



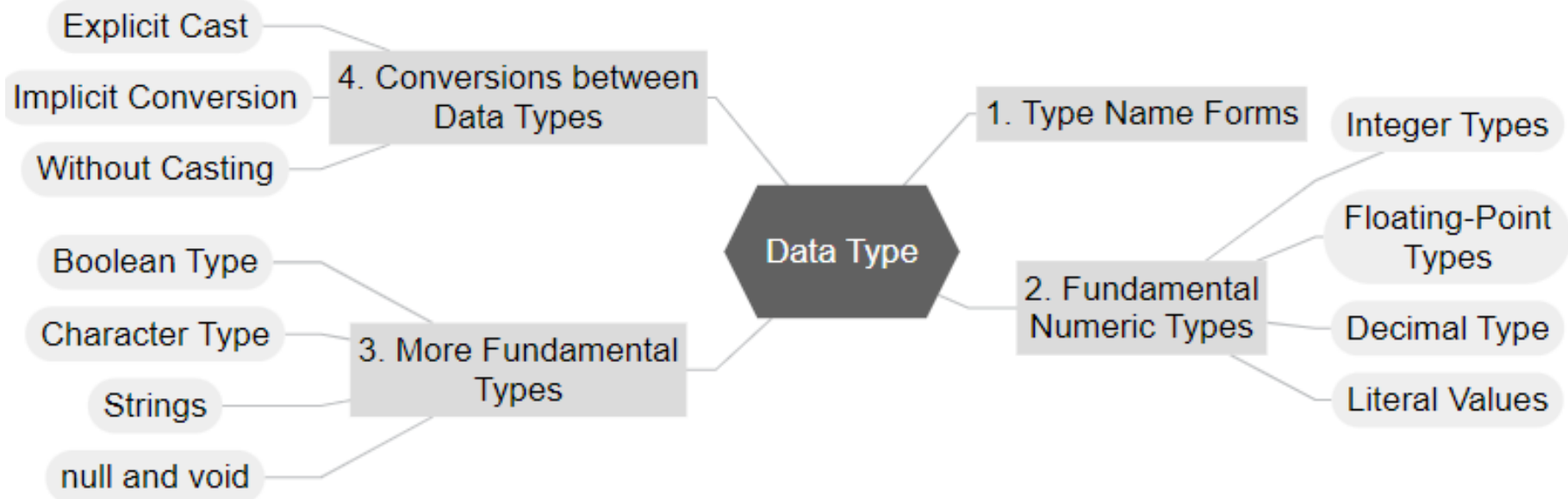
Fundamentals of Programming

Data Types

By: Võ Văn Hải

Email: vovanhai@ueh.edu.vn

Objectives



Type Name Forms

- ▶ All built-in types have essentially three name forms. (e.g., `string` type)
 - the full name (e.g., `System.String`)
 - the implied or simplified name (e.g., `String`), and
 - the keyword (e.g., `string`)
- ▶ The full name provides all the context and disambiguates the type from all other types.
- ▶ All the predefined types themselves are part of the Base Class Library (BCL)—the name given to the set of APIs that comprise the underlying framework. The full name of the types included in the BCL, therefore, is also the BCL name.
- ▶ Guidelines
 - DO use the C# keyword rather than the unqualified name when specifying a data type (e.g., `string` rather than `String`).
 - DO favor consistency rather than variety within your code.

Fundamental Numeric Types

Introduction

- ▶ The basic numeric types in C# have keywords associated with them. These types include integer types, floating-point types, and a special floating-point type called decimal to store large numbers with no representation error.
- ▶ This variety of each numeric type allows you to select a data type large enough to hold its intended range of values without wasting resources.

Fundamental Numeric Types

Integer Types

Type	Size	Range (Inclusive)	BCL Name	Signed	Literal Suffix
sbyte	8 bits	−128 to 127	System.SByte	Yes	
byte	8 bits	0 to 255	System.Byte	No	
short	16 bits	−32,768 to 32,767	System.Int16	Yes	
ushort	16 bits	0 to 65,535	System.UInt16	No	
int	32 bits	−2,147,483,648 to 2,147,483,647	System.Int32	Yes	
uint	32 bits	0 to 4,294,967,295	System.UInt32	No	U or u
Long	64 bits	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	System.Int64	Yes	L or l
ulong	64 bits	0 to 18,446,744,073,709,551,615	System.UInt64	No	UL or ul
nint	Signed 32-bit or 64-bit integer is a [depends on platform]	Depends on platform where the code is executing. Use sizeof(nint) to retrieve size.	System.IntPtr	Yes	
nuint	Unsigned 32-bit or 64-bit integer is a [depends on platform]	Depends on platform where the code is executing. Use sizeof(nuint) to retrieve size.	System.UIntPtr	No	

