|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CS 1400 Lab #15**  **Dice Game Design**  **Version 1.0**  **Objectives:**  The objective of this lab is to help you understand how to design a program that simulates a rolling dice game. You need to carefully design the Activity Diagram and Pseudo-Code that validates input, and uses loops together with branches and random numbers. You will also use functional decomposition to decompose the lab into a set of appropriate methods.  **Study Material**   |  | | --- | | **Do it Again?**  Many programs will loop through some code statements and when they are done, ask the user if they want to do the same operations again. For example, in our dice game program you might ask the user if they want to play again at the end of the game. The basic logic for the dice game is shown in the following Activity Diagram:    **Solution**  The perfect solution to this kind of problem is to use a do-while loop. The code that mirrors the logic in the Activity Diagram shown above, looks like this. |   **static** **private** **void** **PlayDiceGame**()  {  **string** **inputStg** **=** **""**;  **char** **inputChar** **=** **'\0'**;  **do**  {  Console**.Clear**();  Console**.WriteLine**(**"--------------- Dice-Game --------------------\n\n"**);  Console**.Write**(**"\tDo you want to play? Y)es Q)uit: "**);  **if**(**char.TryParse**((**inputStg=**Console**.ReadLine**()), **out** **inputChar**) **==** **false**)  {  **if** (**inputStg** **==** **""**)  **inputStg** **=** **"<empty>"**;  Console**.WriteLine**(**"Invalid menu selection {0}"**, **inputStg**);  Console**.ReadLine**();  **continue**;  }  **else**  {  // code statements to play the game until ‘q’ is entered  }  }**while** (**true**);  }//End PlayDiceGame( )  **Using loops to validate input**   |  |  | | --- | --- | | Testing Values to be Within a RangeProblem OutlinePrograms, that get input from a user, require that the data satisfy some constraints. Consider a simple game where the user has to guess a value between 1 and 10. Testing the user's input data to see if it lies within this range is a common way to use both branches and loops.Solution When gathering input from the user that must meet some constraints, a program needs to follow the steps illustrated in the Activity Diagram shown below:  validating input with constraints  In this example, the program first prompts the user to input some data. If the input is a valid data-type and the input must satisfies some constraint, then you should tell this to the user. Then get the input from the user. Now test it to see if it satisfies the constraint. In our example problem, the input data must be between 1 and 10. The code snippet below illustrates this process.  **string inputStg = “”;**  **int input = 0;**  **if(int.TryParse((inputStg=Console.ReadLine()), out input))**  **{**  **if (input >=1 && input <= 10)**  **//code block**  **else**  **//code block**  **}**  **else**  **//code block**  would handle this test nicely. Note that expression below, places the string from the Keyboard buffer into the string variable ***inputStg.***  **(inputStg=Console.ReadLine())**  Then the method call looks like the code below to the program.  **int.TryParse((inputStg), out input)**  The ***TryParse*** method parses the ***inputStg***, if it can be converted to an integer, places the integer value in the input and returns true. If ***inputStg*** cannot be parsed to an integer, it places a zero in input and returns false. Since we want to print an error message if the constraint is **not** met, we could also have used the inverse condition to test the input  **if (input < 1 || input > 10)**  If the test fails, the program should print a message to the user, and then go back and ask for the input again. The program should stay in this loop until the constraint is satisfied. Once the user input is validated, then the program can go on to process the data. Coding the Solution It should be obvious from the ***Activity Diagram***, that we need some kind of looping mechanism to code up the solution to this problem. What kind of loop would serve best? Notice that we always want to go through the body of the loop at least once. That is, we always want to get some input and test it. Recall that a **do-while** loop meets this criteria. It will perform the body of the loop first, and then test some condition to see if the loop should be executed again. Inside the loop, we want to   * Prompt the user for input. * Get the user's input, as a string, and parse it to the appropriate data type. If it cannot be parsed to a the appropriate data-type prompts the use that the input is invalid and to re-enter the value. * If the input is valid and meets the given constraints for the input continues. * If it does not meet the constraints, print a warning message and prompt for valid input. * Clear the screen and ask for the input again.   