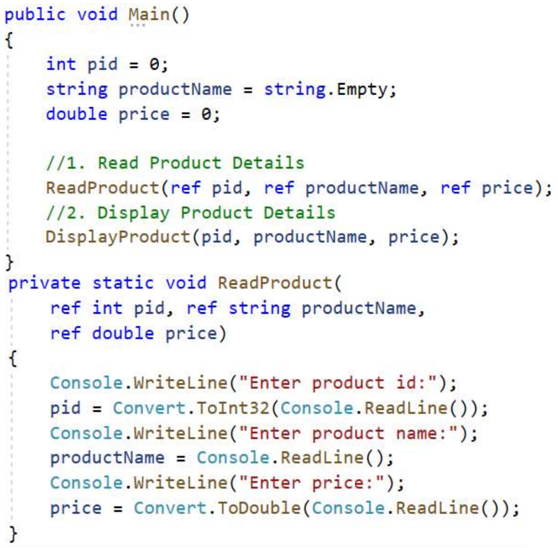
# Passing Parameter using Ref

- The ref keyword indicates that a value is passed by reference. It is used in four different contexts:
  - In a method signature and in a method call, to pass an argument to a method by reference.
  - In a method signature, to return a value to the caller by reference.
  - In a member body, to indicate that a reference return value is stored locally as a reference that the caller intends to modify. Or to indicate that a local variable accesses another value by reference.
  - In a struct declaration, to declare a ref struct or a readonly ref struct.

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

# Passing Parameter using Ref - Example

- **Using reference parameters**
  - Use the **ref** keyword in method declaration and call
  - Match types and variable values
  - Changes made in the method affect the caller
  - Assign parameter value before calling the method



```
public void Main()
{
    int pid = 0;
    string productName = string.Empty;
    double price = 0;

    //1. Read Product Details
    ReadProduct(ref pid, ref productName, ref price);
    //2. Display Product Details
    DisplayProduct(pid, productName, price);
}
private static void ReadProduct(
    ref int pid, ref string productName,
    ref double price)
{
    Console.WriteLine("Enter product id:");
    pid = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter product name:");
    productName = Console.ReadLine();
    Console.WriteLine("Enter price:");
    price = Convert.ToDouble(Console.ReadLine());
}
```

