|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CS 1400 Lab 17**  **Methods**  **Version 1.0**  **Objectives:**  The objective of this lab is to help you understand how to design, write and use your own methods.  After completing this lab, you should be able to design solutions to problems that use your programmer written methods. These methods can be ***instance*** properties & methods or ***static*** property & methods. You should be able to design either method using the ***Pseudo-code*** programming process.  **Study Material**   1. **Overview of Methods**  |  | | --- | | **Introduction to Methods**  A method is a sequence of programming statements that does a particular task (provides a service), returns a value (even void), that has a name we can use when we want to invoke (call) that method and finally has zero or more parameters in its parameter list. You have already encountered a number of built-in methods such as ***Math.Sqrt(…)*** that perform common mathematical tasks. In this assignment you will explore, in detail, how to write and invoke your own methods.  When writing a program we often divide the work that we have to do into smaller tasks. Then each task is written as a method. This is particularly useful when a given task is to be invoked over and over again in a program. Let's take as an example the task of converting a temperature in degrees Fahrenheit to degrees Celsius.  **Method Names**  When you write a method, it is important that you give the method a meaningful name. The name should reflect what it is that the method does. In our example, we are converting Fahrenheit temperatures to Celsius. A useful name then might be ConvertToCelsius or Convert\_To\_Celsius.  **Method Prolog**  Before writing the method, you need to think carefully about what the purpose of the method is, any data and restrictions on the data, that you have to provide to the method so that it can do its work. Also what, if anything, that the method returns, when it has provided its service.  This information is written in the ***Method Prolog***. Let's write the ***Method Prolog***, for our conversion program. We will use comments at the beginning of the method to do this. Remember, every method that you write must have a complete ***Method Prolog*** as illustrated here.  /// <summary>  /// Purpose: Convert the parameter fahrenheitDegrees to celciusDegrees and /// return it.  /// </summary>  /// <param name="farhenheitDegrees">double as Degrees Farhenheit</param>  /// <returns>double as Degrees Celcius</returns>  **Method Outline**  Having written the Method Prolog, we are now ready to write down an outline of how our method will work. This is done by writing down the method declaration (sometimes call a stub method), which includes the method ***header***, an ***opening curly brace*** and a ***closing curly brace***. The method declaration tells the compiler what the name of the method is, what parameters it takes and the type of the data the method will return. Then inside of the curly braces, write down a ***Pseudo-Code*** description of what the method will do (the service it provides). For our ***ConvertToCelsius*** method, the method header would look like this:  method declaration  And our completed method declaration should look like:  /// <summary>  /// Purpose: Convert the parameter fahrenheitDegrees to celciusDegrees and  /// return it  /// </summary>  /// <param name="farhenheitDegrees">double as Degrees Farhenheit</param>  /// <returns>double as Degrees Celcius</returns>  **public** **double** **ConvertToCelcius**(**double** **farhenheitDegrees**)  {  **// declare const Freeze, MULT**  **// declare variable double celciusDegrees.**  **// Subtract 32.0 from fahrenheitDegrees - pseudo-code**  **// Multiply the result by 5.0 Divided the by 9.0**  **// Return the result as celciusDegrees**  //**return** celciusDegrees;  **return** 0.0;  }  **Adding the Code**  Now fill in the code that implements the ***Pseudo-Code*** that you have written. Our example will look like this. Notice how nicely documented our method is.  /// <summary>  /// Purpose: Convert the parameter fahrenheitDegrees to celciusDegrees and  /// return it  /// </summary>  /// <param name="farhenheitDegrees">double as Degrees Farhenheit</param>  /// <returns>double as Degrees Celcius</returns>  **static** **public** **double** **ConvertToCelcius**(**double** **farhenheitDegrees**)  {  **const** **double** **FREEZE** **=** 32.0;  **const** **double** **MULT** **=** 5.0 **/** 9.0;  **double** **celciusDegrees** **=** 0.0;  **// Subtract 32.0 from fahrenheitDegrees - pseudo-code**  **// Multiply thus result by 5.0 divide the result by 9.0**  **celciusDegrees** **=** (**farhenheitDegrees** **-** **FREEZE**) **\*** **MULT**;  **// Return the celciusDegrees**  **return** **celciusDegrees**;  }  **Invoking a Method**  To invoke, or call a method, you simply write down the method name and in the parentheses following the method name write down the name of the variable you want to pass-by-value to the method so it can do its work. For example, to invoke the ConvertToCelsius method, I might write  **double** **tempF** **=** 27.56;  **double** **resultC** **=** **ConvertToCelcius**(**tempF**);  **The Complete Program**  A complete program containing the ConvertToCelsius method and the code that calls it is shown below:  **using** **System**;  **static** **class** Program  {  /// <summary>  /// Purpose: Entry point to your C# program  /// </summary>  **static** **public** **void** **Main**()  {  **double** **tempF** **=** 27.56;  **double** **tempC** **=** **ConvertToCelsius**(**tempF**);  Console**.WriteLine**("Temperature in Fahrenheit {0:F3} converted to  Temperature in Celsius {1:F3}", **tempF**, **tempC**);  Console**.WriteLine**("Press Enter to continue ...");  Console**.ReadLine**();  }**//End Main()**  /// <summary>  /// Purpose: Convert the parameter fahrenheitDegrees to celsiusDegrees  /// and return it  /// </summary>  /// <param name="farhenheitDegrees">double as Degrees  /// Farhenheit</param>  /// <returns>double as Degrees Celcius</returns>  **static** **public** **double** **ConvertToCelsius**(**double** **farhenheitDegrees**)  {  **const** **double** **FREEZE** **=** 32.0;  **const** **double** **MULT** **=** 5.0 **/** 9.0;  **double** **celciusDegrees** **=** 0.0;  **// Subtract 32.0 from fahrenheitDegrees - pseudo-code**  **// Multiply thus result by 5.0 divide the result by 9.0**  **celsiusDegrees** **=** (**farhenheitDegrees** **-** **FREEZE**) **\*** **MULT**;  **// Return the celciusDegrees**  **return** **celsiusDegrees**;  }  }**//End class Program** End class Program |  |  | | --- | | **Stepwise Refinement**  This example is taken from Horstmann and Budd's textbook, "Big C++". One of the most powerful strategies for problem solving is the process of **Stepwise Refinement**. Stepwise refinement simply says that to solve a difficult problem, break it down into a set of simpler tasks. Then keep breaking down each of these simpler tasks into even simpler ones, until you end up with tasks that you know how to solve.  These smaller, simpler tasks are usually excellent candidates for methods. Remember, that a method does one thing, does it well and then moves on! Such methods are well focused, small, and does one thing well.  Consider this problem from real life. You get up in the morning and simply must get some orange juice. How do you get orange juice? You can see if someone else can bring you some. If this fails, then you have to make orange juice. How do you make orange juice? If there is some frozen orange juice in the freezer, then you can make frozen orange juice. How do you make frozen orange juice? Simply thaw frozen orange juice and add water. How do you thaw frozen orange juice? Just open the can and put can in the microwave for a few minutes. What if you don't have any frozen orange juice? The you will have to make fresh orange juice. How do you make fresh orange juice.? Simply slice an orange in half, then squeeze out the juice.  The solution to the orange juice problem breaks down tasks in two different ways, with **decisions** and **refinements**. You are familiar with decisions. **If** you have frozen orange juice, use it, **else** use fresh oranges. Decisions are implemented with **if** and **if/else** statements. A refinement names a composite task and later breaks that task down into smaller pieces: thaw frozen orange juice: open the can ... put can in microwave.  This process is illustrated in the Activity Diagram below:  making oj |  1. [***How Methods Work (Powerpoint Presentation in lab folder on Canvas)***](http://debryro.tc.uvu.edu/1400/labs/lab17/function.ppt)   **Programming Exercise**  **Designing Your Own Method**   |  | | --- | | Designing the IsLeap MethodIntroduction In this exercise, you will design a method using the pseudo-code programming process. Do not write any code for this lab. The Problem You will create a class and a method that determines whether or not a given year is a leap year. A leap year has the property that   * It is evenly divisible by 4, but * It is not divisible by 100 except when it is divisible by 400   For example, the year 2000 was a leap year, but the year 1000 was not. Deriving the Solution This program is a fairly straight forward application of the things that you have learned in this and previous labs. To begin, draw a UML Class Diagram for the class ***Leap\_Year***, also draw and Activity Diagram and the Pseudo-Code that describes the algorithm for determining whether or not a year is a leap year. You should be able to see the logic that is used to determine whether or not it is a leap year. Do not diagram the entire program, just the algorithm for determining if the given year is a leap year and write the method stub ***IsLeapYear( ){…}***. Then using the Activity Diagram, write the pseudo-code for this method. Your method will take an integer value (the year) as its argument and will return a Boolean. If the year is a leap year, the function will return ***true***,  otherwise it will return ***false***;. Hint, see Lab 18.  Add a Method Prologs, as described in the reading material for this lab. Do not write a complete program, just turn in the Pseudo-Code and Method Prolog for your method in Lab\_17. File(s) to Submit: Place your complete Project folder in a zip file and name the zip file Lab\_17\_your-initials\_V1.0.zip. For example, I would name my file Lab\_17\_DAF\_V1.0.zip. Submit this assignment as Lab #17 on Canvas. |  |  |  |  | | --- | --- | --- | |  | ***Grading Checklist*** |  | | # | Program | C(Correct)X(incorrect) | | 1 | Meets & works to specifications | 6 points | | 2 | Error Free, elegant & efficient | 4 points | | 3 | Pseudo-Code | -3 points | | 4 | Style Guidelines | -2 points | | 6 | Source Files(s) & Formatting | -2 points | | 7 | Project Prolog | -1 points | | 8 | Function Prologs | -1 points | | 9 | Zip Filename | -1 points | | 10 | Lab & Project Names | -1 points | | 11 | Zip File is invalid or will not unzip | Lab = 0 pts | |  | Total Points | 10 | 0-9 | |