* chapters. 1-6, 7.1 & 7.2, and 8-9
* For the objective, make sure to know the following, as there were at least three or more questions on each:
* dynamic programming, static programming, functional programming,
* objects & inheritance,
* which programming language does what (specifically understanding what makes C# different from the other C’s, C++, C, Python, just a basic understanding of which compiles and which interprets),
* compiler, interpreter,
* logical operators (==, ===, !=, !==), what the % operator does,
* how to perform basic math using the modulo operator,
* agile, waterfall, and questions on UML, along with identifying the steps shown for each of the design processes.
* Know about program libraries.
* Know the difference between compiled and interpreted languages. Ex: Compiled languages are usually faster. They are usually static. C, C++, Java are all compiled languages. Tip- Compiled has the C languages. Interpreted languages are usually dynamic and do one line at a time, line by line.
* Python is an interpreted language.
* Know about HTML and that it is Markup language.
* Know about the % (Modulo operator) and how to do the math. Ex: 22 % 10 = Mod 2 (10 goes into 22 10 times and 2 is left over.)
* Know how to read flow charts and pseudo code. They will both be on the test.
* Know the different parts of a function statement. Function name Function parameter (input) Return variable (output) And any other parts to the function statement.
* Know the difference between Int, Float, Boolean, and String and also when to use one over the other. Ex: Integer variables are typically used for values that are counted. Floating-point variables are typically used for values that are measured or when dealing with fractions of countable items, such as the average number of cars per household.
* Know valid equality & relation operators: ==, !=, <, >, <=, >=
* Know the difference between a while loop, for loop, and do-while loops and difference scenarios. (Look in the chapter summary for examples and breakdown) <https://learn.zybooks.com/zybook/WGUD278v1/chapter/4/section/11>
* Know about Functions once again. What is the function definition? What is the function call? What is the input for the function? What is the output for the function?
* Know about Algorithms. You will get a few questions about this like putting a sequence of events in order. Ex: 1. Declare variable min. 2. Min=x 3. If y < min, set min = y. 4. Put min to output.
* Know about structural vs behavioral. Structural- Show static items like variations and functions. Behavioral- Shows dynamic behavior like a flow.
* Know about UML diagrams and how they relate to SDLC (System Development Life Cycle) Analysis --Use Case diagram (behavioral) Design --Class diagram (Structural) Implementation -- Activity diagram Testing -- Sequence diagram (behavioral)

1.11 Introduction summary

This chapter's key points included:

* A program consists of instructions (aka statements) that execute one at a time, to get input, process data, and put output.
* A program uses variables to hold data, which may change ("vary").
* A flowchart depicts a program graphically, with a node for each statement.
* A program can output a variable's value, or a string literal consisting of characters (including a newline character).
* A program can contain comments, which are for humans only and ignored when the program runs.
* Most whitespace (regular spaces or newlines) is ignored by a program, but good practice uses whitespace in a consistent way.
* The information age is quite new in human history. Changes are rapid.
* Computers surround us and computer numbers and usage continue to grow.
* Inside a computer, all data (characters, numbers, and more) is represented as bits: 0's and 1's.
* Programming is largely about problem solving, namely creating a methodical solution to a given task.
* Careers in computing are numerous, highly-rated, and growing. Non-computing jobs may benefit from programming.
* Pseudocode is an informal textual representation of a program intended for easy human understanding.

2.16 Variables/Assignments Summary

This chapter's key points included:

* A variable declaration declares a new variable, specifying the variable's name and type.
* An assignment statement assigns the variable on the left-side of the = with the current value of the right-side expression.
* An expression is a combination of items, like variables, literals, operators, and parentheses, that evaluates to a value.
* A name created by a programmer for an item like a variable or function is called an identifier, which must follow certain rules to be valid. Programmers typically follow identifier naming conventions that are defined by their company, team, teacher, etc.
* An expression is evaluated using precedence rules that follow the evaluation order of standard mathematics.
* Incremental development is the process of writing and and testing a small amount of code, then writing and testing a small amount more (an incremental amount), and so on.
* A variable declared as type float stores a floating-point number, which is a real number, like 98.6, 0.0001, or -666.667.
* A programmer should choose a variable's type based on the type of value held. Integer variables are typically used for values that are counted. Floating-point variables are typically used for values that are measured or when dealing with fractions of countable items, such as the average number of cars per household.
* Programming languages typically have built-in functions to perform common operations needed by programmers, such as performing mathematical operations like square root or raising a number to a power.
* A function is a list of statements executed by invoking the function's name, with such invoking known as a function call.
* Programming languages typically have built-in functions for generating random numbers. The integers generated by a random number generator are known as pseudo-random. "Pseudo" means "not actually, but having the appearance of". Internally, the RandomNumber() function has an equation to compute the next "random" integer from the previous one.
* When the operands of / are both integers, the operator performs integer division, which does not generate any fraction.
* For integer division, the second operand of / or % must never be 0, because division by 0 is mathematically undefined. A divide-by-zero error occurs at runtime if a divisor is 0, causing a program to terminate.
* A type conversion is a conversion of one data type to another, such as an integer to a float.
* zyFlowchart, and other programming languages, automatically performs several common conversions between integer and float types, and such automatic conversion is known as implicit conversion.
* If a programmer needs to explicitly convert an item's type, the programmer can use a type cast to converts value of one type to another type.
* The modulo operator (%) evaluates to the remainder of the division of two integer operands.
* A constant is a named value item that holds a value that cannot change. Constants are commonly used in programs to hold the value of mathematical constants or a value that should not change during the program's execution.