The looping condition will test the input value against the constraint. The code to do this could thus be written as shown below:  /// <summary>  /// Purpose: To demonstrate this use of input management  /// </summary>  **static** **public** **void** **TestInput**()  {  **const** **int** **MAX\_VAL** **=** 10;  **const** **int** **MIN\_VAL** **=** 1;  **int** **inputValue** **=** **-**1;  **string** **inputStg** **=** **""**;  **do**  {  **inputValue** **=** **-**1;  Console**.Clear**();  Console**.WriteLine**(**"---------------- Testing for Numbers in a Range -------------------------\n\n"**);  Console**.Write**(**"\tPlease enter a number in the range of 1 to {0}: "**, **MAX\_VAL**);  **if** (**int.TryParse**((**inputStg** **=** Console**.ReadLine**()), **out** **inputValue**) **==** **true**)  {  **if** (**inputValue** **<** **MIN\_VAL** **||** **inputValue** **>** **MAX\_VAL**)  {  Console**.WriteLine**(**"\tYour number {0:D} isn't in the range of 1 to {1:D}"**, **inputValue**, **MAX\_VAL**);  Console**.WriteLine**(**"\tPlease re-enter your number!"**);  Console**.WriteLine**(**"\tPress Enter to continue ... "**);  Console**.ReadLine**();  **continue**;  }  **else**  {  Console**.WriteLine**(**"\tGood number {0}"**, **inputValue**);  Console**.WriteLine**(**"\tGoodbye!"**);  Console**.WriteLine**(**"\tPress enter to continue ... "**);  Console**.ReadLine**();  **break**;  }  }  **else**  {  Console**.WriteLine**(**"\tYour input \"{0}\" was not valid!, Please re-enter"**,**inputStg**);  Console**.WriteLine**(**"\tPress the Enter key to continue ... "**);  Console**.ReadLine**();  **continue**;  }  } **while** (**true**);  }**//End TestInput()** | | |  | | | Random Numbers There are times when it is necessary to simulate some kind of random event in a program. For example, if you wanted a program to model a board game that uses dice, to determine events in the game, you would have to simulate the roll of the dice. The numbers that come up on a roll of the dice is random number, because you can't predict what number will come up next. The .Net library has an instance class, ***Random*** built into it, this generates pseudo-random numbers. They are called pseudo-random numbers because they are not exactly random, but very close to random. To generate random numbers, we need an object of the Random class. You can create an object of the Random class by writing  **Random randNums = new Random( );**  make sure the instantiation of the Random object is NOT in a loop and is only instantiated once in the program!  The Random class has a method called ***Next( )*** that returns a pseudo random number between 0 and 2,147,483,646. You can scale the values returned by Next( ) by including an integer argument when calling the method. For example,  **int dieValue = randomNums.Next( 6 );**  will return one of six values, starting at zero and going up to five. Since we need values for our dice between one and six, we will add a shift value when we call the Next method.  **int dieValue = randomNums.Next( 1, 7 );**  This form of the Next method returns pseudo-random numbers starting from one and going up to six, but not including seven. Example Let's code our dice example to illustrate how to generate random numbers within a range. The following code will generate 10 random dice throws:  **using** **System**;  **static** **class** Program  {  **static** **void** **Main**()  {  **const** **int** **MAX\_CNT** **=** 10;  **const** **int** **BOX\_CAR** **=** 6;  Random **randomNums** **=** **new** Random( );  **int** **d1** **=** 0;  **for** (**int** **i** **=** 0; **i** **<** **MAX\_CNT**; **i++**)  {  **d1** **=** **randomNums.Next**(1, **BOX\_CAR+**1);  Console**.WriteLine**(**"You rolled {0}"**, **d1**);  }  Console**.ReadLine**( );  } **//End Main()**  }**//End class Program** | | | **For Loops**  Carefully study the following figure that illustrates the anatomy of a for loop.  for loop  When the for loop is executed, things happen in the following order: First of all part (a), the initialization part gets done on entering the for loop. In this example, the variable *i*  gets set to zero.   1. Secondly, part (b), the conditional part of the loop is tested. In this case, we test to see if *i* is less than a constant, ***MAX***. If the condition is true, then the body of the loop is executed. If the condition is false, we exit the loop and go on to execute the rest of the program. Finally, part (c), the increment is executed. In this example, ***i*** gets incremented. We now go back to step 2.   