**Name: Jason Truong**

For each of the below questions, write a short sentence or two to express (in your own words) your answer. Keep the answers short, but use complete, correct, English sentences.

If it helps to clarify the questions, feel free to mentally prefix all the questions with the phrase "According to the video…"

* After you’ve watched all the videos, please answer this question:  
  Of all the videos that you watched, if you could pick one video to be re-recorded by the instructor outside of class which would you choose? Why?  
  (Keep in mind the recording outside of class will omit any pauses from the instructor answering student questions, have less hemming and hawing, etc, and generally be more concise)

|  |
| --- |
| < Write your answer here > |

**VIDEO: While Loops**

* In a nutshell, how does the **while** loop execute?

|  |
| --- |
| while (x > 0) <--- Check  { <--- Statement  Console.WriteLine("x is: {0}", x);  x = x - 1 <--  } <--- Finish  <--- "Break"  The while loop follows these sequence of events:  -Check for its conditional-statement to be true  --If 'True' continue  ---Do statement that follows While  ---Finish statement  ---Return to 'check' if while conditional still remains as 'True'  --If 'False', break |

* What is typically true of a situation where you would want to use a while loop?  
  (At the outset/start of the execution of the loop, what does the while loop typically NOT know?)

|  |
| --- |
| Doing a set of actions for an indeterminate amount of times.  The while loop does not know how many times you're going through the loop. |

* Around the 3:10 mark the instructor explains a good example of where you might use a while loop (one that’s much better than the loop that counts down which is used at the start). What is this example ,and why is it a good situation to use a while loop?

|  |
| --- |
| A user-determinate loop, where in each iteration of the loop it; it would ask the user if they wish to continue through the series or 'break' the loop. Basically manually giving a sense of a UX. |

* Syntactically, do we need the opening and closing curly braces that follow the line with the word while on it? If you do NOT have curly braces then which lines are considered to be part of the while loop (i.e,. which lines will be repeated)?

|  |
| --- |
| No, you do not need the brackets; creating a block statement.  If the curly braces are absent, it will only loop the immediate line that follows the while-statement. |

**VIDEO: For Loops**

* A **for** loop is made for doing what?   
  Also – fill in the blank: “Knowing \_\_\_\_\_\_\_\_\_\_ is usually a good indicator that you should use a for loop”

|  |
| --- |
| The for loop is made for 'counting'. Knowing how many times the code should loop is usually a good indicator that you should use a for-loop. |

* Copy the example of the for loop that was show in the video here:

|  |
| --- |
| for (int i=0; i < 10; i++)  {  Console.WriteLin("i is: {0}": i);  } |

* What does **i++** do?

|  |
| --- |
| it adds +1 to the integer in question; which is i in this case. |

* Inside the for loop, what symbol do you use to separate the initialization from the check from the increment?

|  |
| --- |
| A semi-colon: ';'. |

* When you arrive at the for loop (on line 25), how many times do you do the **int i = 0** part? Is it possible that the program might ever skip this part?

|  |
| --- |
| Only once.  If the integer "i" was already pre-established, then you can skip the actualization inside the for-loop. |

* When you arrive at the for loop (on line 25), what is done after the the **int i = 0** part? What is the minimum number of times that this part will be done?

|  |
| --- |
| The condition of the loop that will be checked after each act.  The minimum number of times in the specific example will be 10, as the loop is specified to stop at the condition that "i" is no longer less than 10 - 10 equating to 10 does not make it less than 10.  Ten. |

* Next, the **for** loop will do the body of the loop? What is the body of the loop given in the example in the video?

|  |
| --- |
| Console.WriteLine(" i is: {0}", i++); |

* Next, the **for** loop will do what part of the loop?

|  |
| --- |
| The increment math part of the loop. |

* If any point the check part is false, what does the program skip? What does it do next?

|  |
| --- |
| It skips the body and breaks the loop. Continuing to the next statements after the loop; as per usual. |