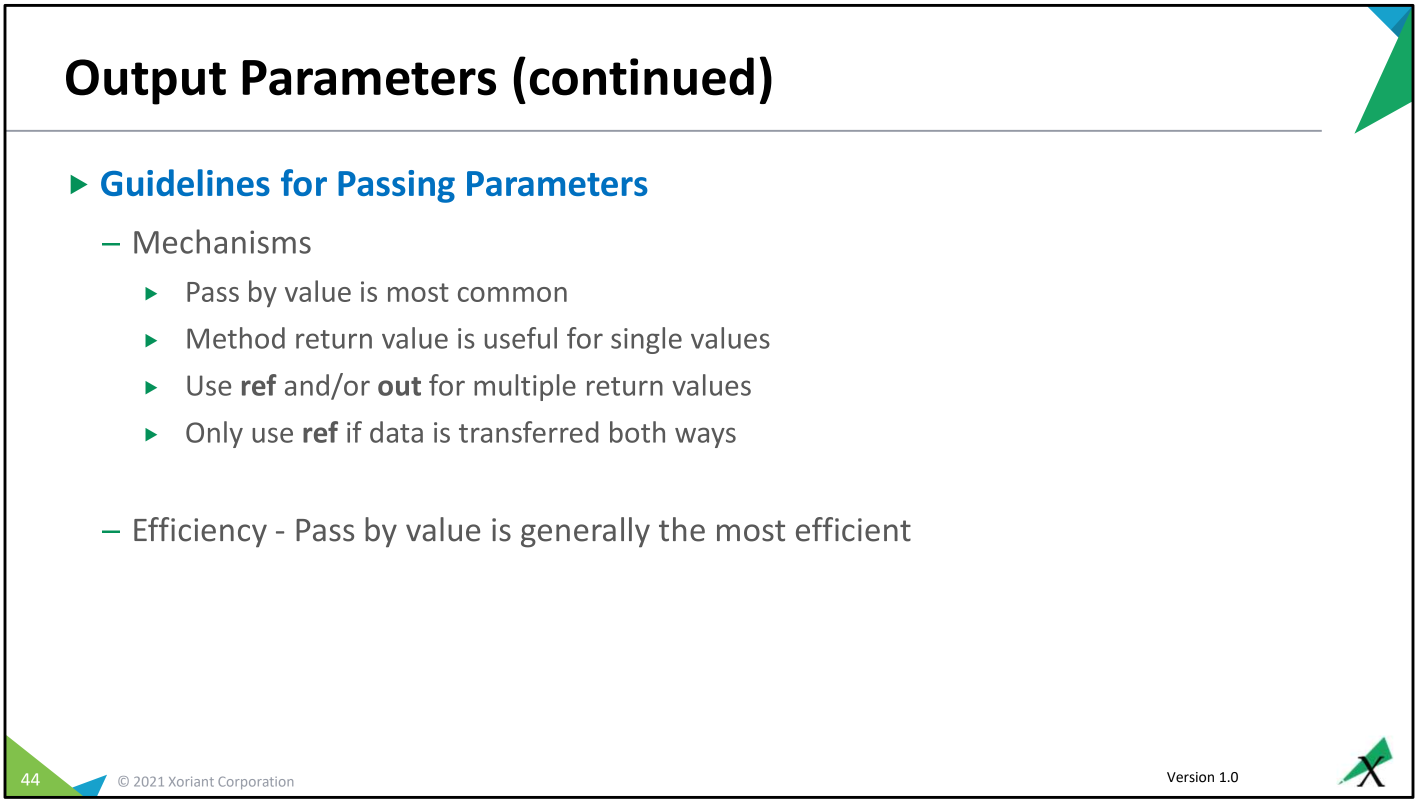
# OUTPUT PARAMETER

# Output Parameters

- What are Output Parameters?

  Values are passed out but not in

- Using output parameters
  - Like **ref**, but values are not passed into the method
  - Use **out** keyword in method declaration and call



# Output Parameters (continued)

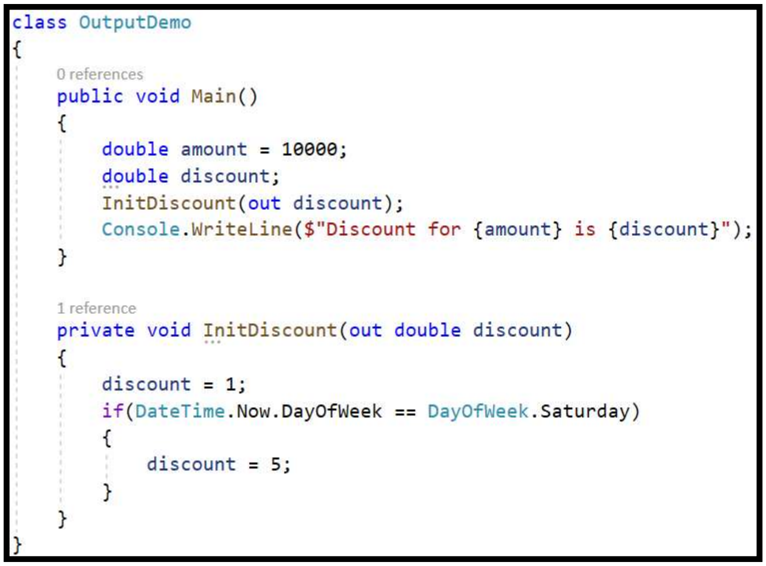
- **Guidelines for Passing Parameters**
  - Mechanisms
    - Pass by value is most common
    - Method return value is useful for single values
    - Use **ref** and/or **out** for multiple return values
    - Only use **ref** if data is transferred both ways
  - Efficiency - Pass by value is generally the most efficient

