

CHAPTER 5: VALUE METHODS

Fall 2019 – CSC 180 – Introduction to Programming



5.1 RETURN VALUES

- Some of the methods we have used, like the Math functions, produce results.

- That is, the effect of invoking the method is to generate a new value, which we usually assign to a variable or use as part of an expression.

- Another example:

```
string str = Console.ReadLine();
```

```
double e = Math.Exp(1.0);  
double height = radius * Math.Sin(angle);
```

- But so far all our methods have been **void**; that is, methods that return no value.

- When you invoke a void method, it is typically on a line by itself, with no assignment:

- Another example of a void method:

```
Console.WriteLine("Hello World!");
```

```
Countdown(3);  
NLines(3);
```

- In this chapter we write **methods that return things**, which we can call **value methods**.

- Instead of **public static void**, which indicates a void method, we see **public static double**, which means that the return value from this method is a double.

- the **return** statement that includes a return **value**.

- The expression you provide can be arbitrarily complicated
- Sometimes it is useful to have **multiple return statements**
- dead code**: read the book ...

```
public static double Area(double radius) {  
    double area = Math.PI * radius * radius;  
    return area;  
}
```

```
public static double Area(double radius) {  
    return Math.PI * radius * radius;  
}
```

5.1 RETURN VALUES (2)

- If you put return statements inside a conditional, then you have to guarantee that every possible path through the program hits a return statement
 - (note: every path must **return** or **throw an exception**)

```
public static double AbsoluteValue(double x) {  
    if ( x < 0 ) {  
        return -x;  
    } else if ( x > 0 ) {  
        return x;  
    }  
} // WRONG!!
```

- Note: **Use IntelliSense** to find more info about these functions ...

```
Console.WriteLine();
```

void Console.WriteLine() (+ 18 overloads)

Writes the current line terminator to the standard output stream.

Exceptions:

System.IO.IOException

- Type **///** to document your methods with XML comments
 - See here more info: <https://docs.microsoft.com/en-us/dotnet/csharp/codedoc> ←this is optional!



5.2 PROGRAM DEVELOPMENT

- Read it on your own
- Read what scaffolding means ...
- Also read section 5.3 Composition (we've seen this already)



5.4 OVERLOADING

- **Overloading**: you can have multiple **methods with the same name** as long as the variables (number and/or types) are different
 - **Example**: define a method that would return the sum of two given numbers.



BOOLEAN VALUES AND EXPRESSIONS

- A **Boolean value** is either **true** or **false**
 - To create a Boolean variable, we can use code similar to: `bool isSaved = true;`
- A **Boolean expression** is an expression that evaluates to either true or false.
 - For this, one can use **relational operators** (**comparison operators**)
 - `x == y` // evaluates to true if: x **equals** y
 - `x != y` // evaluates to true if: x is **not equal to** y
 - `x > y` // evaluates to true if: x is **greater than** y
 - `x < y` // evaluates to true if: x is **less than** y
 - `x >= y` // evaluates to true if: x is **greater than or equal to** y
 - `x <= y` // evaluates to true if: x is **less than or equal to** y
 - To build more complex Boolean expressions, one can use the following **logical operators**:
 - `&&` // **and** operator (evaluates to true when both operands are true)
 - `||` // **or** operator (evaluates to true when at least one operand is true)
 - `!` // **not** operator (evaluates to true when the operand is false – it basically flips the Boolean value)
- A common error is to use `=` instead of `==`.
 - Remember: `=` is an **assignment** operator, `==` is a **comparison** operator



TRUTH TABLES – IF TIME

- A truth table is a small table that allows us to show all possible inputs and to give the results for the logical operators:

| e1 | e2 | (e1 && e2) |
|-------|-------|------------|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

- We typically use T (for true) and F (for false)

| e | ! e |
|---|-----|
| F | T |
| T | F |

| e1 | e2 | (e1 e2) |
|----|----|------------|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

SHORT-CIRCUIT EVALUATION - SKIP

The expression on the left of the `||` operator is evaluated first: if the result is `true`, C# does not (and need not) evaluate the expression on the right — this is called *short-circuit evaluation*. Similarly, for the *and* operator, if the expression on the left of `&&` yields `false`, C# does not need to, nor attempt to, evaluate the expression on the right.

So there are no unnecessary evaluations.

