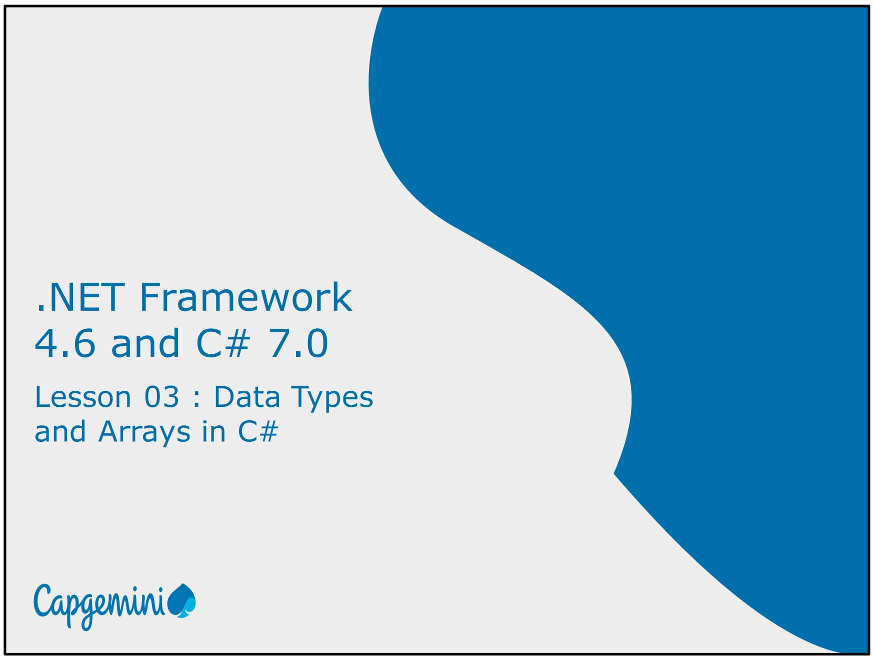
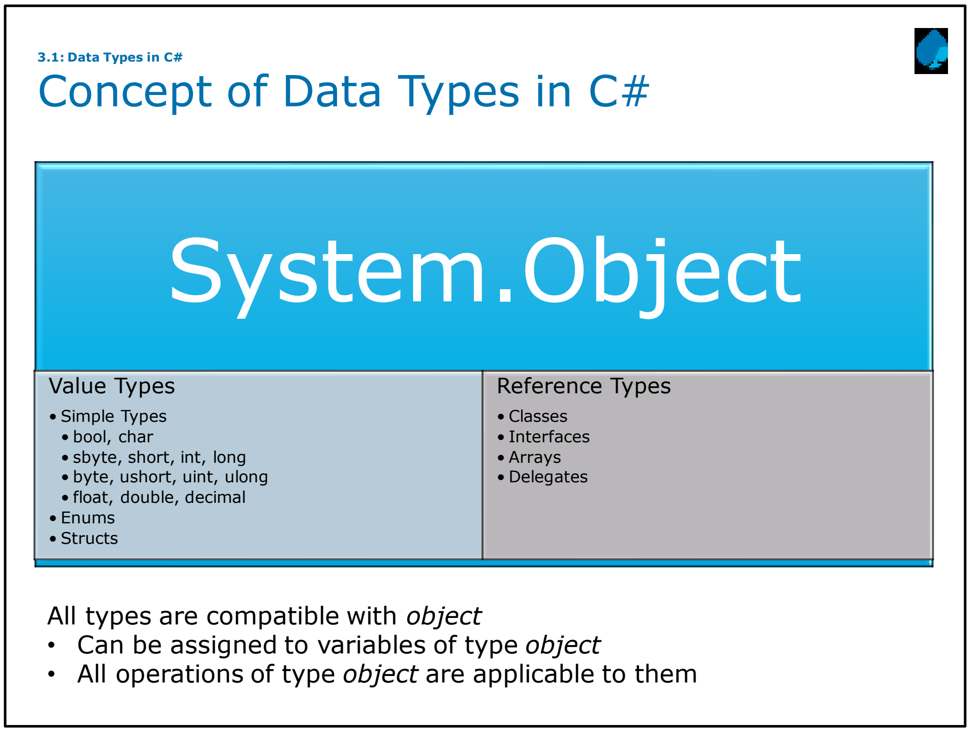


.NET Framework 4.6 and C# 7.0

Lesson 03 : Data Types and Arrays in C#

Lesson Objectives

➢ In this lesson, you will learn:
- Different data types in C#
- Value Types and Reference Types in C#
- Concept of Boxing and Unboxing
- Arrays in C# : Single Dimensional Arrays, Multi Dimensional Arrays, Jagged Arrays
- Nullable Types
- Implicitly Typed Local Variables
- Var v/s Dynamic
- Parse() v/s TryParse() v/s Convert Class Methods
- Is and as operator
- Ref v/s out keywords
- The object base class in .NET
- Equals v/s ==
- String v/s StringBuilder
- Various string class methods
- Optional parameters, named parameters



Concept of Data Types in C#

System.Object

Value Types	Reference Types
• Simple Types • bool, char • sbyte, short, int, long • byte, ushort, uint, ulong • float, double, decimal • Enums • Structs	• Classes • Interfaces • Arrays • Delegates

All types are compatible with *object*
- Can be assigned to variables of type *object*
- All operations of type *object* are applicable to them

Datatypes in C#:

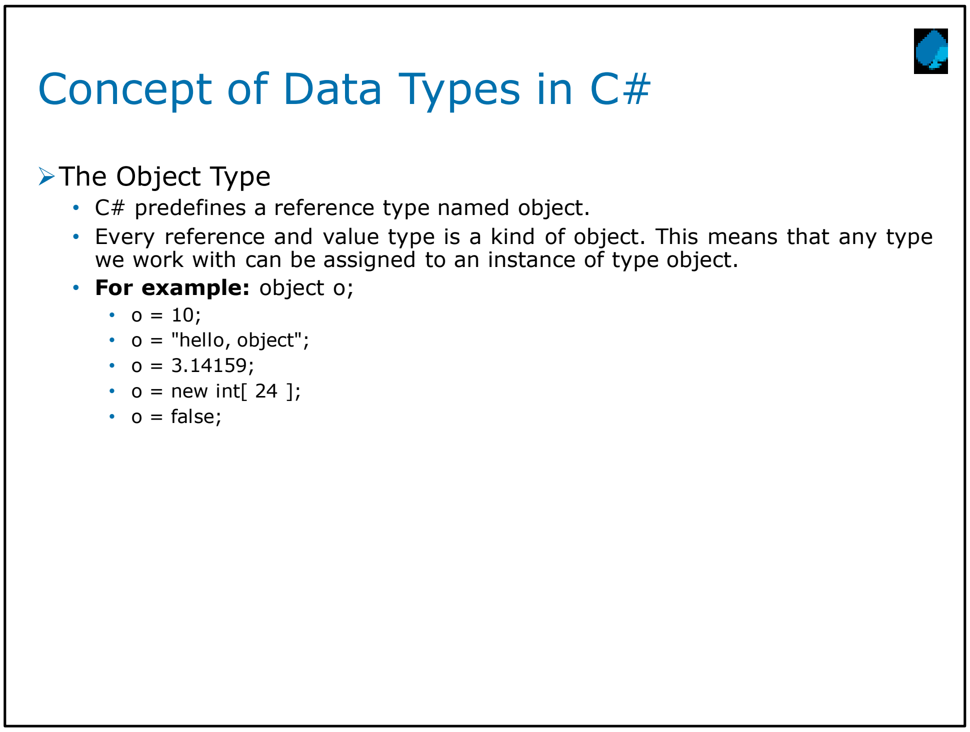
Every entity in a C# program is an object that lives on either the stack or the managed heap. Every method is defined in a class or struct declaration. There are no such things as free functions, defined outside the scope of class or struct declarations, as there are in C++. Even the built-in value types, such as int, long, double, and so on, have methods associated with them implicitly.

C# Type	**.Net Framework Type**
Bool	System.Boolean
Byte	System.Byte
Spite	System.SByte
Char	System.Char
Decimal	System.Decimal
Double	System.Double
Float	System.Single
Int	System.Int32
Uint	System.UInt32
Long	System.Int64
Ulong	System.UInt64
Object	System.Object
Short	System.Int16
Ushort	System.UInt16
String	System.String

Concept of Data Types in C#

- Storage of Basic Types:

Type	Storage
char, unsigned char, signed char	2 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	8 bytes
float	4 bytes
double	8 bytes

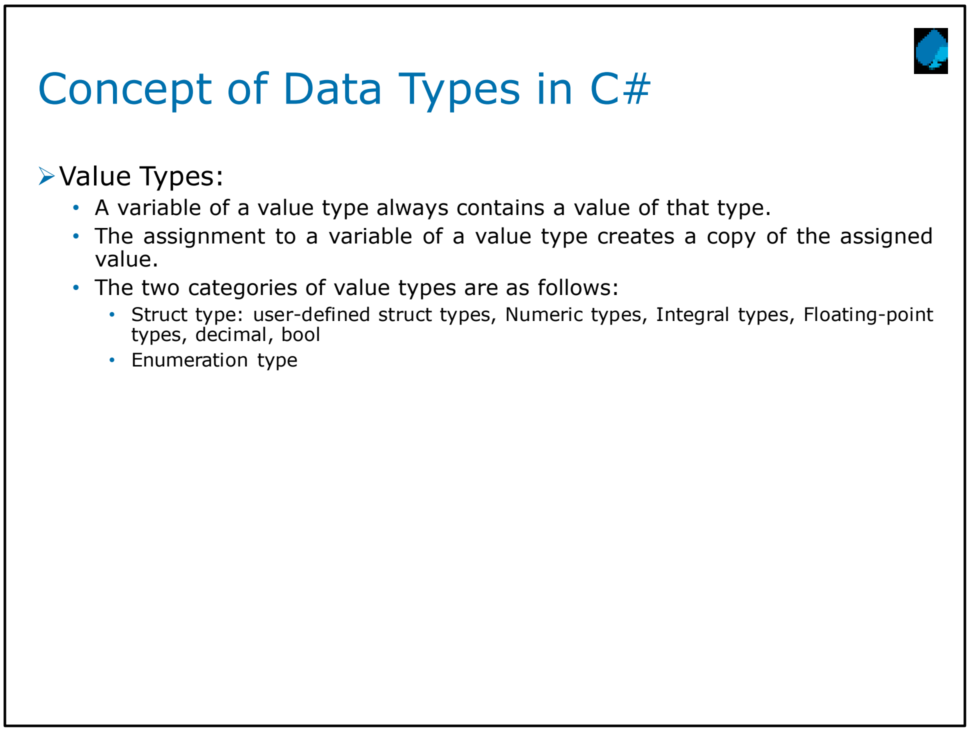


Concept of Data Types in C#

- The Object Type
  - C# predefines a reference type named object.
  - Every reference and value type is a kind of object. This means that any type we work with can be assigned to an instance of type object.
  - **For example:** object o;
    - o = 10;
    - o = "hello, object";
    - o = 3.14159;
    - o = new int[24];
    - o = false;

Data Types in C#:
Every object in the CLR derives from System.Object.
Object is the base type of every type. In C#, the object keyword is an alias for System.Object. It can be convenient that every type in the CLR and in C# derives from Object.

Value types are stored on stack



Concept of Data Types in C#

- Value Types:
  - A variable of a value type always contains a value of that type.
  - The assignment to a variable of a value type creates a copy of the assigned value.
  - The two categories of value types are as follows:
    - Struct type: user-defined struct types, Numeric types, Integral types, Floating-point types, decimal, bool
    - Enumeration type

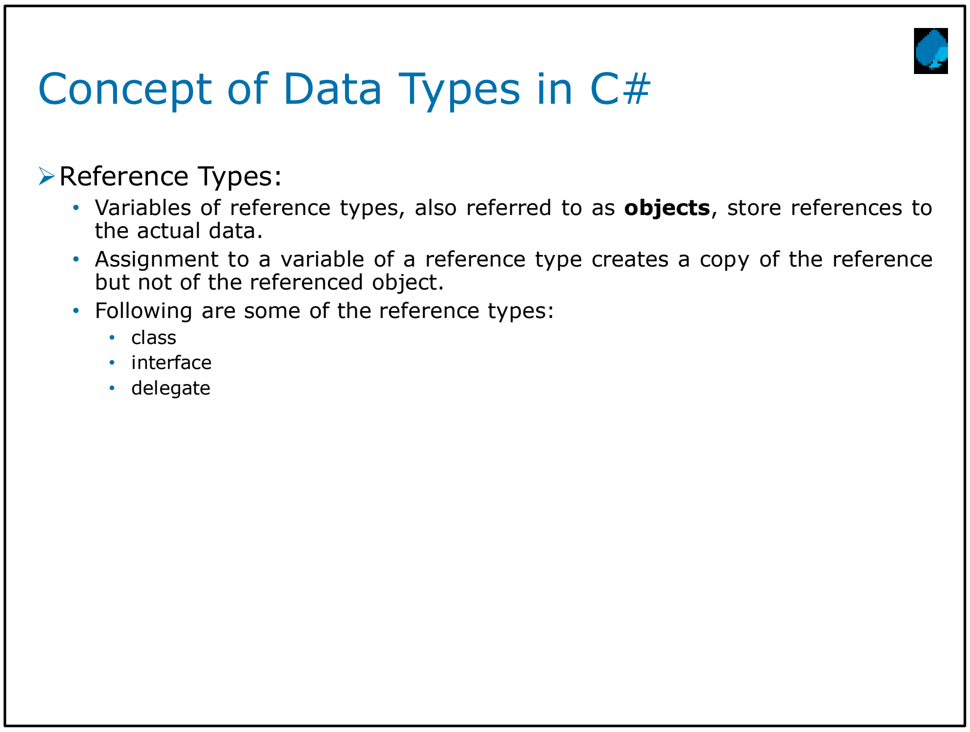
Datatypes in C#:

In the managed world of the CLR, there are two kinds of types – Value Types and Reference Types.