The following program will print a square pattern of asterisks using a pair of nested loops. Start a new project and copy this code into it. Build the project and run the program to see what it does. Study the code carefully until you understand how it works.  using System;   class Program {    static void Main()    {       const char STAR = '\*';       const int MAX = 6;         // Each time through this loop prints one row       for (int i = 0; i < MAX; i++)       {          // Each time through this loop prints one star          for (int j = 0; j < MAX; j++)          {             // print a star, but stay on the same line             Console.Write(STAR);          }          // end the line          Console.WriteLine();       }       Console.ReadLine();    }//End Main() }//End class Program |   **The Problem**  For this assignment, you will be designing a program that simulates rolling a pair of dice. Your program will perform the following steps:   1. Create a Random Number generator object.  * Ask the user if they want to roll the dice * Get the user's response and validate that it is either 'y' or 'n'. If it is not a 'y' or an 'n', tell the user that the input is invalid and ask for another response. * If the user responds with a 'y' then   + Generate two random numbers in the range 1 - 6   + If the two numbers are 6 and 6, display the message "You rolled Boxcars"   + If the two numbers are 1 and 1 display the message "You rolled Snake-eyes"   + In all other cases and show the values of the two random numbers.   + Pause the Console so the user can read the output.   + Allow the user to press the “***Enter Key***” or enter the ‘***Q***’ key.   + If the user presses the “***Enter Key***” continue to play; however, if the user enters the ‘***Q***’ key then return to step 2 and ask the user again if they want to play the dice game.   + If the user responds with an '***N***', print a goodbye message and quit.   **Deriving the Solution**  This program is an application of the things that you have studied in this lab. To begin, Fill out the ***Algorithm Design Worksheet***. Draw an ***Activity Diagram*** that describes the steps outlined above and finally write the ***Pseudo-Code*** for this program as you have learned to do in previous labs.  **Writing the Psuedo-Code**  Start Visual Studio and create a new Console project. Call it lab #16. Copy the line-by-line description of your program from the ***Algorithm Design Worksheet*** to your new project. Use your ***Algorithm Design Worksheet*** and ***Activity Diagram*** and develop each line of Pseudo-Code as a comment in your project code page. Your ***Pseudo-Code*** must be complete and describe the methods, loops, decisions and other code required to make your program work. As you are fill out the ***Algorithm Design Worksheet***, develop your ***Activity Diagram*** as well as your ***Pseudo-Code*** the solution to this lab remember that you need to break the solution into methods. Your program methods should be structured as shown below.  //method prolog  void static Main( )  {  new Dice\_Game( ). PlayDiceGame( );  //the purpose of Main is to create the object and start your program  //the name (identifier) of you methods is up to you  }  //method prolog  public void PlayDiceGame( )  {  //statements to play the dice game  }  //other methods required for the lab  A working version of this program can be found on Canvas. ***Pay special attention to the display of the high score and current score.***  **File(s) to Submit:**  Place your complete project folder in zip file and name the zip file Lab\_15\_your-initials\_V1.0.zip. For example, I would name my file  Lab\_15\_DAF\_V1.0.zip. Submit this assignment as Lab #15 on Canvas. Please do not include any other files in your submission.   |  |  |  | | --- | --- | --- | |  | **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 | |