**VIDEO: Do-While Loops**

* What is the major difference between a do-while loop and the other two types of loops that we looked at?

|  |
| --- |
| The Do-Whille loop acts on the loop before it makes the check. Guaranteeing that the loop always runs once. |

* Do…while loops MUST have a semi-colon where?

|  |
| --- |
| At the end of the while conditional statement. |

* In C# (and many other languages), what is **whitespace**? What implication does this definition have in terms of where we can put extra blank lines in our programs?

|  |
| --- |
| Whitespace is the space between code that the program does not care to look at, so as long as the syntax is correct, you can have as much whitespace as possible.  Obnoxious for the readers of the code, but the program does not mind. |

**VIDEO: Increment (++) and decrement (--) operators**

* Exactly how does **x--** work?

|  |
| --- |
| It decrements the value of integer x after the integer's usage.  So it would be for example; x = 31  31  then  30 |

* Give the example of C# code that illustrates this that was used in the video, and make sure to explain why it prints 10 and NOT 9.

|  |
| --- |
| Console.WriteLine("x is: {0}", x--);  X prints the 10 first because the code engages with the value stored in the integer before the decrement modifying the integer stored. |

* Exactly how does **--x** work?

|  |
| --- |
| It decrements the value of the integer x before it's usage.  So it would be for example;x = 31  30  used  and held at 30. |

* Give the example of C# code that illustrates this that was used in the video, and make sure to explain why it prints 9 twice and NOT 10 then 9 (and then prints 8 twice instead of 9 and then 8, etc, etc).

|  |
| --- |
| Console.WriteLine("x is: {0}", --x);  Console.WriteLine("x is {0}", x);  It decreases x by one first, and prints the value of x second.  The value of x still sustains into the second print statement. |

* In the following code snippet, why does it not matter whether use you i++ (as written) or ++i?  
  for( int i = 0; i < 10; **i++**)

|  |
| --- |
| Because we're not engaging with the increment of i beyond the loop. Its in its own entity, engaged in its own little world. |

**VIDEO: Compound operators (+=, \*=, etc)**

* In addition to C#, what other languages have the ++ and -- operators? Very briefly explain why.

|  |
| --- |
| The function was originally built on the language of C. So anything built off of C implements this feature. |

* Does **++x++;** work?   
  (You only need a yes/no answer for this question, and then you need to remember this answer when writing up your own programs. )

|  |
| --- |
| No. |

* Instead of writing out **x = x + 2;**, how can I write an equivalent statement using a compound operator?

|  |
| --- |
| x += 2 |

* List out the other compound operators used in the video, and which basic operation each one does:

|  |
| --- |
| += // Addition  -=// Subtraction  \*=// Multiplication  /=// Division |

**VIDEO: Random Numbers: Seed Values**

* Can C# actually generate a truly random number? What is the brief, intuitive explanation that’s offered in the video?

|  |
| --- |
| You cannot generate truly random numbers, and you can only feign randomness through intense algorithmic piecewise-design. |

* Copy down the line that creates a Random object from the example in the video.  
  If the parentheses are left empty, what is used as the “seed value”?

|  |
| --- |
| It would pick the date in milliseconds, and takes this arbitrary number and store it as a seed value. |

* How does C# generate a random number each time that my program calls the **nums.Next()** method?

|  |
| --- |
| C# will take the last instance, grind it through a set of calculations, and plug it into the called Next() function. |

* If one creates a second Random object that is given the same seed value (as a previously created Random object) what will be true about the sequence of “random” numbers that it generates?

|  |
| --- |
| You plug in the same value, you'll get out the same sequence of numbers every single time. |

* Briefly give an example of where this might be useful:

|  |
| --- |
| It's super useful for testing programs which will try to do the same sequence of events to reproduce effects, or to try new events; then recreate that same event in succession. |

* Given that the Random object is an adequate (but not great) random number generator, give an example or two of situations where you should avoid using the normal, **Random** class to generate random numbers:

|  |
| --- |
| If the number is generated and there's high stakes involved, utilizing an abusable 'random' generator is not ideal. So gambling or cryptography. |