Value types:

Value types are defined in C# by using the struct keyword. Instances of value types are the only kind of instances that can live on the stack. They live on the heap if they are members of reference types or if they are boxed, which is discussed later.

Reference types are stored on Heap



Concept of Data Types in C#

➢ Reference Types:

- Variables of reference types, also referred to as **objects**, store references to the actual data.
- Assignment to a variable of a reference type creates a copy of the reference but not of the referenced object.
- Following are some of the reference types:
  - class
  - interface
  - delegate

Data Types in C#:

Reference types:

Reference types are defined in C# by using the class keyword. They are called reference types because the variables you use to manipulate them are actually references to objects on the managed heap. In fact, in the CLR reference-type variables are like value types that reference an object on the heap.

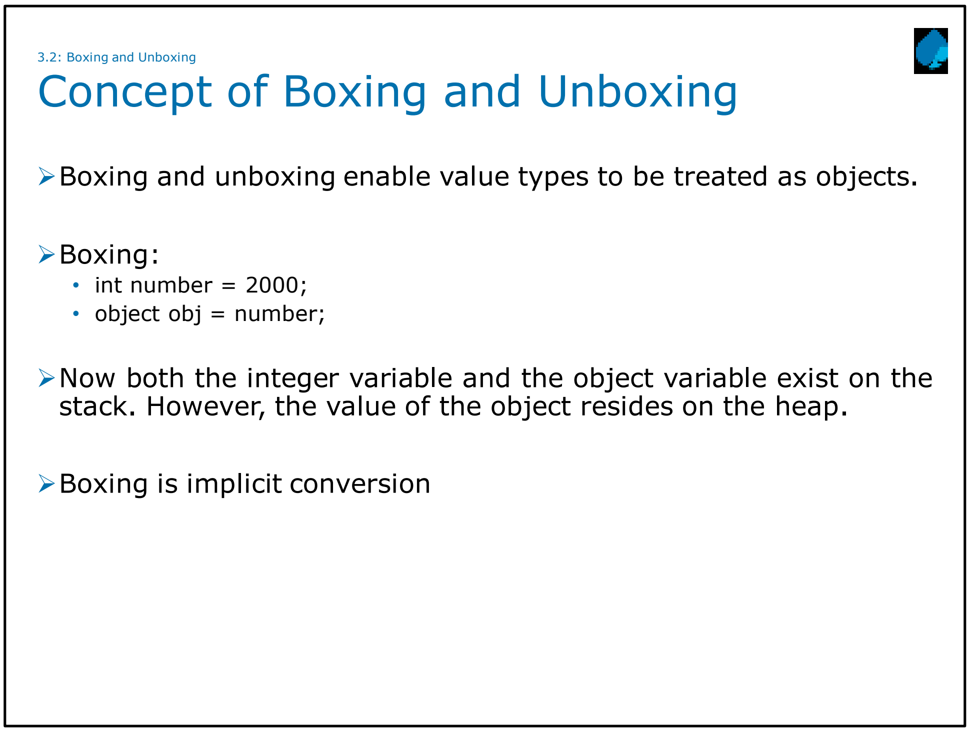
Concept of Data Types in C#

- Reference Types (contd.):
  - The following are built-in reference types:
    - object
    - string

Comparison between Data Types in C#

➢ Let us differentiate between value and reference types:

Value Types	Reference Types
The variable contains the value directly	The variable contains a reference to the data (data is stored in separate memory area)
Allocated on stack	Allocated on heap using the new keyword
Assigned as copies	Assigned as references
Default behavior is pass by value	Passed by reference
== and != compare values	== and != compare the references, not the values
simple types, structs, enums	classes



Concept of Boxing and Unboxing

- Boxing and unboxing enable value types to be treated as objects.
- Boxing:
  - int number = 2000;
  - object obj = number;
- Now both the integer variable and the object variable exist on the stack. However, the value of the object resides on the heap.
- Boxing is implicit conversion

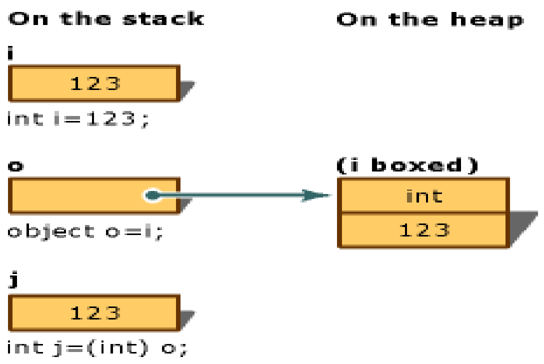
When value type is converted to ref type Boxing takes place

Note: Value types, including both struct types and built-in types, such as int, can be converted to and from the type object.

Concept of Boxing and Unboxing

➢Unboxing Conversions

- int number = 2000;
- object obj = number;
- int anothernumber = (int)obj;



Concept of Arrays

- An array is a data structure that contains a number of variables called the elements of the array.
  - All of the array elements must be of the same type, which is called the element type of the array.
  - An array can be a single-dimensional array, a multidimensional array, or a jagged array (Array of Arrays).
  - Array types are reference type derived from the abstract base type System.Array.

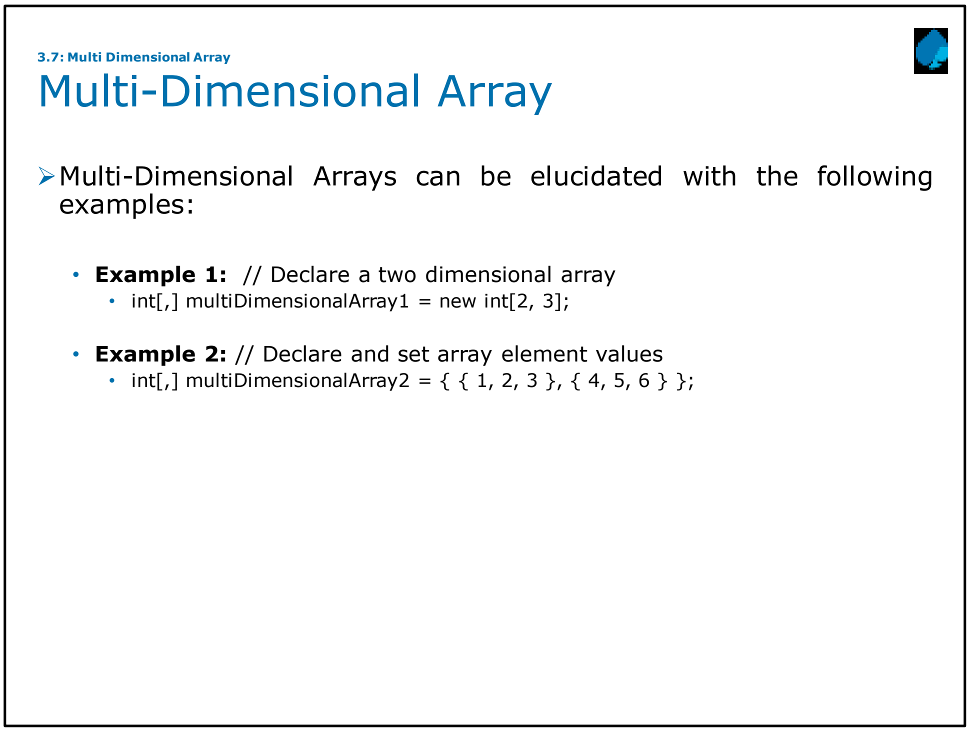
Arrays:

C# arrays are zero indexed. That is, the array indexes start at zero.

Array elements can be of any type, including an array type.

Single Dimensional Array

- Single Dimensional Arrays exhibit the following features:
  - They declare a single-dimensional array of five integers.
    - **Example:** int[] array1 = new int[5];
  - It is an array that stores string elements.
    - **Example:** string[] stringArray = new string[6];
  - It declares and sets array element values.
    - **Example:** int[] array2 = new int[] { 1, 3, 5, 7, 9 };
  - The alternative syntax is as follows:
    - **Example:** int[] array3 = { 1, 2, 3, 4, 5, 6 };



3.7: Multi Dimensional Array

Multi-Dimensional Array

- Multi-Dimensional Arrays can be elucidated with the following examples:
  - **Example 1:** // Declare a two dimensional array
    - int[,] multiDimensionalArray1 = new int[2, 3];
  - **Example 2:** // Declare and set array element values
    - int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

Arrays:

Examples of defining Arrays:

Single-Dimensional Array:

Example 1: int[] numbers = new int[5] {1, 2, 3, 4, 5};

Example 2: string[] names = new string[3] {"Matt", "Joanne", "Robert"};

Example 3: int[] numbers = {1, 2, 3, 4, 5};

Example 4: string[] names = {"Matt", "Joanne", "Robert"};

Multidimensional Array:

Example 1: int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };

Example 2: string[,] siblings = new string[2, 2] { {"Mike","Amy"}, {"Mary","Albert"} };

Example 3: int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };

Example 4: string[,] siblings = new string[,] { {"Mike","Amy"}, {"Mary","Ray"} };

Example 5: int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };

Example 6: string[,] siblings = { {"Mike", "Amy"}, {"Mary", "Albert"} };

Jagged Arrays

- Jagged array is an array whose elements are arrays.
- The elements of a jagged array can be of different dimensions and sizes.
- A jagged array is sometimes called an “array of arrays”.
- Following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

  Example:

  ```
  int[][] jaggedArray = new int[3][];
  ```

Jagged Arrays

- Let us see an example on jagged arrays:

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];

jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Array of Elements:
Each of the elements is a single-dimensional array of integers
The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.
It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size.

Explain to participants how to iterate via jagged array elements.

Demo

- Demo with Arrays in C#

Concept of Nullable Types

- Nullable types represent value-type variables that can be assigned the value of null.
- You cannot create a nullable type based on a reference type: Reference types already support the null value.
- A nullable type can represent the normal range of values for its underlying value type, plus an additional null value.
- Example:
  - A Nullable<Int32>, pronounced "Nullable of Int32", can be assigned any value from -2147483648 to 2147483647, or it can be assigned the null value.
  - A Nullable<bool> can be assigned the values true or false, or null.
- The ability to assign null to numeric and Boolean types is particularly useful when dealing with databases and other data types containing elements that may not be assigned a value.
- Example:
  - A Boolean field in a database can store the values true or false, or it may be undefined.

Concept of Nullable Types

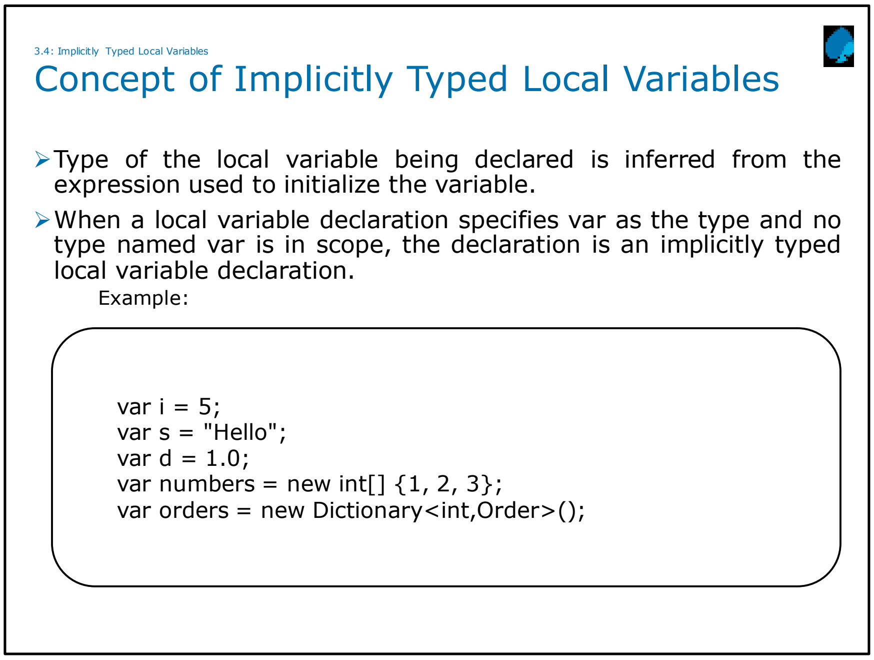
- The syntax T? is shorthand for System.Nullable<T>, where T is a value type

```
static void Main()
{
    int? num = null;
    if (num.HasValue == true)
    {
            Console.WriteLine("num = " + num.Value);
    }
    else
    {
            Console.WriteLine("num = Null");
    }
}
```

While demonstrating the data types, please demonstrate assigning value to and accepting values from the user for variables of type float and decimal. Explain Convert, Parse and TryParse methods.

