Lecture 23 — Library Functions

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 26, 2016

The system provides a number of classes and types (in various namespaces, like `System.Console`) for programmers' use.

We often ask students, as exercises, to implement functions like absolute value or exponentiation.

In practical situations, these functions are already available to us as static methods of specific classes.

Many of the built-in classes are in the `System` namespace.
 Or in a namespace enclosed within `System`, like `System.Console`.

Example: the `System.Math` class contains a number of different mathematical operations, including, but not limited to:

Function	Returns
Abs	Absolute Value
Cos	Cosine
Max	The larger of two values
Round	The closest integer

The usage is straightforward:

```
double z = Math.Pow( x, y );
```

The object is `Math`, and `Pow` is a static method.

There is no specific instance of `Math` created.

The following code:

```
double d = -3.14;
Console.WriteLine( Math.Abs( d ) );
Console.WriteLine( Math.Floor( d ) );
Console.WriteLine( Math.Round( Math.Abs( d ) ) );
```

... produces as output:

```
3.14
-4
3
```

In fact, an instantiation like `new Math()` makes no sense.
The class `Math` is declared as `static`.
Therefore such an instantiation cannot take place.

A `static` class may not be instantiated; there is only one copy of it.
Accordingly, no instance constructor may be created.

As you would expect, a static class may contain only static members (variables as well as methods).

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor.

Recall: a private method cannot be called from outside the class.

A private constructor prevents instantiation from outside the class. If the class contains no such instantiation, it will never be instantiated.

The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added.

The compiler ensures that instances of this class cannot be created.

Examining Value Types

It's not only the system-provided namespaces and classes that have functions available to use; even built-in types have some!
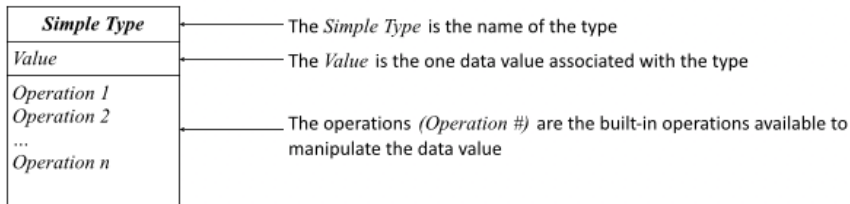
You may have wondered how a built-in value type can parse input.

In reality, the C# built-in value types are aliases for structured types.

For example, the `int` type is really a `struct` that defines:

- Constants for MinValue and MaxValue
- Methods for parsing, string conversion, comparison, and other operations common to all value types

Simple types can be represented by the following diagram:

Simple Type
Value
Operation 1   *Operation 2*   ...   *Operation n*

- The *Simple Type* is the name of the type
- The *Value* is the one data value associated with the type
- The operations *(Operation #)* are the built-in operations available to manipulate the data value

Different types will have different methods associated with them.

The char type defines a built-in type, but in C# it's also a struct called System.char that provides static methods to test characters:

Static Method	Checks if ch is ...
IsDigit(char ch)	A digit
IsLetter(char ch)	A letter
IsLetterOrDigit(char ch)	A letter or a digit
IsLower(char ch)	A lowercase letter
IsUpper(char ch)	An uppercase letter
IsWhiteSpace(char ch)	A white space character

(Don't attempt to memorize this table.)

When reading data in from the user with `Console.ReadLine()`, we have sometimes parsed it into a number type.

This has resulted in an Exception if the user entered some letters or other garbage data where a number was expected.

Though we now know how to handle an Exception with the try-catch statement, we can now check the input before parsing.

Because strings are just arrays of characters, we can use the `char` static methods on individual characters within a string.

```
String input = Console.ReadLine( );
bool validInput = false;
while( !validInput )
{
    for ( char c in input )
    {
        if( !char.isDigit( c ) )
        {
            break;
        }
    }
    validInput = true;
}
int parsedValue = int.Parse( input );
```

But what if the input is empty...?

The `string` type provides many methods to help you work with them:

Method	Returns
`bool s1.StartsWith( string s2 )`	true if s1 begins exactly with s2
`bool s1.EndsWith( string s2 )`	true if s1 ends exactly with s2
`int s1.IndexOf( char ch )`	the first position of ch within s1
`int s1.IndexOf( char ch, int pos )`	the first position of ch within s1 after (pos - 1)
`int s1.IndexOf( string s2 )`	the first position of s2 within s1
`int s1.IndexOf( string s2, int pos )`	the first position of s2 within s1 after (pos - 1)
`string s1.Substring( int pos )`	a copy of the substring from s1 starting at index pos and ending at the end of s1
`string s1.Substring( int pos, int len )`	a copy of the substring from s1 starting at index pos and ending `len` characters later

(Don't attempt to memorize this table, either.)

Here are some examples of working with the built-in string functions.
The string we are working with is `String s = "Lecture23"`.