* Copy down the line that creates a Random object with a seed value of 100 from the example in the video

|  |
| --- |
| Random numTwo = new Random(100); |

**VIDEO: Random Numbers: the .Net-specific .Next(min, maxPlusOne) method**

* Pseudo-mathematically speaking, what do we mean when we say the numbers we’re generating are random?

|  |
| --- |
| If we were to place all the numbers appearing along the line of a bell-curve, there should be no peak in the curve whatsoever. And a truly randomly generated set of numbers will be equal in appearances. |

* Given a Random object named nums, what possible numbers will **nums.Next()** generate?   
  Make sure to be clear about the smallest possible number, the largest possible number, and the type (double, float, int, etc)

|  |
| --- |
| The numbers capable of being generated would be between an estimated range of 0 to 2.1billion.  Regarding the status of integers, floats, doubles, and decimals - all would retain that previously mentioned trait just proactively limited by their innate range. |

* Given a Random object named nums, what possible numbers will **nums.Next(1, 7)** generate?   
  Make sure to be clear about the smallest possible number, the largest possible number, and the type (double, float, int, etc)

|  |
| --- |
| It would generate a number from the number of the beginning integer til' the highest digit, but not on.  So the lowest possible number is 1  The largest is 6  But this is in regards to integers.  For finer decimal numbers, such as floats; doubles; and decimals, they would generate a ".0" after the number. |

* Given a Random object named nums, what is the largest possible number that **nums.Next(1, 7)** might generate? (Yes, this duplicates what’s being asked for in the prior question, but getting this wrong is a very common mistake so it’s good to write your answer out twice)

|  |
| --- |
| The largest possible value is 6, while the smallest plausible number is 1. |

* Given a Random object named nums, what possible numbers will **nums.NextDouble()** generate?   
  Make sure to be clear about the smallest possible number, the largest possible number, and the type (double, float, int, etc)

|  |
| --- |
| The smallest possible number generated would be 0.  The largest value generated for double would be 6 followed by a determinate amount of 9's after the decimal - oriented around the data-types limit. |

**VIDEO: Random Numbers: Seed Values & non-random numbers**

* Describe what will the following block of code produce as output (obviously you can’t know the exact numbers it will produce, so instead focus on what distinctive behavior it will demonstrate)

for (i = 0; i < 5; i++)

{

Random nums = new Random(); // seed value based on time

Console.WriteLine("Next number: {0}", nums.Next());

}

|  |
| --- |
| It will generate the same number in the loop, the random function doesn't have a chance to create a new clean object. |

* **Why** does the code produce the output that it does in the prior question?

|  |
| --- |
| Its looping so fast within the same millisecond, that it doesn't have enough time to create a new random. |

* How should you fix this problem?

|  |
| --- |
| Delay the loop.  Thread.sleep(1); will slow it down.  A better way is to just move the seed value initializing outside the loop. |

* Ideally, how many random number generators should your program create?

|  |
| --- |
| In the absolute ideal, you would only need one. |

**VIDEO: Random Numbers: Using the plain .Next() & modulus ( % )**

* If I wanted to randomly generate whole numbers in the range of [1,5], what C# expression would I use?

|  |
| --- |
| (rng.Next() % 5) + 1 |

* Using the above expression, if **rng** randomly generates 1 what value will the overall expression produce? Make sure to explain each and every ‘step’ that the computer goes through in order to arrive at your answer.

|  |
| --- |
| 1 would enter modulus 5, which is a result of 1, because of the remainder functionality.  And it would add 1 after the modulus finishes, thus becoming 2.  (rng.Next(1) % 5) + 1 = 2. |

* What part of the expression (that you listed for the question that’s two before this question) determines how many values you get? Which part determines the lowest possible value that you’ll see?

|  |
| --- |
| rng.Next() determines my values initially.  % determines the lowest possible value I'll get. |