Demo

- Demo on Data Types, Boxing and Unboxing, and Nullable Types



Concept of Implicitly Typed Local Variables

- Type of the local variable being declared is inferred from the expression used to initialize the variable.
- When a local variable declaration specifies var as the type and no type named var is in scope, the declaration is an implicitly typed local variable declaration.

  Example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

Implicitly Typed Local Variables:
In an implicitly typed local variable declaration, the type of the local variable being declared is inferred from the expression used to initialize the variable. When a local variable declaration specifies var as the type and no type named var is in scope, the declaration is an implicitly typed local variable declaration.
Example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

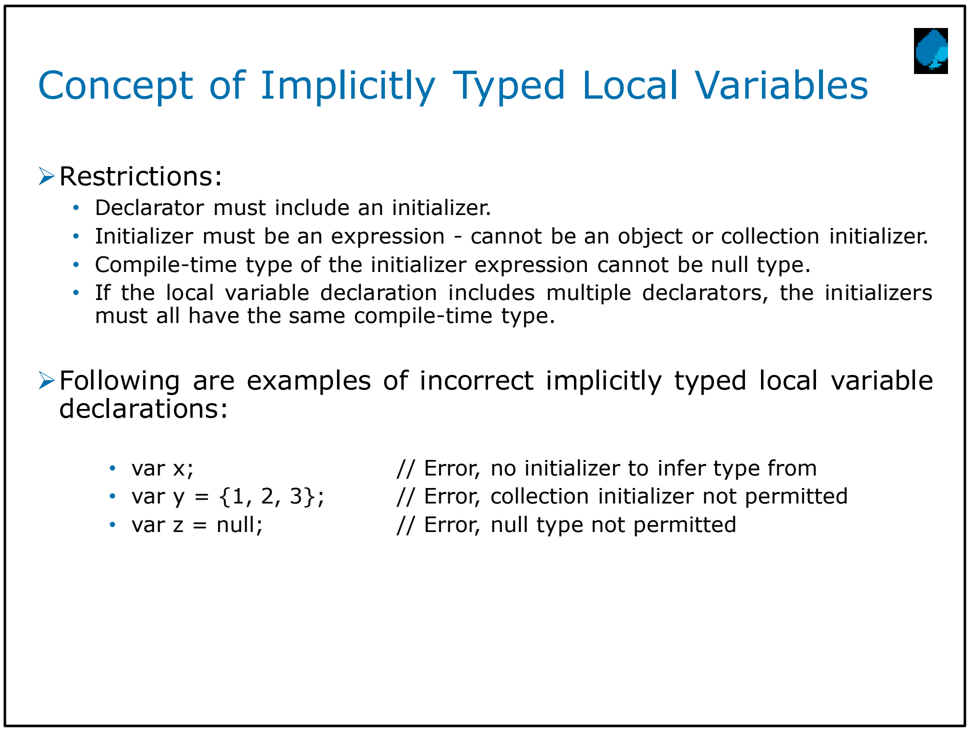
The implicitly typed local variable declarations shown above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Concept of Implicitly Typed Local Variables

- The above declarations are equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```



Concept of Implicitly Typed Local Variables

- Restrictions:
  - Declarator must include an initializer.
  - Initializer must be an expression - cannot be an object or collection initializer.
  - Compile-time type of the initializer expression cannot be null type.
  - If the local variable declaration includes multiple declarators, the initializers must all have the same compile-time type.
- Following are examples of incorrect implicitly typed local variable declarations:

```
• var x;                // Error, no initializer to infer type from
• var y = {1, 2, 3};    // Error, collection initializer not permitted
• var z = null;         // Error, null type not permitted
```

Implicitly Typed Local Variables:

A local variable declarator in an implicitly typed local variable declaration is subject to the following restrictions:

The declarator must include an initializer.

The initializer must be an expression. The initializer cannot be an object or collection initializer by itself, but it can be a new expression that includes an object or collection initializer.

The compile-time type of the initializer expression cannot be the null type.

If the local variable declaration includes multiple declarators, the initializers must all have the same compile-time type.

For reasons of backward compatibility, when a local variable declaration specifies var as the type and a type named var is in scope, the declaration refers to that type. However, a warning is generated to call attention to the ambiguity. Since a type named var violates the established convention of starting type names with an upper case letter, this situation is unlikely to occur.

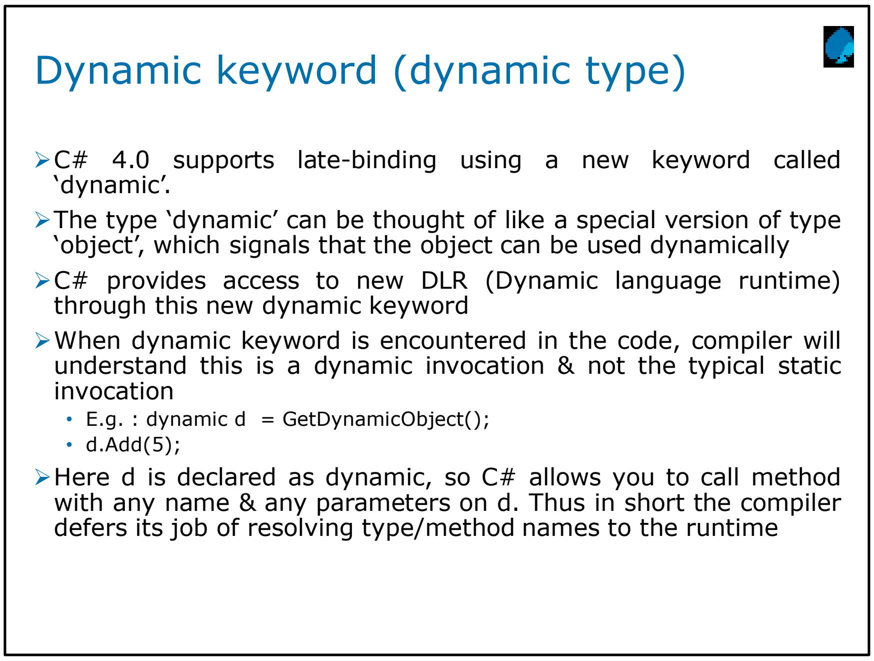
Concept of Implicitly Typed Local Variables

- the type of n is inferred to be int

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers)
     Console.WriteLine(n);
```

Implicitly Type Local Variables:
The for-initializer of a for statement and the resource-acquisition of a using statement can be an implicitly typed local variable declaration. Likewise, the iteration variable of a foreach statement may be declared as an implicitly typed local variable, in which case the type of the iteration variable is inferred to be the element type of the collection being enumerated.
In the following example, the type of n is inferred to be int, the element type of numbers.

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers)
          Console.WriteLine(n);
```



Dynamic keyword (dynamic type)

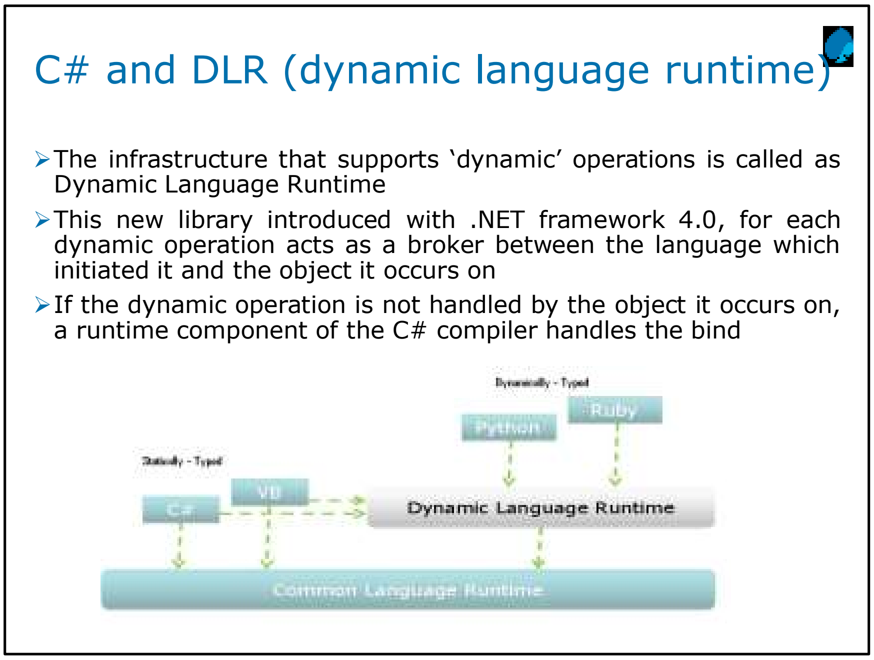
- C# 4.0 supports late-binding using a new keyword called ‘dynamic’.
- The type ‘dynamic’ can be thought of like a special version of type ‘object’, which signals that the object can be used dynamically
- C# provides access to new DLR (Dynamic language runtime) through this new dynamic keyword
- When dynamic keyword is encountered in the code, compiler will understand this is a dynamic invocation & not the typical static invocation
  - E.g. : dynamic d = GetDynamicObject();
  - d.Add(5);
- Here d is declared as dynamic, so C# allows you to call method with any name & any parameters on d. Thus in short the compiler defers its job of resolving type/method names to the runtime

C# now supports late-binding. C# has always been strongly typed & continues to be in version 4.0 as well. But Microsoft thought providing the concept of dynamic binding in C#. It would make life easier for C# developers while they communicate with systems using COM objects or other non .NET based objects.

C# 4.0 introduces a new static type named dynamic. This resolves issues with type binding especially if the receiving object is a non .NET based object. Here whatever you do with a dynamic type will take effect only at runtime i.e. it will be resolved at runtime.

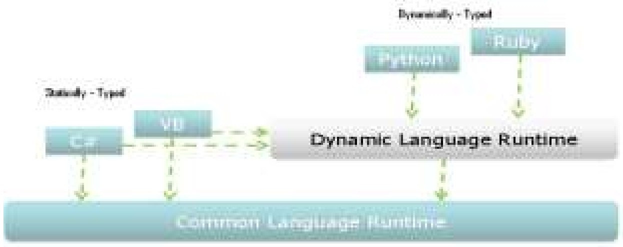
```
E.g. : dynamic d  = GetDynamicObject();
                    d.Add(5);
```

C# allows you to call a method with any name and any arguments on d because it is of type dynamic. At runtime the actual object that d refers to will be examined to determine what it means to “call Add with an int” on it. This approach using dynamic type is much similar to using object type, but there are subtle differences between the two.



C# and DLR (dynamic language runtime)

- The infrastructure that supports 'dynamic' operations is called as Dynamic Language Runtime
- This new library introduced with .NET framework 4.0, for each dynamic operation acts as a broker between the language which initiated it and the object it occurs on
- If the dynamic operation is not handled by the object it occurs on, a runtime component of the C# compiler handles the bind



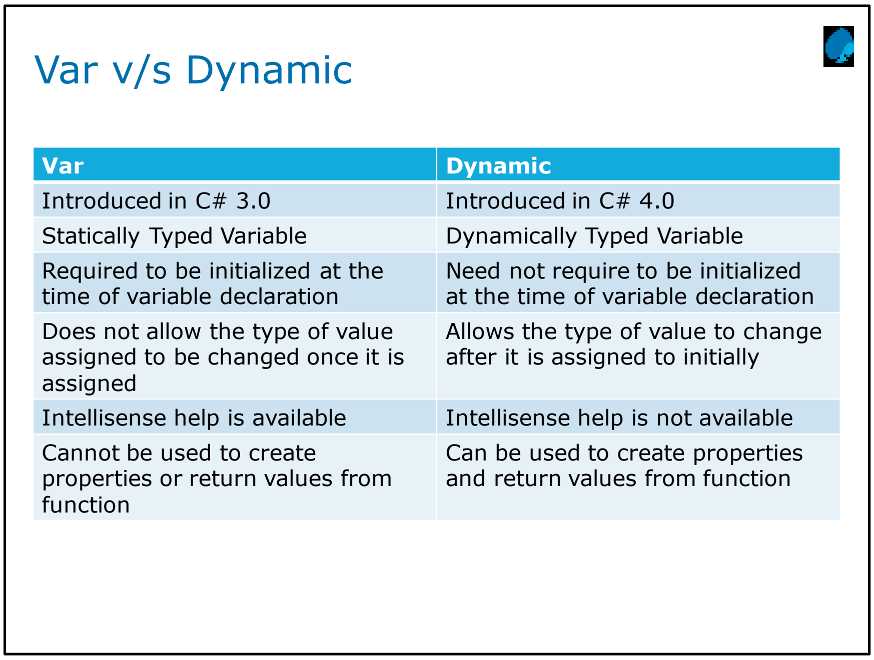
Dynamic binding provides a unified approach to selecting operations dynamically. With dynamic binding developer does not need to worry about whether a given object comes from e.g. COM, IronPython, the HTML DOM or reflection; operations can uniformly be applied to it and the runtime will determine what those operations mean for that particular object.