Fundamental Numeric Types

Floating-Point Types (float, double)

- ▶ Floating-point numbers have varying degrees of precision, and binary floating-point types can represent numbers exactly only if they are a fraction with a power of 2 as the denominator.
- ▶ The accuracy of a floating-point number is in proportion to the magnitude of the number it represents. A floating-point number is precise to a certain number of digits of precision, not by a fixed value such as ± 0.01 .
- ▶ There are at most 17 significant digits for a double and 9 significant digits for a float

Type	Size	Significant Digits	BCL Name	Significant Digits	Literal Suffix
float	32 bits	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	System.Single	7	F or f
double	64 bits	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	System.Double	15–16	D or d

Fundamental Numeric Types

Decimal Type

- ▶ C# also provides a decimal floating-point type with 128-bit precision. This type is suitable for financial calculations.
- ▶ Unlike binary floating-point numbers, the decimal type maintains exact accuracy for all denary numbers within its range. With the decimal type, therefore, a value of 0.1 is exactly 0.1.
- ▶ However, while the decimal type has greater precision than the floating-point types, it has a smaller range. Thus, conversions from floating-point types to the decimal type may result in overflow errors.
- ▶ Also, calculations with decimals are slightly (generally imperceptibly) slower.

Type	Size	Range (Inclusive)	BCL Name	Significant Digits	Literal Suffix
decimal	128 bits	1.0×10^{-28} to approximately 7.9×10^{28}	System.Decimal	28–29	M or m

Fundamental Types

Boolean Type (bool)

- ▶ Another C# primitive is a *Boolean* or conditional type, *bool*, which represents true or false in conditional statements and expressions. Allowable values are the keywords *true* and *false*. The BCL name for *bool* is *System.Boolean*.

```
bool ignoreCase = true;
string option = "/help";
// ...
int comparison = string.Compare(option, "/Help", ignoreCase);
bool isHelpRequested = (comparison == 0);
// Displays: Help Requested: True
Console.WriteLine($"Help Requested: {isHelpRequested}");
```


Fundamental Types

Character Type (char)

- ▶ A char type represents 16-bit characters whose set of possible values are drawn from the Unicode character set's UTF-16 encoding.
- ▶ A char is the same size as a 16-bit unsigned integer (ushort), which represents values between 0 and 65,535. However, char is a unique type in C# and code should treat it as such.
- ▶ The BCL name for char is System.Char.
- ▶ To construct a literal char, place the character within single quotes, as in 'A'. Allowable characters comprise the full range of ANSI keyboard characters, including letters, numbers, and special symbols.

```
Console.WriteLine('A');  
Console.WriteLine('\\');
```

Fundamental Types

Character Type (char) – Escape sequence

- Some characters cannot be placed directly into the source code and instead require special handling.

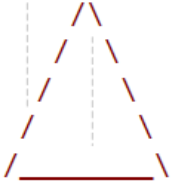
Escape Sequence	Character Name	Unicode Encoding
\'	Single quote	\u0027
\"	Double quote	\u0022
\\	Backslash	\u005C
\0	Null character	\u0000
\a	Alert (system beep)	\u0007
\b	Backspace	\u0008
\f	Form feed	\u000C
\n	Line feed (often referred to as a newline)	\u000A
\r	Carriage return	\u000D
\t	Horizontal tab	\u0009
\v	Vertical tab	\u000B
\uxxxx	Unicode character in hexadecimal	\u0029
\x[n][n][n]n	Unicode character in hex (first three placeholders are optional); variable-length version of \uxxxx	\u3A
\U00xxxxxx,	Unicode escape sequence for creating surrogate pairs (max supported sequence is \U0010FFFF)	\U00020100
\uxxxx\uxxxx	Unicode escape sequence for creating surrogate pairs beyond \U0010FFFF	\uD83D\uDE00