44

Version 1.0

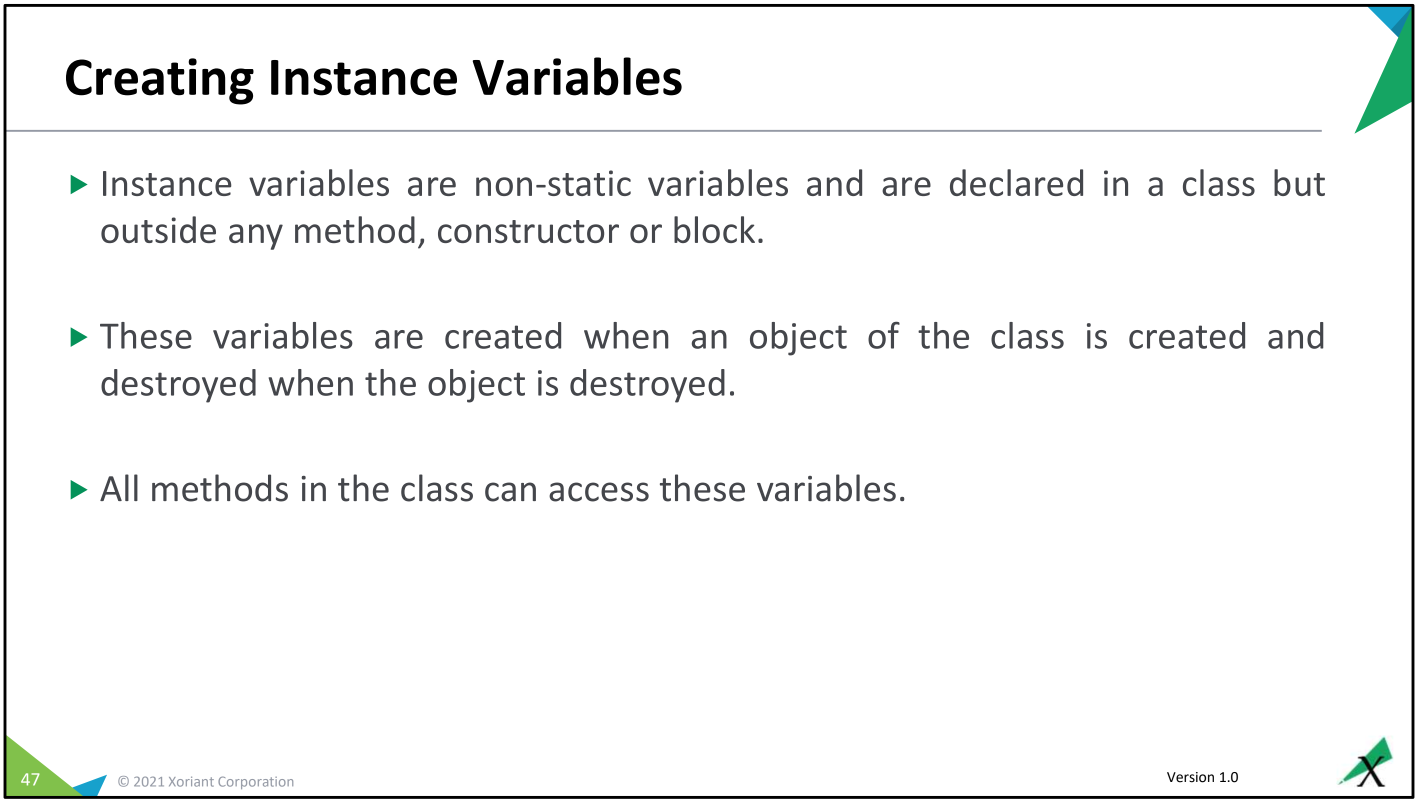
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier

# Output Parameters - Example



```
class OutputDemo
{
    0 references
    public void Main()
    {
        double amount = 10000;
        double discount;
        InitDiscount(out discount);
        Console.WriteLine($"Discount for {amount} is {discount}");
    }

    1 reference
    private void InitDiscount(out double discount)
    {
        discount = 1;
        if(DateTime.Now.DayOfWeek == DayOfWeek.Saturday)
        {
            discount = 5;
        }
    }
}
```

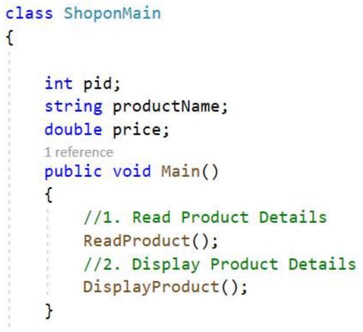
# CREATING INSTANCE VARIABLE



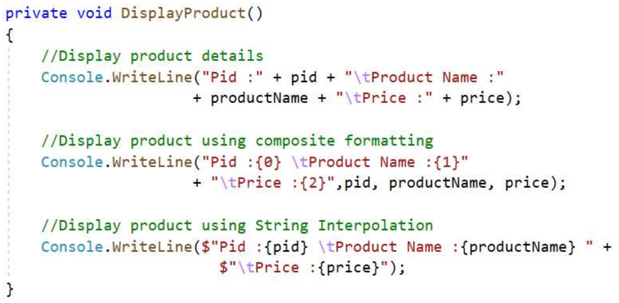
## Creating Instance Variables

- Instance variables are non-static variables and are declared in a class but outside any method, constructor or block.
- These variables are created when an object of the class is created and destroyed when the object is destroyed.
- All methods in the class can access these variables.