This affords enormous flexibility, and can greatly simplify the source code, but it does come with a significant drawback: Static typing is not maintained for these operations. A dynamic object is assumed at compile time to support any operation, and only at runtime will an error occur if it was not so.

Dynamic Programming

```
E.g. - : dynamic d  = new MyDynamicObject();
                  d.Add(5);
```

By declaring d to be of type dynamic, the code that consumes the MyDynamicObject instance effectively opts out of compile-time checking for the operations d participates in. Use of dynamic means "I don't know what type this is going to be, so I don't know what methods or properties there are right now. Compiler, please let them all through and then figure it out when you really have an object at run time.” So the call to Add compiles even though the compiler doesn’t know what it means. Then at run time, the object itself is asked what to do with this call to Add.



Var v/s Dynamic

Var	Dynamic
Introduced in C# 3.0	Introduced in C# 4.0
Statically Typed Variable	Dynamically Typed Variable
Required to be initialized at the time of variable declaration	Need not require to be initialized at the time of variable declaration
Does not allow the type of value assigned to be changed once it is assigned	Allows the type of value to change after it is assigned to initially
Intellisense help is available	Intellisense help is not available
Cannot be used to create properties or return values from function	Can be used to create properties and return values from function

1. Introduced in
   - Var was introduced in C# 3.0
   - Dynamic was introduced in C# 4.0
2. Type inference of variables
   - Var is statically typed variable. It gives strongly typed variable, whose type is known at compile time.
   - Dynamic is dynamically typed variable. Type will be inferred at run time.
3. Initialization of variables
   - Var type of variable needs the initialization at the time of declaration of variable, otherwise it will encounter compile time error – "Implicitly-typed local variables must be initialized"
   - Dynamic type of variable do not require initialization, while declaring variables

4. Changing type of value assigned
    - Var does not allow the type of value assigned to be changed once it is assigned.

      For Example :

      ```
      var val = 10;
      val = “variable”;   //Error – Cannot
      ```

      implicitly convert type string to int
    - Dynamic allows the type of value to change after it is assigned to initially

      For Example :

      ```
      dynamic dynamicVal = 10;
      dynamicVal = “variable”
      ```

      In this case this code will not only compile but also work at run-time. This is because, at run-time, first 10 will be assigned to dynamicVal and it will be of type System.Int32. Then after that variable will be assigned to dynamicVal and it will become System.String.
5. Intellisense help
    - Intellisense help is available for the var type of variable, since it’s type is known at compile time, the compiler has all information related to the type.
    - Intellisense help is not available for dynamic type of variable since the type is unknow until run-time.
6. Restrictions on the usage
    - Var type cannot be used to create properties or return values from function
    - Dynamic type can be used to create properties or return values from function

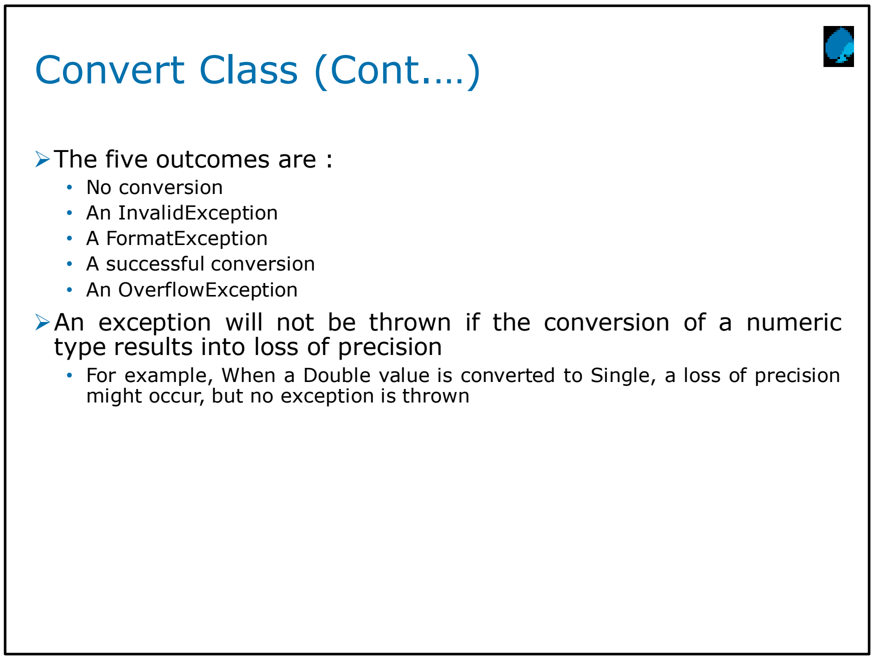
Convert Class

- Convert class provides static methods to convert a base type to another base type
- The supported base data types are Boolean, Char, SByte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime and String
- A conversion method exists to convert every base type to every other base type
- The actual call to a particular conversion method can produce one of five outcomes depending on the value of the base type at run-time and the target base type

The five outcomes are :

- No conversion – This occurs when trying to the type value in same type. In this case, the method simply returns an instance of original type.
  - For example, by calling Convert.ToInt32(Int32), here trying to covert in Int32 and value is also Int32
- An InvalidException – This occurs when a particular conversion is not supported. An InvalidException thrown for the following conversions:
  - Conversions from Char to Boolean, Single, Double, Decimal or DateTime
  - Conversions from Boolean, Single, Double, Decimal or DateTime or Char
  - Conversions from DateTime to any other type except String
  - Conversions from any other type, except String, to DateTime
- A FormatException – This occurs when trying to convert a string value to any other base type fails because the string is not in proper format. The exception is thrown for the following conversions:
  - A String to be converted to Boolean value
  - A String to be converted to Char value, which consists of multiple characters

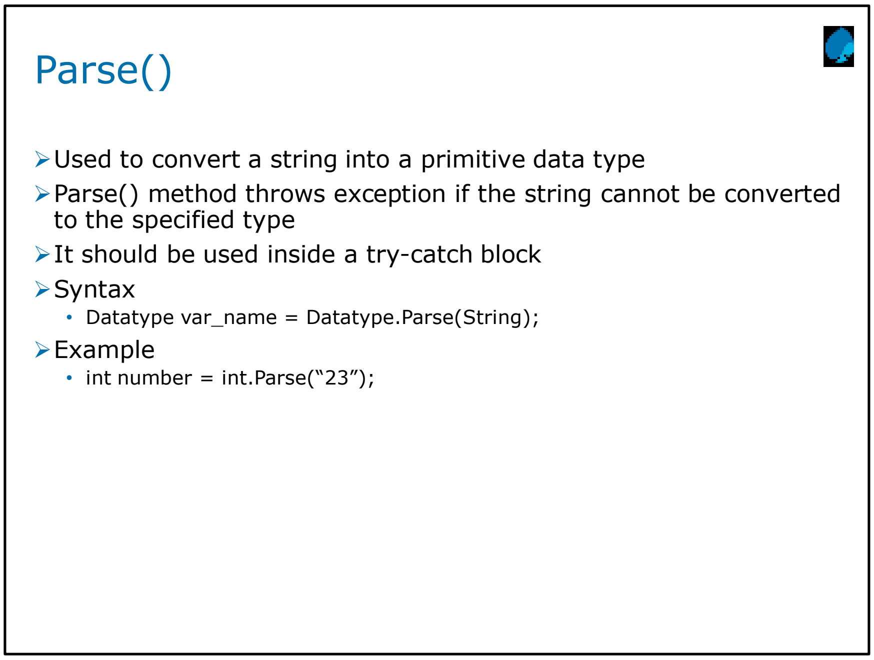
- A String to be converted to any numeric type, which is not a valid number
- A String to be converted to DateTime is not recognized as a valid date and time value



Convert Class (Cont….)

- The five outcomes are :
  - No conversion
  - An InvalidException
  - A FormatException
  - A successful conversion
  - An OverflowException
- An exception will not be thrown if the conversion of a numeric type results into loss of precision
  - For example, When a Double value is converted to Single, a loss of precision might occur, but no exception is thrown

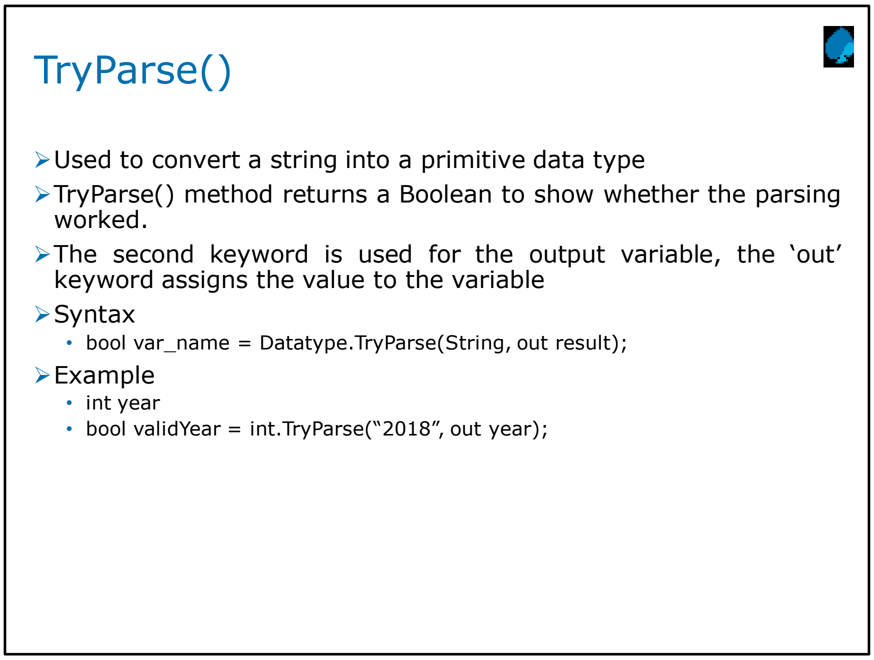
- A successful conversion – For conversions between two different base types, which are not listed in the previous outcomes, compatible with each other. All widening conversions as well as all narrowing conversions that do not result in a loss of data will succeed and the method will return a value of the targeted base type
- An OverflowException – This occurs when a narrowing conversion results in a loss of data.
  - For example, trying to covert a Int32 instance whose value is 10000 to a Byte type throws an OverflowException because 10000 is outside the range of the Byte Data Type



Parse()

- Used to convert a string into a primitive data type
- Parse() method throws exception if the string cannot be converted to the specified type
- It should be used inside a try-catch block
- Syntax
  - Datatype var_name = Datatype.Parse(String);
- Example
  - int number = int.Parse("23");

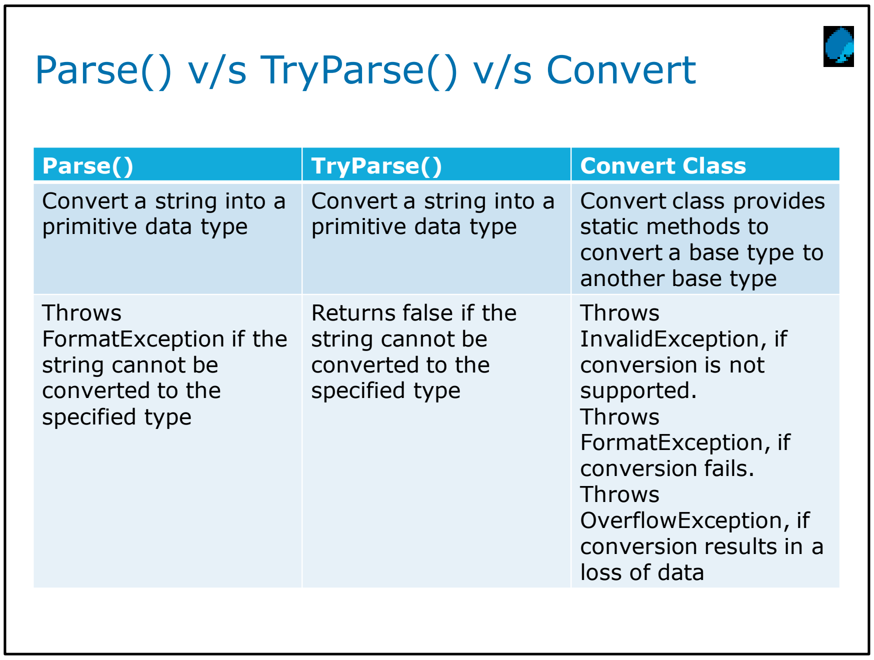
Parse() method ignores white space at the beginning and at the end of the string, but all other characters must be the characters that form the appropriate numeric type
Any whitespace within the characters that form the number cause an error



TryParse()