Fundamental Types

Strings


- ▶ A finite sequence of zero or more characters is called a string. The C# keyword is `string`, whose BCL name is `System.String`.
- ▶ The `string` type includes some special characteristics that may be unexpected to developers familiar with other programming languages.
- ▶ Strings include a “verbatim string” prefix character of `@`, support for string interpolation with the `$` prefix character.
 - Use the `@` symbol in front of a string to signify that a backslash should not be interpreted as the beginning of an escape sequence.
 - The resultant verbatim string literal does not reinterpret just the backslash character. Whitespace is also taken verbatim when using the `@` string syntax.

```
public static void Main(string[] args){  
    Console.WriteLine(@"Begin  
    End");  
}
```



Microsoft Visual Studio D

```
Begin  
End
```



Fundamental Types

Strings - Raw String Literals

- ▶ Raw string literals enable embedding any arbitrary text, including whitespace, newlines, additional quotes, and other special characters, without requiring escape sequences.

```
public static void Main(string[] args) {  
    Console.WriteLine(  
        ""Mama said, "Life was just a box of chocolates..." """);  
}
```

- ▶ Raw String Literals with Interpolation

```
public static void Main(string[] args) {  
    String firstName = "Teo", lastName = "Nguyen";  
    Console.WriteLine(  
        $"Hello, I'm {firstName}. {firstName} {lastName}");  
}
```

Fundamental Types

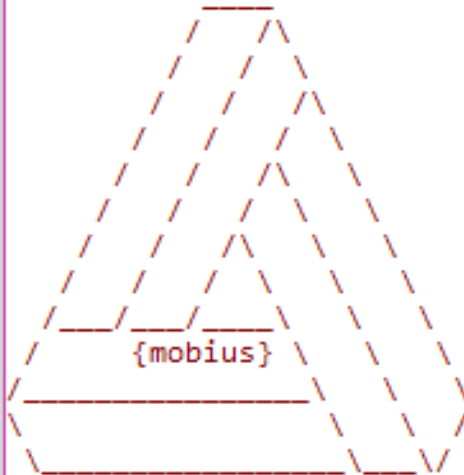
Raw String Literals with Interpolation

- ▶ However, an additional complexity is that the curly brace count into which you place the expression must match the number of dollar symbols (\$) you place at the beginning of the string. For example, two-dollar symbols would require each expression to be surrounded by two curly braces.

[illegible]