Method Call	Returns
`s.StartsWith("L")`	true
`s.StartsWith("l")`	false
`s.StartsWith("Lect")`	true
`s.IndexOf("2")`	7
`s.Substring(1)`	`"ecture23"`
`s.Substring(2, 1)`	`"c"`

Formatting Methods of the `string` Class

The `string` class defines a number of interesting methods for formatting strings:

Static Method	Returns
`string s1.ToLower( )`	a copy of s1 using lowercase letters
`string s1.ToUpper( )`	a copy of s1 using uppercase letters
`string s1.Trim( )`	a copy of s1 with leading and trailing white space characters removed
`string s1.TrimStart( )`	a copy of s1 with leading white space characters removed
`string s1.TrimEnd( )`	a copy of s1 with trailing white space characters removed
`string[] s1.Split( )`	a `string[]` containing words found within s1

(Once again, don't try to memorize this.)

Here are some examples of working with the built-in string functions.
Let's now work with `String s = " Lecture23 "`.

Method Call	Returns
s.ToLower( )	" lecture23 "
s.ToUpper( )	" LECTURE23 "
s.Trim( )	"Lecture23"
s.TrimStart( )	"Lecture23 "
s.TrimEnd( )	" Lecture23"

Remember that strings are immutable, so we have to assign the result of any of these methods to a variable.
Example `string s2 = s.ToLower( );`

Built-In Static Methods

The previous slides showed static methods for built-in types.

Their purpose is to illustrate that the language provides methods that are useful so that you do not have to write your own implementation.

If you want to perform a common operation, like convert a string to upper case, your first step: check for a method that does it for you.

Take a look through the online documentation (e.g., Microsoft Developer Network [MSDN]), or use Google.

We can send just about anything to the console, but the output may not be particularly human-readable, or may not make sense.

The console output operation thus permits us to format the data to a style of our choosing.

Motivation: suppose we store a price like $104.50 in a `double`.

Printing this to the screen might look like `104.5`
What we'd like is `$104.50`.

The specification of a C# output format resembles the following:

```
{N,M:s}
```

Where:
  `N` is the position of the item in the list of values;
  `M` is the width of the region to contain the formatted output; and
  `s` is the formatting code.

Fortunately, the designers of C# have created a bunch of useful format specifications for us.

In the following table, the symbol 'ͺ' indicates a blank space.

Specifier	Letter	Supported Numbers	Example	Number	Sample Output
General	G	All	{0,8:G}	1234.5F	ͺͺ1234.5
			{0,8:G}	1234	ͺͺͺͺ1234
Fixed Point	F	All	{0,8:F2}	1234.5F	ͺ1234.50
Round Trip	R	All	{0,8:R2}	1234.5F	1234.5
Number	N	All	{0,8:N1}	1234.5F	1,234.5
Exponential	E	All	{0,14:E6}	1234.5F	ͺ1.234500E+003
Decimal	D	Integers Only	{0,8:D}	1234	00001234
Currency	C	All	{0,10:C}	1234.5	ͺ$1,234.50
Percentage	P	All	{0,8:P}	0.89	ͺ89.00ͺ%
Hexadecimal	X	Integers Only	{0,8X}	1234	000004D2

By default, right alignment is used. A negative sign can be used to force left alignment.

When using a format specifier in a `Console.WriteLine` statement, the `Console.WriteLine` statement must be changed.

If we have an integer `x` and we are going to print it to the screen in a formatted way, the command is:

```
Console.WriteLine( "{0}", x );
```

Two things to note in this command: the `{0}` and the appearance of `x`.

`{0}` is the simplest format. It uses the default.
We could also choose `{0,14:E6}` if we wanted exponential format.

A variable to be output is placed after the string, after a comma.

Additional variables can appear after the first, separated by commas.
```
Console.WriteLine( "Integers = {0}, {1}", x, y );
```

In the string, we put a number inside { } braces. This is like an escape sequence, but it tells the compiler: use the variables after the string.

The number placed in the braces matches the position of the variable after the string, starting at 0 (zero) rather than 1.

```
using System;
class OutputFormatting
{
    static void Main( )
    {
        int x = 36;
        Console.WriteLine( "Integer = {0}", x );
        Console.WriteLine( "Fixed Point = {0:F2}", x );
        Console.WriteLine( "Hexadecimal = {0:X}", x );
        Console.WriteLine( "Exponential = {0:E4}", x );
        Console.WriteLine( "Currency = {0,8:C}", x );
        Console.WriteLine( "Currency = {0,-8:C}", x );
    }
}
```

[Demo: This program's output.]

The examples shown in the previous slides (e.g., `char` and `string`) show library (system-provided) functions.

Although we may ask you to work with these methods or to implement them, it is rarely a good practice to "reinvent the wheel".

If you attempt to reinvent something that is already created, there is a good chance you will make some mistakes, even if they are obscure.

For an interesting real-world example, see:
`http://blog.codinghorror.com/whats-wrong-with-turkey/`

When a library function exists to do the operation you want to perform, make use of that library function.

Be sure, however, to check the documentation to learn how to call the function properly (e.g., order of parameters).