- Used to convert a string into a primitive data type
- TryParse() method returns a Boolean to show whether the parsing worked.
- The second keyword is used for the output variable, the ‘out’ keyword assigns the value to the variable
- Syntax
  - bool var_name = Datatype.TryParse(String, out result);
- Example
  - int year
  - bool validYear = int.TryParse(“2018”, out year);

TryParse() method ignores white space at the beginning and at the end of the string, but all other characters must be the characters that form the appropriate numeric type
Any whitespace within the characters that form the number cause an error

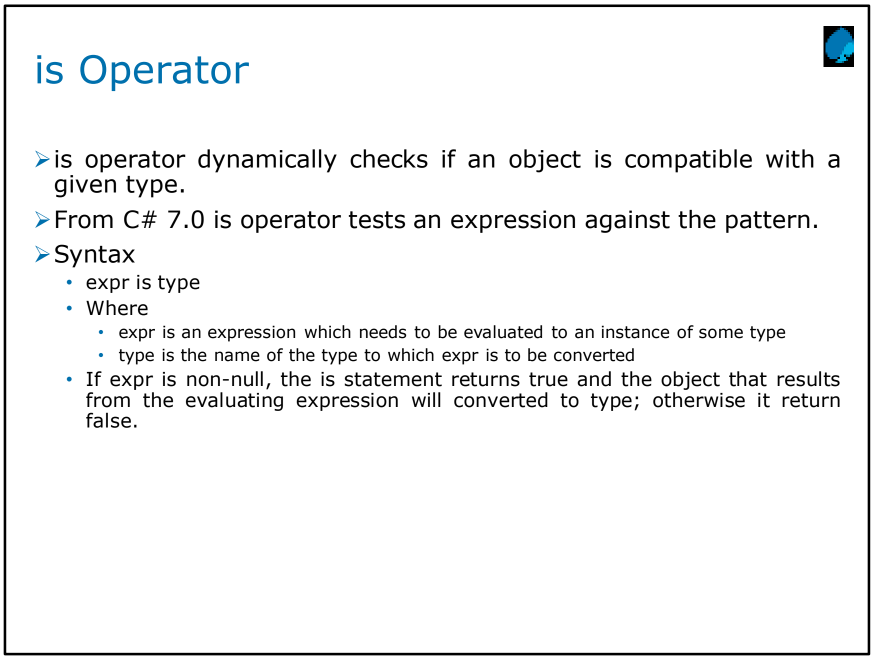


Parse() v/s TryParse() v/s Convert

Parse()	TryParse()	Convert Class
Convert a string into a primitive data type	Convert a string into a primitive data type	Convert class provides static methods to convert a base type to another base type
Throws FormatException if the string cannot be converted to the specified type	Returns false if the string cannot be converted to the specified type	Throws InvalidException, if conversion is not supported. Throws FormatException, if conversion fails. Throws OverflowException, if conversion results in a loss of data

Demo

- Demo on Parse() Method, TryParse() Method and Convert class



is Operator

- is operator dynamically checks if an object is compatible with a given type.
- From C# 7.0 is operator tests an expression against the pattern.
- Syntax
  - expr is type
  - Where
    - expr is an expression which needs to be evaluated to an instance of some type
    - type is the name of the type to which expr is to be converted
  - If expr is non-null, the is statement returns true and the object that results from the evaluating expression will converted to type; otherwise it return false.

Look for the below example :

```
public class Circle {
    int radius;
    public Circle(int r) { radius = r; }
}

Circle c =new Circle(10);
Object obj = c;
int num = (int) obj;
```

In this code there is no problem at compile time, but runtime is more suspicious. If the type of object in memory does not match the cast, the runtime will throw an InvalidCastException. We should be ready to catch this exception and handle it appropriately.

The is operator evaluates the type compatibility at run-time. It checks whether an object instance or the result of an expression can be converted to a specified type.

The is operator cannot be overloaded.
The is operator can be used to check reference, boxing and unboxing conversions, but not for user-defined conversions
The type compatibility of the expression will be evaluated at run-time, but the C# compiler will generate a warning during compilation when the is operator is used on types that are not compatible

For example :

```
public class Circle{
    int radius;
    public Circle(int r) { radius = r; }
}

public class Square{
    int length;
    public Square(int l) { length = l; }
}

public class IsTest{
    public static void Test(object o)
    {
        Circle c;
        Square s;
        if (o is Circle)
        {
            Console.WriteLine("o is Circle");
            c = (Circle)o;
        }
        else if (o is Square)
        {
            Console.WriteLine("o is Square");
            s = (Square)o;
        }
        else
        {
            Console.WriteLine("o is neither Circle nor Square");
        }
    }
    public static void Main(string[] args)
    {
        Circle c = new Circle(10);
        Square s = new Square(12);
        Test(c);
        Test(s);
```

```
                Test(“Passing string value instead of circle or square”);
        }
}
```

is Operator (Pattern Matching)

- Starting with C# 7.0, the is and switch statements support pattern matching.
- The is keyword supports the following patterns :
  - Type Pattern
  - Constant Pattern
  - var Pattern

is Operator (Type Pattern)

- When is operator using the type pattern, it performs pattern matching.
- In pattern matching is operator tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type
- Syntax
  - expr is type varname
    - expr is an expression that evaluates to some instance type
    - type is the name of the type to which the result of expr is to be converted
    - varname is the object to which the result of expr is converted if the is test returns true
  - The is expression returns true if expr is not null, any of the following is true:
    - expr is an instance of the same type as type
    - expr is an instance of a type that derives from type
    - expr has a compile-time type that is a base class of type, and expr has a run-time type that is a type or is derived from type
    - expr is an instance of a type that implements the type interface
- If expr returns true and is operator is used with an if statement, varname is assigned and has local scope within the if statement only

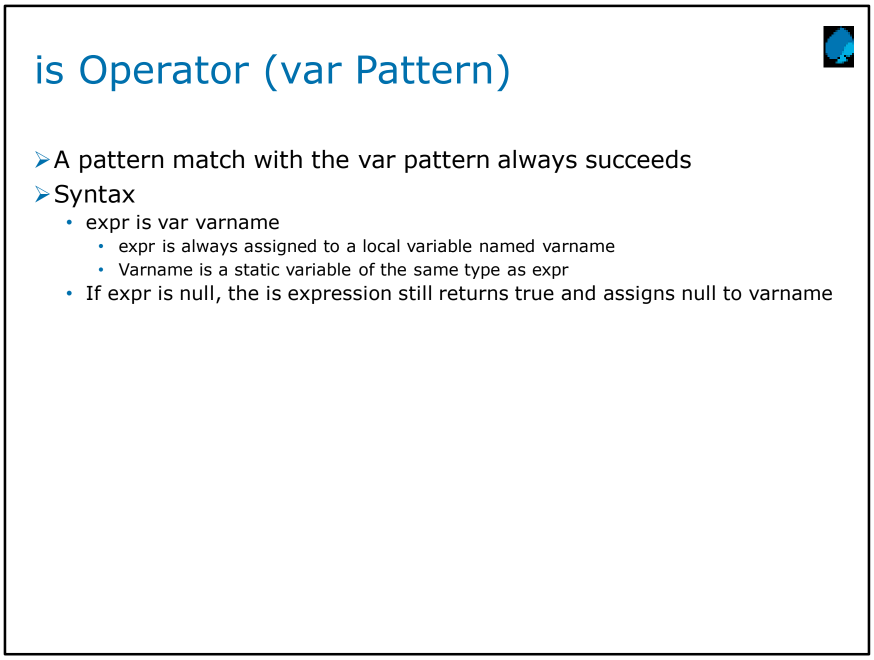
is Operator (Constant Pattern)

- When is operator performing pattern matching with the constant pattern, it tests whether an expression equals a specified constant.
- In C# 6.0 and earlier versions, the constant pattern is supported by the switch statement.
- Starting with C# 7.0, the it is supported by the is statement as well.
- Syntax
  - expr is constant
    - expr is the expression to evaluate
    - constant is the value to test for
      - constant can be any of the following constant expressions:

        A literal value

        The name of a declared const variable

        An enumeration constant
- If expr and constant are integral types, the C# equality operator determines whether the expression returns true (e.g. expr == constant)
- Otherwise the value of the expression is determined by a call to the static Object.Equals(expr, constant) method



is Operator (var Pattern)

- A pattern match with the var pattern always succeeds
- Syntax
  - expr is var varname
    - expr is always assigned to a local variable named varname
    - Varname is a static variable of the same type as expr
  - If expr is null, the is expression still returns true and assigns null to varname

as Operator

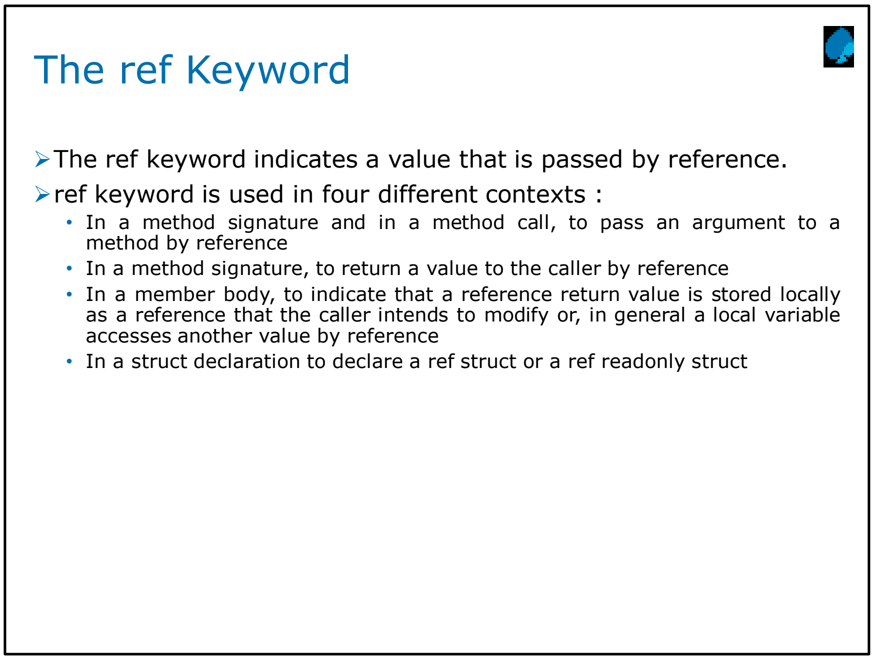
- The as operator do the same job of is operator, but the difference is instead of bool, it returns the object if they are compatible to that type, else it returns null
- The as operator performs certain types of conversions between compatible reference types or nullable types

is v/s as operator

is Operator	as Operator
➢ Used to check if the run-time type of an object is compatible with the given type or not	➢ Used to perform conversion between compatible reference types or nullable types
➢ Returns Boolean type value	➢ Returns non-Boolean type value
➢ Returns true if the given object is of the same type	➢ Returns the object when they are compatible with the given type
➢ Returns false if the given object is not of the same type	➢ Returns null if conversion is not possible
➢ Used only for reference type, boxing and unboxing conversions	➢ Used only for nullable, reference and boxing conversions

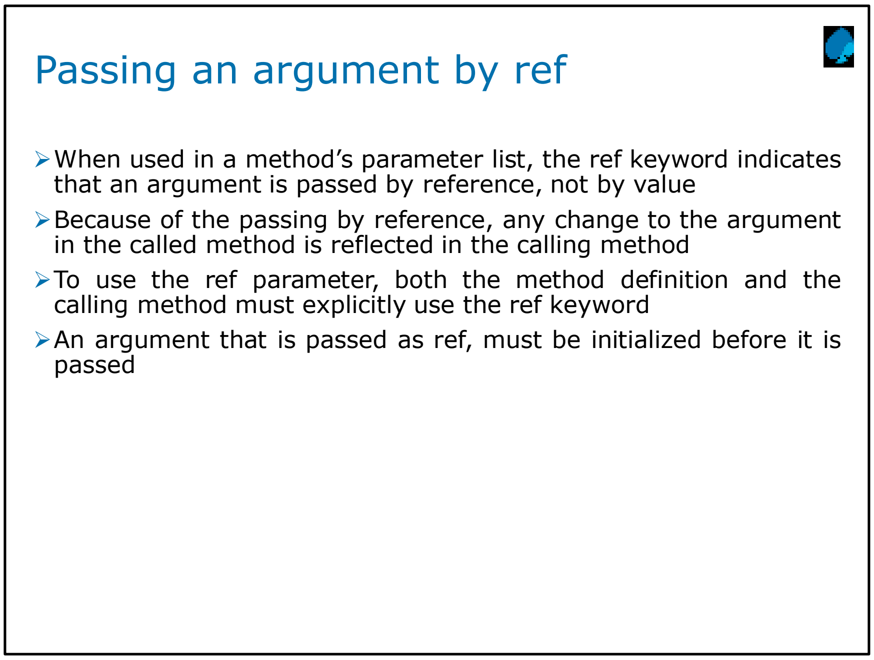
Demo

- Demo on is operator
- Demo on as operator



The ref Keyword

- The ref keyword indicates a value that is passed by reference.
- ref keyword 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, in general a local variable accesses another value by reference
  - In a struct declaration to declare a ref struct or a ref readonly struct


Passing an argument by ref

- When used in a method’s parameter list, the ref keyword indicates that an argument is passed by reference, not by value
- Because of the passing by reference, any change to the argument in the called method is reflected in the calling method
- To use the ref parameter, both the method definition and the calling method must explicitly use the ref keyword
- An argument that is passed as ref, must be initialized before it is passed

The **ref** parameters are input/output parameters.
That means, they can be used for passing a value to a function as well as for getting back a value from a function.
We create a **ref** parameter by preceding the parameter data type with a **ref** modifier.
Whenever, a **ref** parameter is passed, a reference is passed to the function.