Microsoft Visual Studio Debug Console

Begin

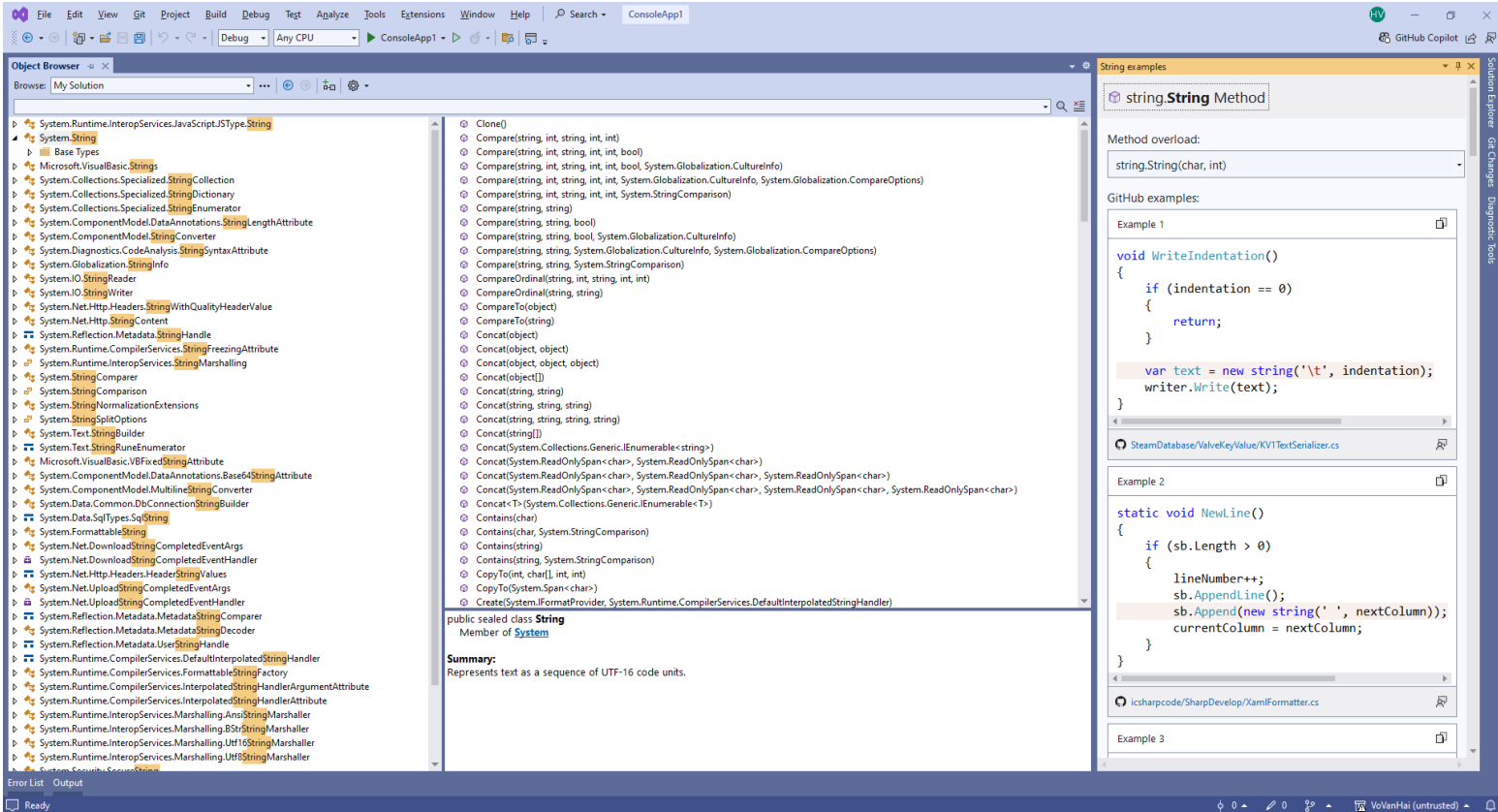


End

Fundamental Types

String methods

- Use Object navigator for observing



- Use MSDN <https://learn.microsoft.com/en-us/dotnet/api/system.string?view=net-8.0>

Fundamental Types

null type

- ▶ The *null* value, identified with the *null* keyword, indicates that the variable does not refer to any valid object.
- ▶ *null* can also be used as a type of “literal.” The value *null* indicates that a variable is set to nothing. Code that sets a variable to *null* explicitly assigns a “nothing” value. In fact, it is even possible to check whether a variable refers to a *null* value.
- ▶ Assigning the value *null* is not equivalent to not assigning it at all. In other words, a variable that has been assigned *null* has still been set, whereas a variable with no assignment has not been set and, therefore, will often cause a compile error if used prior to assignment.
- ▶ When we want to use a variable/reference that may be *null*, the nullable modifier (?) can be used in such a situation.

```
int? age;  
//...  
// Clear the value of age  
age = null;  
// ...
```

Fundamental Types

*The **void** “Type”*

- ▶ Sometimes the C# syntax requires a data type to be specified, but no data is passed.
 - For example, if no return from a method is needed, C# allows you to specify void as the data type instead.
- ▶ The **void** is used to indicate the absence of a type or the absence of any value altogether.
- ▶ The use of **void** as the return type indicates that the method is not returning any data and tells the compiler not to expect a value. **void** is not a data type per se but rather an indication that there is no data being returned.

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


Conversions between Data Types

Introduction

- ▶ Given the thousands of types predefined in .NET and the unlimited number of types that code can define, it is important that types support conversion from one type to another where it makes sense.
- ▶ Consider the conversion between two numeric types: converting from a variable of type long to a variable of type int.
 - A long type can contain values as large as 9,223,372,036,854,775,808; however, the maximum size of an int is 2,147,483,647.
 - As such, that conversion could result in a loss of data—for example, if the variable of type long contains a value greater than the maximum size of an int.
 - Any conversion that could result in a loss of data (such as magnitude and/or precision) or an exception because the conversion failed requires an explicit cast.
 - Conversely, a numeric conversion that will not lose magnitude and will not throw an exception regardless of the operand types is an implicit conversion.