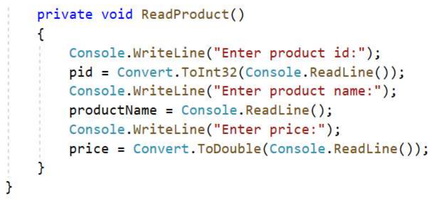
# Creating Instance Variables - Example



```
class ShoponMain
{

    int pid;
    string productName;
    double price;
    1 reference
    public void Main()
    {
        //1. Read Product Details
        ReadProduct();
        //2. Display Product Details
        DisplayProduct();
    }
```

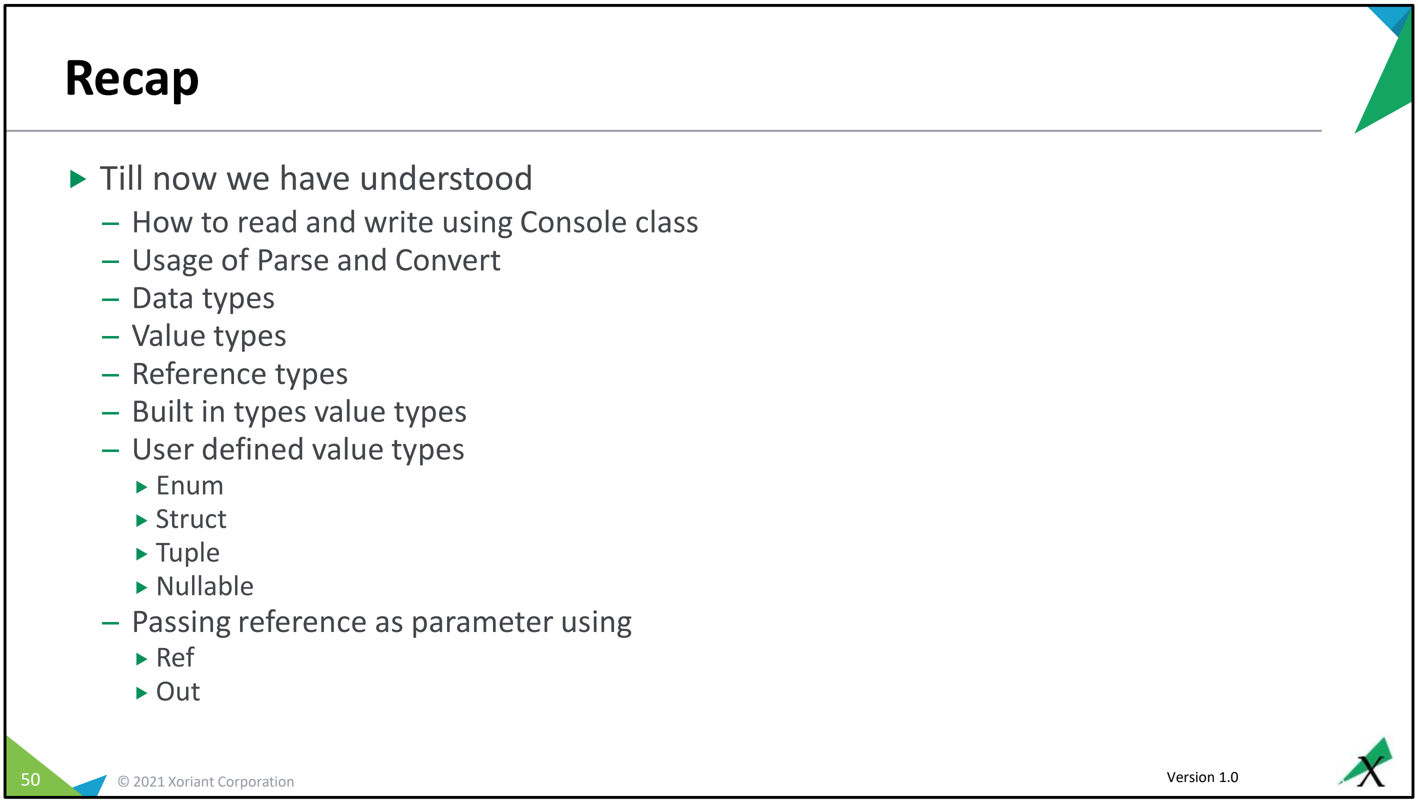


```
private void DisplayProduct()
{
    //Display product details
    Console.WriteLine("Pid :" + pid + "\tProduct Name :"
                     + productName + "\tPrice :" + price);

    //Display product using composite formatting
    Console.WriteLine("Pid :{0} \tProduct Name :{1}"
                     + "\tPrice :{2}",pid, productName, price);

    //Display product using String Interpolation
    Console.WriteLine($"Pid :{pid} \tProduct Name :{productName} " +
                        $"\tPrice :{price}");
}
```



```
    private void ReadProduct()
    {
        Console.WriteLine("Enter product id:");
        pid = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter product name:");
        productName = Console.ReadLine();
        Console.WriteLine("Enter price:");
        price = Convert.ToDouble(Console.ReadLine());
    }
}
```