Passing an argument by ref

- Output of the below program would be 105, since the ref parameter acts as both input and output.

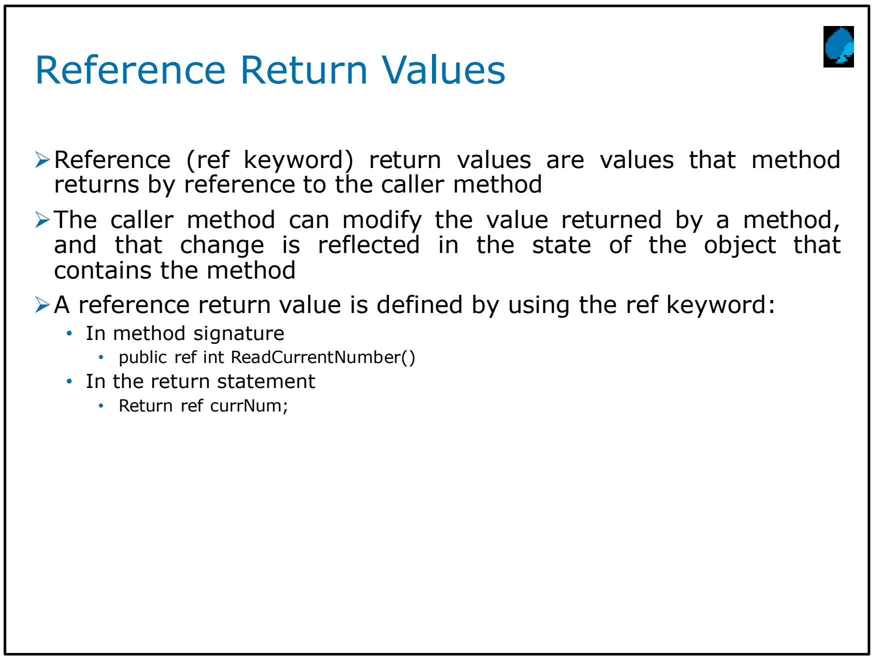
```
static void Mymethod(ref int Param1)
{
      Param1=Param1 + 100;
}
```

```
static void Main()
{
      int myValue=5;
      MyMethod(ref myValue);
      Console.WriteLine(myValue);
}
```

- Output of the below program would be “Hello from Method”, since the ref parameter acts as both input and output.

```
static void Hello(ref string message)
{
      message = "Hello from Method";
}
```

```
static void Main()
{
      string msg=“Hi”;
      MyMethod(ref msg);
      Console.WriteLine(msg);
}
```



Reference Return Values

- Reference (ref keyword) return values are values that method returns by reference to the caller method
- The caller method can modify the value returned by a method, and that change is reflected in the state of the object that contains the method
- A reference return value is defined by using the ref keyword:
  - In method signature
    - public ref int ReadCurrentNumber()
  - In the return statement
    - Return ref currNum;

For the caller method to modify the object’s state, the reference return value must be stored to variable that is explicitly defined as a ref local
The called method may declare the return value as ref readonly to return the value by reference, and enforce that the calling code cannot modify the returned value

Ref locals

- A ref local variable is used to refer the values using return ref
- A ref local variable cannot be initialized to a non-ref return value
- Any changes made to the value of the ref local are reflected in the state of the object whose method returned by the value by reference
- Define ref local by using the ref keyword before the variable declaration, as well as immediately before the method call that returns the value by reference.
  - ref int number = ref ReadCurrentNumber();

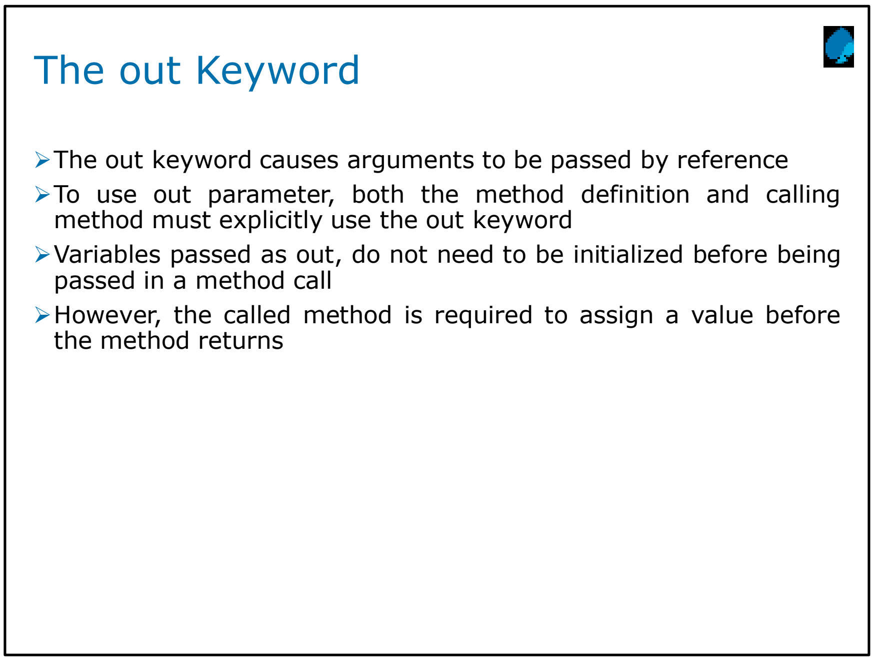
The ref keyword must be used in both places, or the compiler will generate error : “Cannot initialize a by-reference variable with a value”

Ref readonly locals

- A ref readonly local is used to refer to values returned by the method or property that has ref readonly in its signature and uses return ref
- A ref readonly variable combines the properties of a ref local variable with a readonly variable: it is an alias to the storage it's assigned to, and it cannot be modified

Ref struct types

- Adding the ref modifier to a struct declaration defines that instances of that type must be allocated on stack
- ref struct types are allocated on stack with several rules that compiler enforces :
  - ref struct type cannot be assigned to a variable of type object, dynamic or any interface type
  - ref struct type cannot implement interface
  - ref struct cannot be declared as a member of a class or another struct
  - Cannot declare ref struct variables as a local variable in async method
  - Cannot declare ref struct variables as a local variable in iterators
  - Cannot capture ref struct variable in lambda expressions or local functions
- These restrictions if not followed while using ref struct, will promote it to the managed heap



The out Keyword

- The out keyword causes arguments to be passed by reference
- To use out parameter, both the method definition and calling method must explicitly use the out keyword
- Variables passed as out, do not need to be initialized before being passed in a method call
- However, the called method is required to assign a value before the method returns

The **out** parameters are ‘output only’ parameters. That means, they can only return a value from a function.
We create an **out** parameter by preceding the parameter data type with an **out** modifier.
Whenever an **out** parameter is passed only an unassigned reference is passed to the function.

The **out** modifier should precede the parameter being passed even in the calling part.
The **out** parameters cannot be used within the function before assigning a value to it.
A value should be assigned to the **out** parameter before the method returns.

The out Keyword

- Output of the above program is 100, since the value of the out parameter is passed back to the calling part.

static void Mymethod(out int Param1)
{
    Param1=100;
}
	static void Main()
	{
	    int myValue=5;
	    MyMethod(out myValue);
	    Console.WriteLine(myValue);
	}

ref v/s out

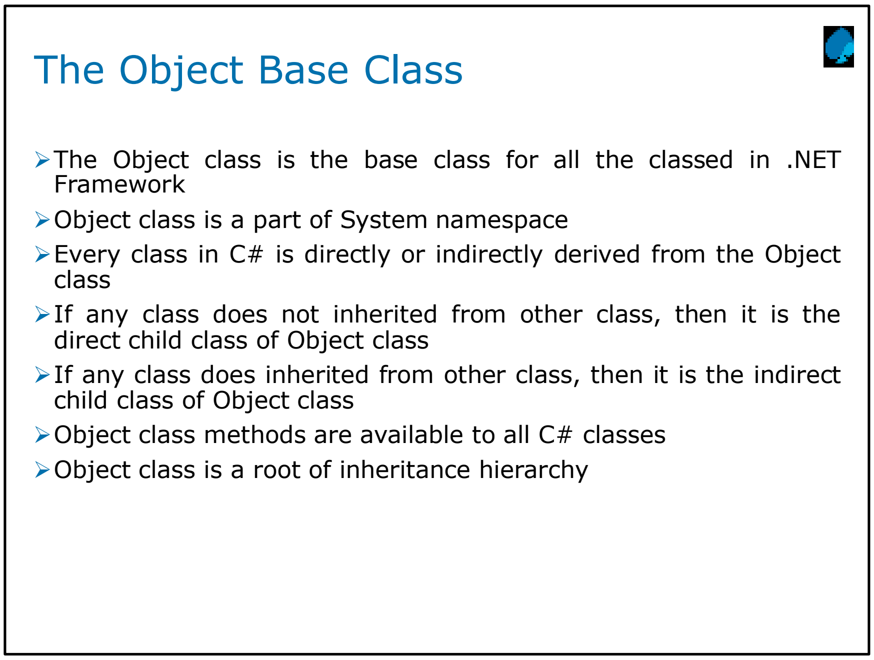
ref	out
The parameter must be initialized first before it is passed to method as a ref	It is not compulsory to initialize parameter before it is passed to method as an out
It is not required to assign or initialize the parameter which is passed as ref before returning to the calling method	A called method is required to assign or initialize a value of a parameter which is passed as out before returning to the calling method
Passing a parameter by ref is useful when the called method also needed to modify the parameter	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method
It is not compulsory to initialize a parameter before leaving the calling method	A parameter value must be initialized before leaving the calling method
Parameters can be used bi-directionally	Parameters can be used unidirectional

ref and out

- Both ref and out are treated differently at run-time and they are treated the same at the compile time
- Properties are not variables; therefore it cannot be passed as an out or ref parameter
- You can’t use the out keyword for the following kinds of methods:
  - Async methods, which you define by using the async modifier
  - Iterator methods, which include yield return or yield break statement

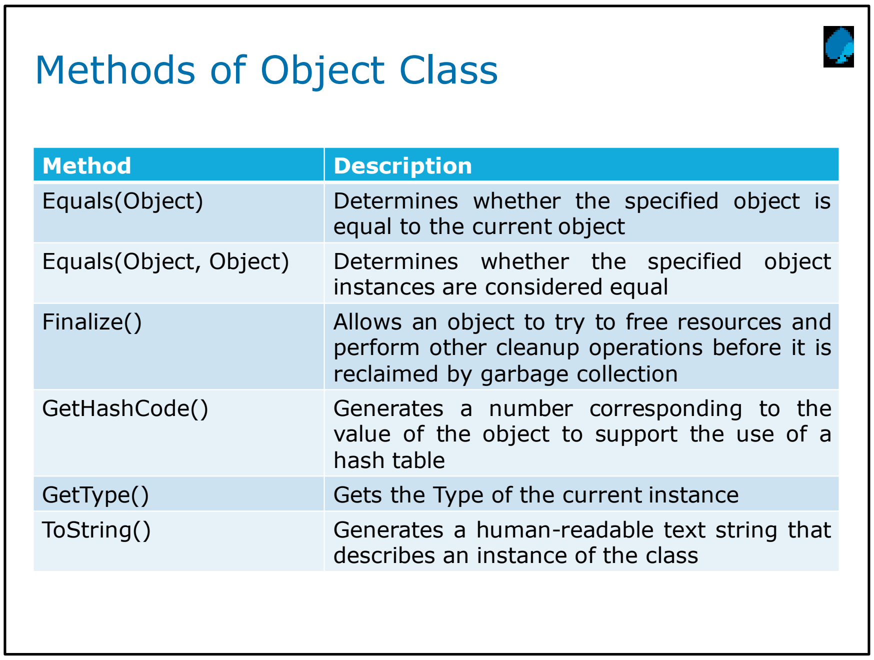
Demo

- ➢Demo Ref Keyword
- ➢Demo Out Keyword



The Object Base Class

- ➢The Object class is the base class for all the classed in .NET Framework
- ➢Object class is a part of System namespace
- ➢Every class in C# is directly or indirectly derived from the Object class
- ➢If any class does not inherited from other class, then it is the direct child class of Object class
- ➢If any class does inherited from other class, then it is the indirect child class of Object class
- ➢Object class methods are available to all C# classes
- ➢Object class is a root of inheritance hierarchy