Conversions between Data Types

Implicit Conversion

- ▶ In other instances, such as when going from an int type to a long type, there is no loss of precision, and no fundamental change in the value of the type occurs.
- ▶ In these cases, the code needs to specify only the **assignment operator**; the conversion is **implicit**. In other words, the compiler can determine that such a conversion will work correctly.

```
public static void Main(string[] args){  
    int intNumber = 31416;  
    long longNumber = intNumber;  
}
```

Conversions between Data Types

Explicit Cast

- ▶ In C#, you cast using the cast operator.
- ▶ By specifying the type, you would like the variable converted to within parentheses, you acknowledge that if an explicit cast is occurring, there may be a loss of precision and data, or an exception may result.
- ▶ The code in Listing 2.26 converts a long to an int and explicitly tells the system to attempt the operation.

```
public static void Main(string[] args){  
    long longNumber = 50918309109;  
    int intNumber = (int)longNumber; // (int) is a cast operator.  
    Console.WriteLine(intNumber);  
}
```

Conversions between Data Types

Explicit Cast - Checked and Unchecked Conversions

- Give a situation

```
public static void Main(string[] args)
{
    // int.MaxValue equals 2147483647
    int n = int.MaxValue;
    n = n + 1;
    Console.WriteLine(n);
}
```

C# Microsoft Visual Studio Debug Console

-2147483648

```
public static void Main(string[] args){
    unchecked
    {
        // int.MaxValue equals 2147483647
        int n = int.MaxValue;
        n = n + 1;
        Console.WriteLine(n);
    }
}
```

```
public static void Main(string[] args){
    checked
    {
        // int.MaxValue equals 2147483647
        int n = int.MaxValue;
        n = n + 1;
        Console.WriteLine(n);
    }
}
```

Exception Unhandled

System.OverflowException: 'Arithmetic operation resulted in an overflow.'

 Ask Copilot | [Show Call Stack](#) | [View Details](#) | [Copy Details](#) | [Start Live Share session](#)

► [Exception Settings](#)



Which is your choice? Why?

Conversions between Data Types

Type Conversion without Casting

- ▶ No conversion is defined from a string to a numeric type, so methods such as `Parse()` are required. Each numeric data type includes a `Parse()` function that enables conversion from a string to the corresponding numeric type.

```
public static void Main(string[] args) {  
    string text = $"{9.11E-31}";  
    float kgElectronMass = float.Parse(text);  
}
```

- ▶ *System.Convert* supports only a small number of types and is not extensible. It allows conversion from any of the types *bool*, *char*, *sbyte*, *short*, *int*, *long*, *ushort*, *uint*, *ulong*, *float*, *double*, *decimal*, *DateTime*, and *string* to any other of those types.
- ▶ All types support a *ToString()* method that can be used to provide a string representation of a type.

Conversions between Data Types

*Type Conversion without Casting – The **TryParse** method*

- ▶ All the numeric primitive types include a static **TryParse()** method.
- ▶ This method is similar to the **Parse()** method, except that instead of throwing an exception if the conversion fails, the **TryParse()** method returns false.

```
public static void Main(string[] args) {  
    double number;  
    string input;  
  
    Console.Write("Enter a number: ");  
    input = Console.ReadLine();  
    if (double.TryParse(input, out number)){  
        // Converted correctly, now use number  
        // ...  
    }  
    else{  
        Console.WriteLine(  
            "The text entered was not a valid number.");  
    }  
}
```

Exercises

- ▶ The Celsius scale is centigrade, 100 divisions separate the freezing point from the boiling point of water. On the Fahrenheit scale of Anglo-Saxons, these two points are 180 degrees apart. The Kelvin scale is an absolute scale used in science.
- ▶ Create a C# program to convert from degrees Celsius to Kelvin and Fahrenheit. Request the user the number of degrees celsius to convert them using the following conversion tables:
 - $\text{kelvin} = \text{celsius} + 273$
 - $\text{fahrenheit} = \text{celsius} \times 1.8 + 32$
 - Input
 - 33
 - Output
 - kelvin= 306
 - fahrenheit= 91

Exercises

- ▶ Create a program in C# for calculate the surface and volume of a sphere, given its radius.
 - $\text{surface} = 4 * \pi * \text{radius squared}$
 - $\text{volume} = 4 / 3 * \pi * \text{radius cubed}$
 - Input
 - 60
 - Output
 - Surface: 45238,93
 - Volume: 678584,1
- ▶ Write a program in C# that calculates the result of adding, subtracting, multiplying and dividing two numbers entered by the user.
 - In addition you should also calculate the rest of the division on the last line.
 - Input
 - 12
 - 3
 - Output
 - $12 + 3 = 15$
 - $12 - 3 = 9$
 - $12 \times 3 = 36$
 - $12 / 3 = 4$
 - $12 \bmod 3 = 0$



*Thank you
for Listening!*