Short circuit evaluation means that the order in which one writes the tests can make a difference. And programmers take advantage of that fact. For example, dividing by zero will cause a run-time error. Consider these two fragments of code:

```
...      (k != 0) && (x / k > 10)      // this works even when k == 0
...      (x / k > 10) && (k != 0)      // this crashes when k == 0
```

If `k` is zero, the first one works, and returns `false` without even attempting to do the problematic division. But the second one will crash. So we have to be careful to order our expressions correctly.

EXAMPLE ...

- Logical operators can simplify nested conditional statements. For example:

```
if ( x > 0 ) {  
    if ( x < 10 ) {  
        Console.WriteLine("x is a positive single digit.");  
    }  
}
```



5.7 BOOLEAN METHODS

- Methods can return boolean values just like any other type:

```
public static bool IsSingleDigit(int x) {  
    if ( x >= 0 && x < 10 ) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

- This can be simplified to:

```
public static bool IsSingleDigit(int x) {  
    return (x >= 0 && x < 10);  
}
```

- To use the method, one could write:

```
Console.WriteLine(IsSingleDigit(2));
```



EXAMPLE - FACTORIAL

- Let's create a recursive method that would compute $n!$ (n factorial)
 - For example: $7! = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
 - In general: $N! = N \cdot (N - 1) \cdot \dots \cdot 3 \cdot 2 \cdot 1$.
 - We'll use the formula: $N! = N \cdot (N - 1)!$, for $N > 1$, and $N! = 1$, for $N = 1$.



EXAMPLE - FIBONACCI

- Fibonacci's sequence: 1, 1, 2, 3, 5, 8, ... ($x_n = x_{n-1} + x_{n-2}$)
 - Write a method that would compute the nth value in the Fibonacci's sequence.
 - What happens if $n < 0$? (throw an exception)



REF PARAMETERS

- Can a method **return** more than one values?
 - The short answer is no. But one solution is to use **ref** parameters.
 - By default, when you pass a **value-type** argument to invoke a method **only a copy of it's value is sent**.
 - That means that any changes made to the argument, will not be reflected upon the original variable.
 - Using **ref** keyword, you are telling the compiler to pass the parameter by reference, hence any change inside the called method will be reflected back into the caller code.
- **Program 9:** To see this behavior, let's implement the following methods:
 - Increase(int x) vs Increase(ref int x)
 - Swap(int a, int b) vs Swap(ref int a, ref int b)



WHAT IS A METHOD (FUNCTION) ?

- **Method (function):** a named collection of statements that performs a useful function.
 - “Don’t repeat yourself” (DRY) – **code reuse**
 - Improves **maintainability** of programs
 - Improves **readability** of programs
 - Simplifies the process of writing programs
- **Method call:** a statement that causes a function/method to execute
- **Method definition:** statements that make up a method
- **A method name** has the same syntactic restrictions as a **variable name**.
 - A method must start with a letter or an underscore and can only contain letters, underscores, and numeric characters
 - **Use verbs** or verb phrases to name methods. This helps other developers to understand the structure of your code
 - By convention, methods start with an upper case letter and **use Pascal case**," just like **WriteLine**



METHOD DEFINITION

- The **method** named **Main** is special, but its syntax is similar for other **methods**:

```
public static void Name(<list of parameters>)  
{  
    //statements = the body of the method  
}
```

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

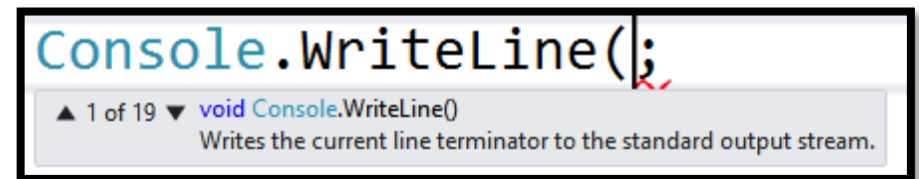
- We'll learn about **public** and **static** later in the semester, so just use them for now.
- **Name** is an **identifier** that you can choose for your **Method** (also called **Function**)
- The **(list of parameters)** specifies what information, if any, you have to provide to **use** (**invoke, call**) the method.
 - empty parentheses **()** mean that the method takes **no parameters (no required parameters)**.
- **void** means the method doesn't return anything – it simply performs a task and then terminates
 - Instead of it, one can put any type (such as **int**, **string**, ...) – then the method must **return** a value of that type



GLOSSARY – KNOW IT!

- Please make sure to read the book!

- **method**: A named sequence of statements that performs a useful function.
 - Methods may or may not take parameters, and may or may not return a value.
- **parameter**: A piece of information a method requires before it can run.
 - Parameters are variables: they contain values and have types.
- **argument**: A value that you provide **when you invoke** a method.
 - This value must have the same type as the corresponding parameter.
- **return type**: The part of a method declaration that indicates what type of value the method returns.
 - **return value**: The value provided as the result of a method invocation.
- **overloading**: Having more than one method with the same name but different parameters.
 - when you invoke an overloaded method, the system knows which version to use by looking at the arguments you provide.

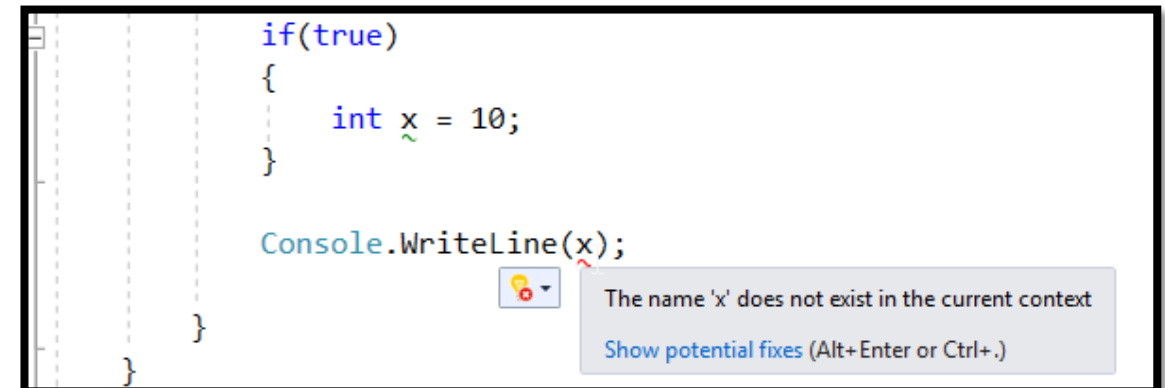


- **Example**: understand how to **write (define)** and **invoke (call)** methods that take parameters:
 - Write the first line of a method named **Zool** that takes three **parameters**: an int and two strings.
 - Write a line of code that invokes **Zool**, passing as **arguments** the value 11, the name of your first pet, and the name of the street you grew up on.



VARIABLE **SCOPE** - DISCUSSION

- **Scope of a variable** is the portion of the code that can access that variable.
- Variables defined inside a function are **local** to that function. They are hidden from the statements in other functions, which normally cannot access them.
 - Other functions may have separate, distinct variables with the same name.
- A function's local variables exist only while the function is executing. This is known as the **lifetime** of a local variable.
 - When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.
 - This means that **any value stored in a local variable is lost between calls** to the function in which the variable is declared.
- If a variable is declared inside a block, it is only accessible inside that block ... ➔ ➔ ➔ ➔ ➔ ➔ ➔



```
if(true)
{
    int x = 10;
}

Console.WriteLine(x);
}
```

The name 'x' does not exist in the current context
[Show potential fixes \(Alt+Enter or Ctrl+.\)](#)


EXCEPTION HANDLING – SIMPLIFIED – IF TIME

- An **exception** is an indication of an error or exceptional condition
- Use the **throw** keyword to throw a new exception
 - **When you throw an exception, execution of the current block of code terminates and the CLR passes control to the first available exception handler that catches the exception.**
- Use **try/catch** blocks to **handle exceptions**
 - When a method throws an exception, the calling code must be prepared to detect and handle this exception.
 - If the calling code does not detect the exception, the code is aborted and the exception is automatically propagated to the code that invoked the calling code.
- Example: ask the user for an **int** and enter a **double** ...
 - You could **catch** the exception ... or use **Int32.TryParse** instead

```
throw new Exception("The 'Name' parameter is null.");
```

```
try
{ //code that may throw an exception
}
catch (Exception ex)
{
    // Catch an exception
}
```

```
string str = Console.ReadLine();
int x;
Int32.TryParse(str, x)
```

 **bool** `int.TryParse(string s, out int result)` (+ 1 overload)

Converts the string representation of a number to its 32-bit signed integer equivalent. A return value indicates whether the conversion succeeded.

HOMework FOR CHAPTER 5

- Requirements: see moodle for details
- Deadline: see moodle
- **Reminder: If your code does not compile, crashes at start, or contains no meaningful comments, it will automatically be graded with 0!**