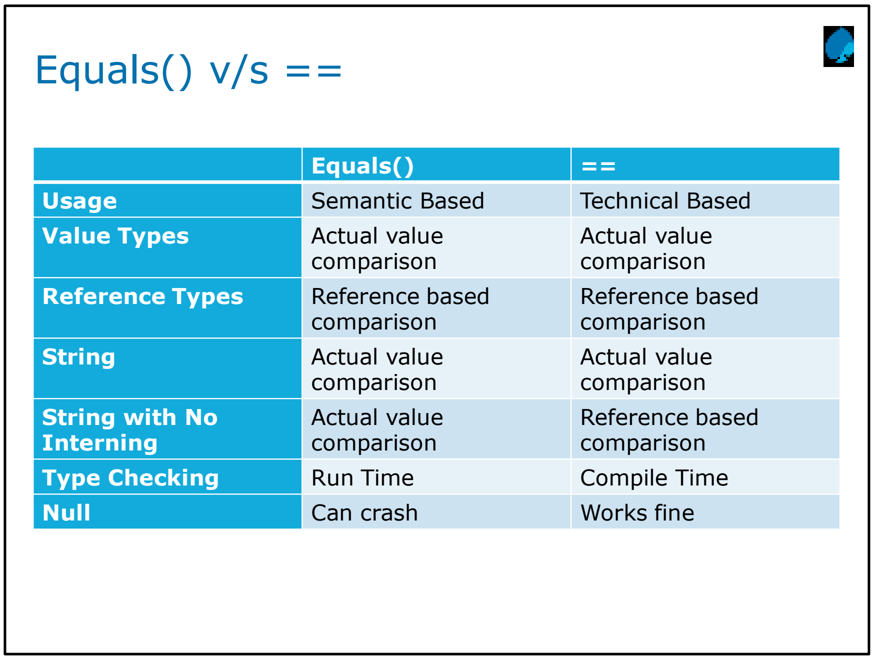


Methods of Object Class

Method	Description
Equals(Object)	Determines whether the specified object is equal to the current object
Equals(Object, Object)	Determines whether the specified object instances are considered equal
Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection
GetHashCode()	Generates a number corresponding to the value of the object to support the use of a hash table
GetType()	Gets the Type of the current instance
ToString()	Generates a human-readable text string that describes an instance of the class

Demo

- Demo Object Class



Equals() v/s ==

	Equals()	==
Usage	Semantic Based	Technical Based
Value Types	Actual value comparison	Actual value comparison
Reference Types	Reference based comparison	Reference based comparison
String	Actual value comparison	Actual value comparison
String with No Interning	Actual value comparison	Reference based comparison
Type Checking	Run Time	Compile Time
Null	Can crash	Works fine

"==" is a C# operator while "Equals" is a polymorphic method. In other words "==" is a language feature while "Equals" is an object oriented programming feature which follows polymorphism.

Now comparison is of two types one is purely based on content and reference, means computer based comparison and other is based on semantics. Semantics means the actual meaning of a thing. For example 1 <> 70 numerically (technically) but 1 $ = 70 Rs in real world semantically.

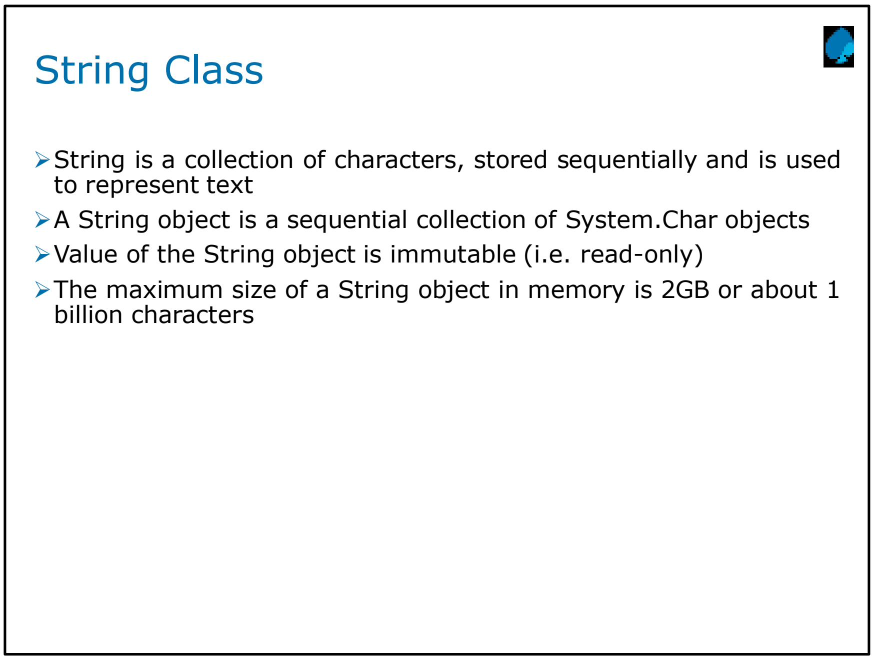
Some more examples:-
Technically: - 1 is equal to 1.
Semantically: - 1 Dollar is not equal to 1 Rs.
Technically: - "Destination " word is not equal to "Last Stop".
Semantically: - "Destination" means same as "Last Stop".
So technical comparison is computer based while semantic comparison is business based or we can say there is some kind of domain rule for comparison purpose.

So now when to use "==" and when to use "Equals":-

If you are looking for technical comparison then use “==” and most of the time “==” suffices as developers mostly do technical comparison.
If you are comparing semantically then you need over the “equals” with the semantic comparison logic and you need to use “equals” method while comparing.

Demo

- Demo Equals() and ==



String Class

- String is a collection of characters, stored sequentially and is used to represent text
- A String object is a sequential collection of System.Char objects
- Value of the String object is immutable (i.e. read-only)
- The maximum size of a String object in memory is 2GB or about 1 billion characters

Object of the String will be created by using following methods :

By assigning the string literal to String variable
By using String class constructor
By using String concatenation operator (+)
By retrieving a property or calling a method that returns a string
By calling a formatting method to convert a value or an object to its string representation

String Class Properties

Property	Description
Chars[Int32]	Gets a Char object at a specified position in the current String object
Length	Gets the number of characters in the current String object

String Class Methods

Method	Description
Compare(String, String)	Compares two string objects and returns an integer that indicates their relative position in the sort order
Concat(String[])	Concatenates the elements of a specified String array
Contains(String)	Returns a value indicating whether a specified substring occurs within this string
EndsWith(String)	Determines whether the end of this string instance matches the specified string
Equals(String)	Determines whether the current instance and another specified String object have the same value
IndexOf(Char)	Reports the zero-based index of the first occurrence of the specified Unicode character in this string.

String Class Methods (Cont….)

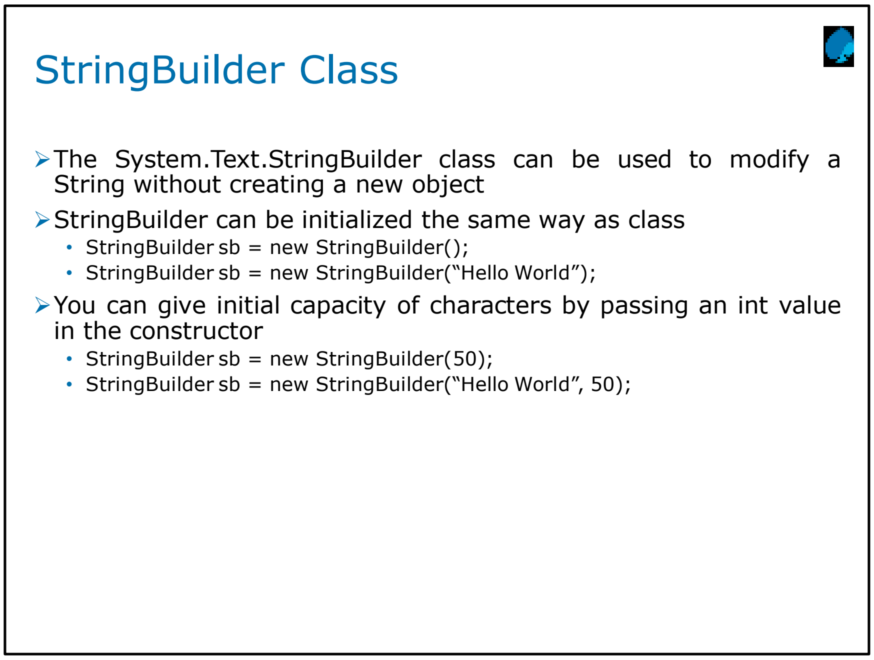
Method	Description
IndexOf(Char, Int32)	Reports the zero-based index of the first occurrence of the specified Unicode character in this string. The search starts at a specified character position
Insert(Int32, String)	Returns a new string in which a specified string is inserted at a specified index position in this instance
IsNullOrEmpty(String)	Indicates whether the specified string is null or an empty string ("")
Join(String, String[])	Concatenates all the elements of a string array, using the specified separator between each element
LastIndexOf(Char)	Reports the zero-based index position of the last occurrence of a specified Unicode character within this instance

String Class Methods (Cont….)

Method	Description
Replace(String, String)	Returns a new string in which all occurrences of a specified string in the current instance are replaced with another specified string
Split(Char[])	Splits a string into substrings that are based on the characters in an array
StartsWith(String)	Determines whether the beginning of this string instance matches the specified string
SubString(Int32)	Retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string
ToCharArray()	Copies the characters in this instance to a Unicode character array
ToLower()	Returns a copy of this string converted to lowercase

String Class Methods (Cont….)

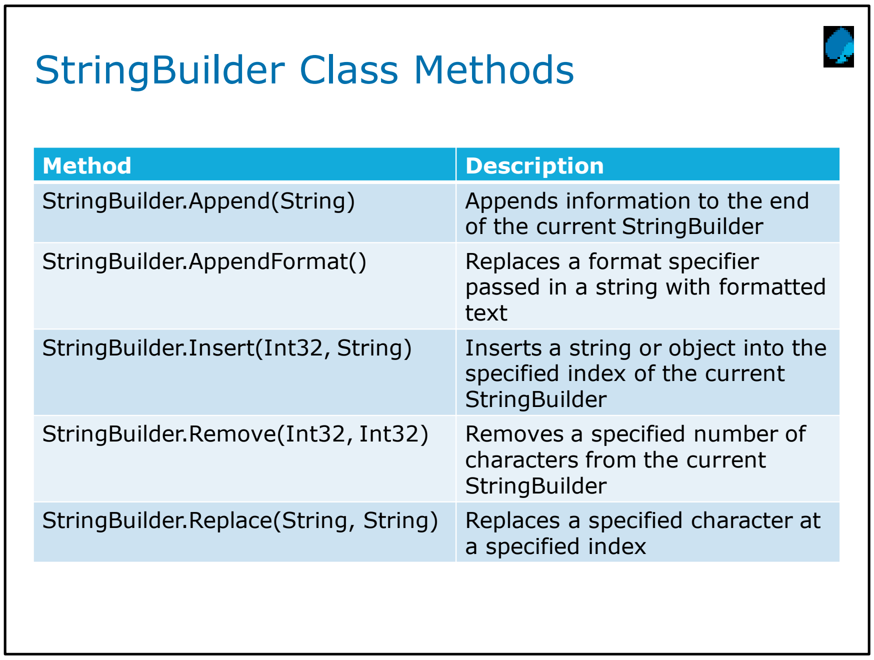
Method	Description
ToUpper()	Returns a copy of this string converted to uppercase
Trim()	Removes all leading and trailing white-space characters from the current String object



StringBuilder Class

- The System.Text.StringBuilder class can be used to modify a String without creating a new object
- StringBuilder can be initialized the same way as class
  - StringBuilder sb = new StringBuilder();
  - StringBuilder sb = new StringBuilder("Hello World");
- You can give initial capacity of characters by passing an int value in the constructor
  - StringBuilder sb = new StringBuilder(50);
  - StringBuilder sb = new StringBuilder("Hello World", 50);

The String object is immutable
When you use any method of in System.String class, it creates a new String object in memory, which requires a new allocation of space for that new object
It'll increase overhead associated with creating new String object can be costly
The System.Text.StringBuilder class can be used to modify a String without creating a new object