# LET'S RECAP

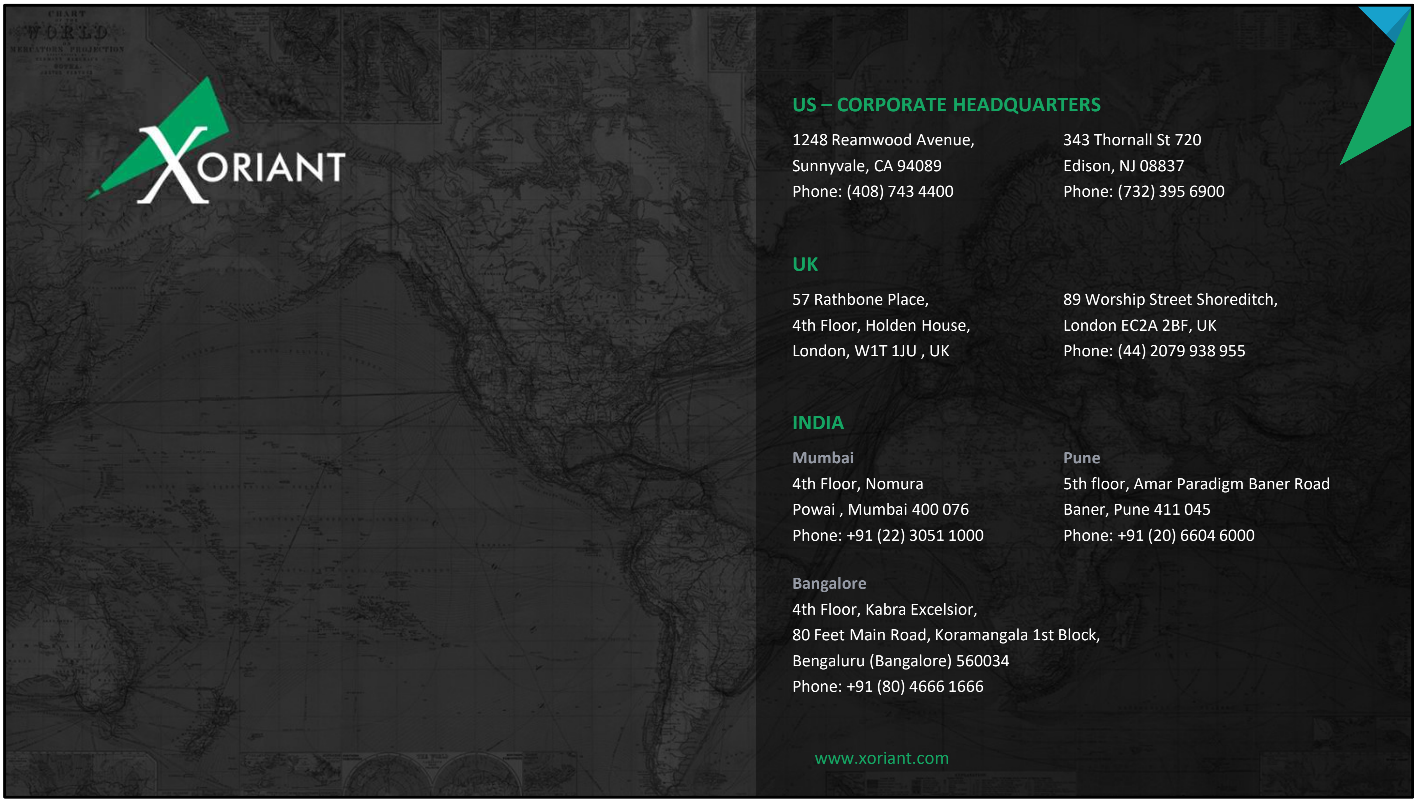


# Recap

- Till now we have understood
  - How to read and write using Console class
  - Usage of Parse and Convert
  - Data types
  - Value types
  - Reference types
  - Built in types value types
  - User defined value types
    - Enum
    - Struct
    - Tuple
    - Nullable
  - Passing reference as parameter using
    - Ref
    - Out

# Useful Links

- https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.enum?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier



XORIANT

## US – CORPORATE HEADQUARTERS

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

## UK

57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK

89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

## INDIA

**Mumbai**
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000

**Pune**
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000

**Bangalore**
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666

www.xoriant.com