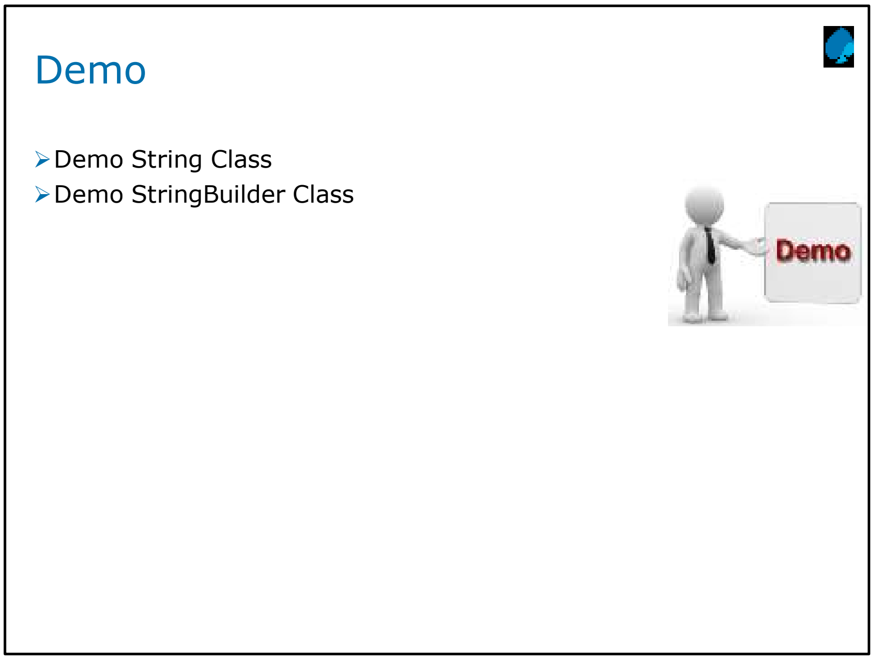


StringBuilder Class Methods

Method	Description
StringBuilder.Append(String)	Appends information to the end of the current StringBuilder
StringBuilder.AppendFormat()	Replaces a format specifier passed in a string with formatted text
StringBuilder.Insert(Int32, String)	Inserts a string or object into the specified index of the current StringBuilder
StringBuilder.Remove(Int32, Int32)	Removes a specified number of characters from the current StringBuilder
StringBuilder.Replace(String, String)	Replaces a specified character at a specified index

String v/s StringBuilder

String	StringBuilder
The String object is immutable	The System.Text.StringBuilder object is mutable
Once we create a string object, we cannot modify the value of the string object in the memory	Once we create StringBuilder object we can perform any operation that appears to change the value without creating new instance for every time
Any operation that appears to modify the string, it will discard the old value and it will create new instance in memory to hold the new value	It can be modified in any way and it doesn't require creation of new instance

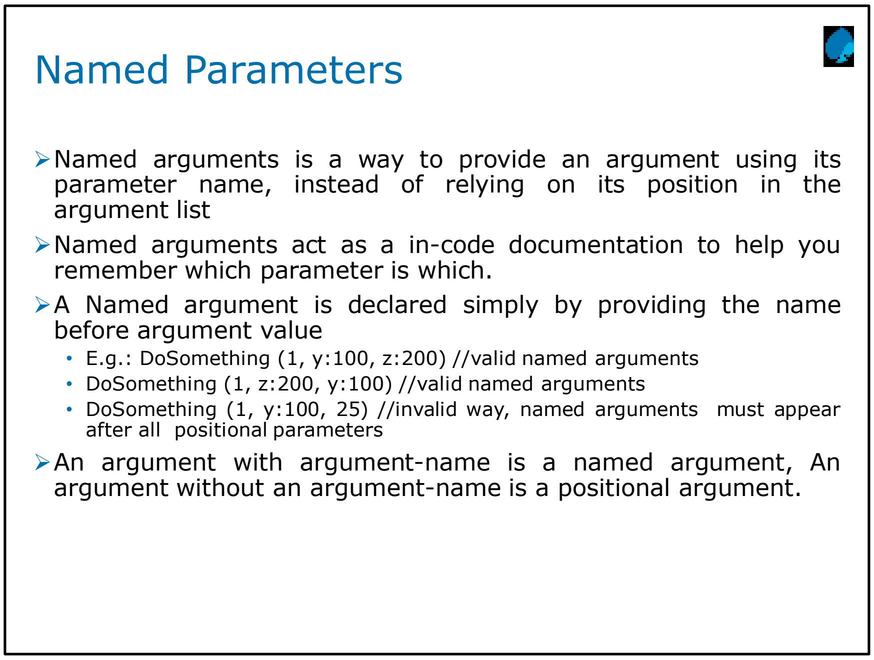


Demo

- ➢ Demo String Class
- ➢ Demo StringBuilder Class

Optional Parameters

- Optional parameters allows you to omit arguments in method invocation
- They become very handy in cases where parameter list is too long
  - E.g. : In case of COM method calls
- A parameter is declared optional simply by providing a default value for it
- Order of parameters is important, mandatory parameters should be declared first in the parameter list
  - E.g. : public void DoSomething(int x, int y = 5, int z = 10)
- Here y and z are optional parameters and can be omitted in calls:
  - DoSomething(1, 2, 3); //ordinary call to DoSomething
  - DoSomething(1, 2); //omitting z, equivalent to DoSomething (1, 2, 10);
  - DoSomething(1); // omitting y and z, equivalent to DoSomething (1, 5,10)
- C# does not permit you to omit arguments between commas as in DoSomething (1, , 3). This could lead to highly unreadable code
- Instead any argument can be passed by name

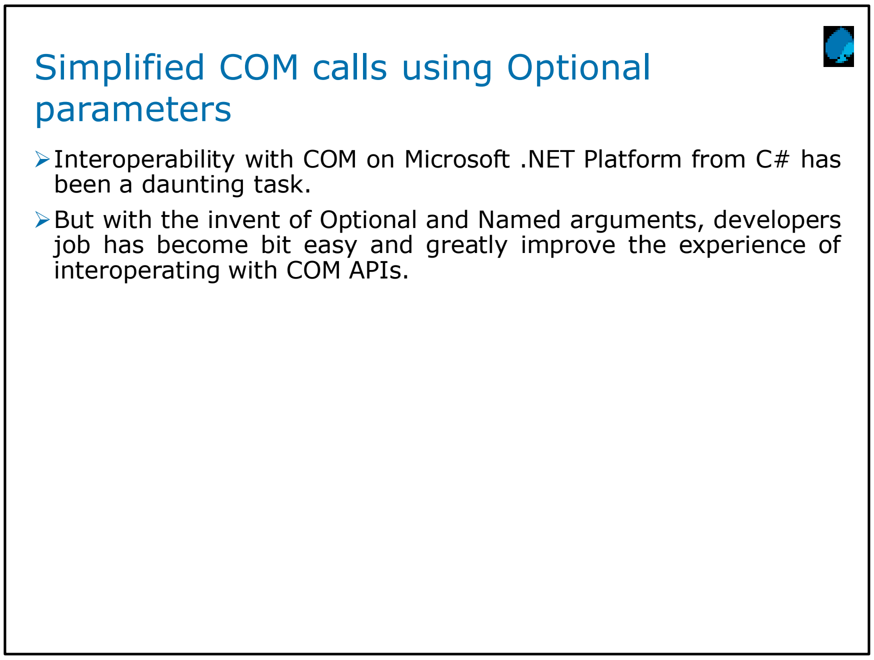


Named Parameters

- Named arguments is a way to provide an argument using its parameter name, instead of relying on its position in the argument list
- Named arguments act as a in-code documentation to help you remember which parameter is which.
- A Named argument is declared simply by providing the name before argument value
  - E.g.: DoSomething (1, y:100, z:200) //valid named arguments
  - DoSomething (1, z:200, y:100) //valid named arguments
  - DoSomething (1, y:100, 25) //invalid way, named arguments must appear after all positional parameters
- An argument with argument-name is a named argument, An argument without an argument-name is a positional argument.

Demo

- Optional Parameters and Named Parameters



Simplified COM calls using Optional parameters

- Interoperability with COM on Microsoft .NET Platform from C# has been a daunting task.
- But with the invent of Optional and Named arguments, developers job has become bit easy and greatly improve the experience of interoperating with COM APIs.

Simplified COM Calls with Optional Parameters and Named Arguments Interoperability with COM on Microsoft .Net Platform from C# has been a daunting task. But with the invent of Optional and Named arguments, developers job has become bit easy and greatly improve the experience of interoperating with COM APIs.

Omit REF keyword at COM call sites

- COM uses a different programming model where it uses a lot of reference parameters to attain performance benefit.
- Actually, a COM method call will not modify its parameters even when they are passed by reference.
- So declaring temporary variables and passing them as COM arguments seems to be unnecessary.
- In COM, compiler allows to declare method call passing the arguments by value
- And it automatically generates the necessary temporary variables to hold the values in order to pass them by reference.
- It will also discard their values after the call returns

Omitting ref
Since COM uses a different programming model, it usually uses a lot of reference parameters because of a perceived performance benefit. Actually in common case, a COM method call will not modify its parameters even when passed by reference. In this case, it seems unnecessary for a caller to declare temporary variables for all these arguments and pass them by reference. Specifically for COM methods, the compiler allows to declare the method call passing the arguments by value and will automatically generate the necessary temporary variables to hold the values in order to pass them by reference and will discard their values after the call returns.
For e.g.:

```
object filename = “Test.docx”;
object missing  = Missing.Value;
document.SaveAs(ref.filename,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing);
```

Can now be written like this :

```
document.SaveAs(“Test.docx”,
                    Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value,
                    Missing.Value, Missing.Value, Missing.Value);
```

Omitting REF: Example

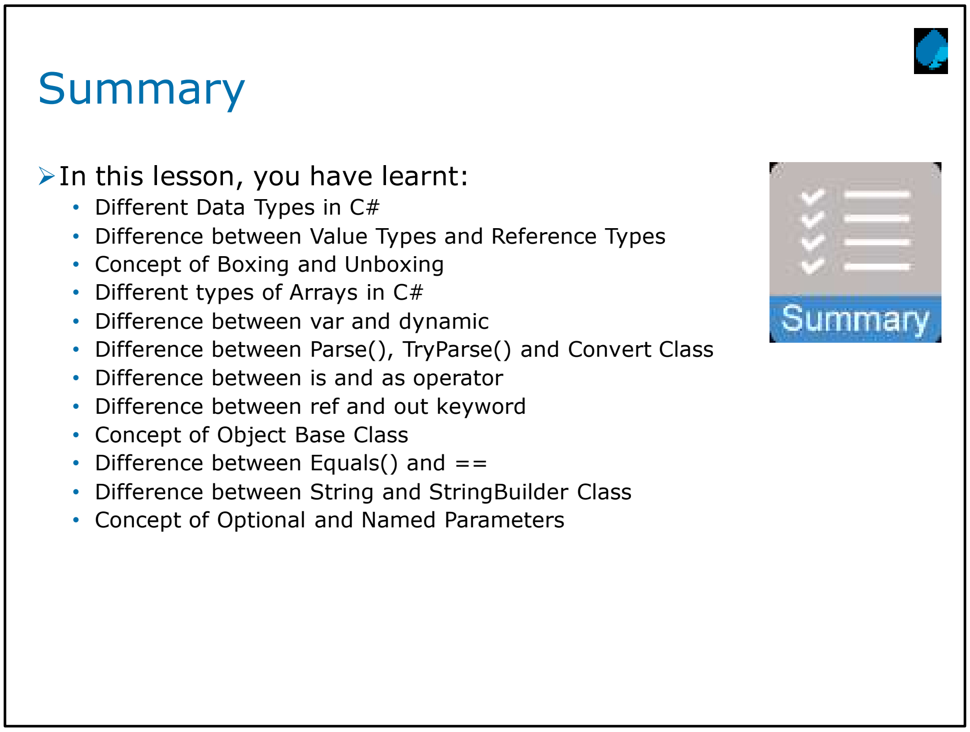
```
Word.Document document = new Word.Document();
object filename = “Test.docx”;
object missing  = Missing.Value;
document.SaveAs(ref filename,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing,
                    ref missing, ref missing, ref missing);
```

Can now be written as :

```
document.SaveAs(FileName : “Test.docx”);
```

Demo

- Simplified COM Calls using Optional Parameters
- Omitting REF with COM calls



Summary

➢ In this lesson, you have learnt:
- Different Data Types in C#
- Difference between Value Types and Reference Types
- Concept of Boxing and Unboxing
- Different types of Arrays in C#
- Difference between var and dynamic
- Difference between Parse(), TryParse() and Convert Class
- Difference between is and as operator
- Difference between ref and out keyword
- Concept of Object Base Class
- Difference between Equals() and ==
- Difference between String and StringBuilder Class
- Concept of Optional and Named Parameters

Review Questions

- Question 1: How are Value Types different from Reference Types?
- Question 2: What is Boxing and Unboxing in C#?
- Question 3: What are Jagged Arrays?
- Question 4: What is the difference between is and as operator?
- Question 5: Explain ref and out keyword in C#
- Question 6: Explain Object Class
- Question 7: What is the difference between Parse(), TryParse() and Convert Class?
- Question 8: How StringBuilder class is different than String Class?
- Question 9: Explain Optional and Named Parameter

Answers for the Review Questions:

Answer 1:
a) The variable of Value Type contains the value directly where as the variable of reference type contains reference to the data (data is stored in a separate memory area).

b) Value Types are allocated on stack and assigned as copies where as Reference Types are allocated on heap using the new keyword and are assigned as references.

c) Examples of Value Types: simple types, structs, enums.

Examples of Reference Types: classes, interfaces, String, object etc.

Answer 2:
Boxing is conversion of value type to reference type. Unboxing is conversion of reference type back to value type.

Answer 3:
Jagged Array is array of arrays.