3.11 Branches summary

This chapter's key points included:

* In a flowchart, a decision creates two branches, one for when the decision's expression is true (the if branch), another when false (the else branch).
* If-else branches have statements in each branch. An if branch has no statements in the else branch.
* If-elseif branches have cascaded decisions along the false branches. Only one true branch can execute.
* A branch can itself have a decision, known as nested branches.
* Multiple if branches can be created, which are independent, so more than one true branch can execute.
* Valid equality and relational operators are ==, !=, <, <=, >, >=.
* If-elseif branches are commonly used to detect ranges, with the lower end of the range implicit.
* Logical operators are: and, or, not.
* In an expression, operators are evaluated in a specific order based on precedence rules (just like in math).
* Because floating-point numbers aren't represented exactly, they shouldn't be compared for equality (using ==). Instead, they can be compared for "close enough".
* Branches in pseudocode use words like if, else, and elseif, and a branch's statements appear indented starting on a next line.

# **4.11 Loops summary**

This chapter's key points included:

* A loop is a program construct that repeatedly executes the loop's statements (known as the loop body) while the loop's expression is true; when false, execution proceeds past the loop. Each time through a loop's statements is called an iteration.
* A common programming task is to use a loop to examine a list of values one value at a time and update variables along the way. Common tasks include computing an average, counting the number of negative items in a list, finding the maximum, etc..
* Programmers often use loops to execute a computation until a done condition is reached. That done condition is reached when the loop expression is false.
* An infinite loop is a loop that never stops iterating.
* A loop iterating a specific number of times commonly consists of three parts: a loop variable initialization before the loop, a decision statement for the loop expression, and a loop variable update at the end of the loop body.
* A **while loop** is a loop that repeatedly executes the loop body while the **loop's expression evaluates to true**.
* A **for loop** is a loop that typically describes iterating for a specific **number of times**. A for loop consists of a loop variable initialization, a loop expression, and a loop variable update statement.
* A nested loop is a loop that appears in the **body of another loop**. The nested loops are commonly referred to as the **inner loop** and outer loop.
* A **do-while loop** is a loop that first executes the loop body's statements, then **checks the loop condition**. Compared to a while loop, a do-while loop is useful **when the loop should iterate at least once**.

# **5.7 Arrays summary**

This chapter's key points included:

* An array variable stores multiple items, each accessible using an index. In contrast, a scalar variable stores just one item.
* Array indices start from 0, not 1. An index may be represented as an expression.

# **6.7 Functions summary**

This chapter's key points included:

* A function is a named list of statements.
* A function definition consists of the new function's name and a block of statements. The function's name can be any valid identifier.
* A function call is an invocation of a function's name, causing the function's statements to execute.
* A program's execution begins with the Main function.
* A programmer can influence a function's behavior via an input. A parameter is a function input specified in a function definition. An argument is a value provided to a function's parameter during a function call.
* A parameter is like a variable declaration. Upon a call, the parameter's memory location is allocated, and the parameter is assigned with the argument's value.
* A function may return one value by assigning a return variable with the return value.
* A function call evaluates to the returned value. Thus, a function call often appears within an expression.
* Decomposing a program into functions can greatly aid program readability, helping yield an initially correct program, and easing future maintenance.
* Programmers commonly use functions to write programs modularly and incrementally.
* Modular development is the process of dividing a program into separate modules that can be developed and tested separately and then integrated into a single program.
* Incremental development is a process in which a programmer writes and tests a few statements, then writes and tests a small amount more (an incremental amount), and so on.
* A function can be defined once, then called from multiple places in a program, thus avoiding redundant statements.
* The skill of decomposing a program's behavior into a good set of functions is a fundamental part of programming that helps characterize a good programmer.
* Each function should have easily-recognizable behavior, and the behavior of the Main function (and any function that calls other functions) should be easily understandable via the sequence of function calls.
* A function's statements may include branches, loops, calls to other functions, and other statements.

# **7.8 Algorithms summary**

This chapter's key points included:

* A programmer must first create a correct algorithm: A sequence of steps to solve a problem.
* For large data, also relevant is an algorithm's time efficiency: The number of calculations needed to solve a problem.

# **8.6 UML summary**

This chapter's key points included:

* UML uses different kinds of diagrams to visualize programs. A structural diagram shows static items like variables and functions. A behavioral diagram shows dynamic behavior like flow.
* A UML use case diagram is behavioral and shows how a user interacts with a program. A class diagram is structural and shows a program's classes. A sequence diagram is behavioral and shows interactions and event orderings.

# **8.7 Software design processes summary**

This chapter's key points included:

* Programs may be developed in various phases known as the system development life cycle (SDLC).
* Programs may be developed in phases known as the system development life cycle (SDLC), wherein analysis defines a program goals, design defines specifics of how to build a program, implementation writes the program, and testing checks the program.
* A waterfall approach does the phases in sequence once, while an agile approach (aka spiral approach) does smaller amounts of each phase and repeats.

# **9.3 Software languages and libraries summary**

This chapter's key points included:

* Many kinds of languages exist. Compiled languages are first converted to machine code, while interpreted languages instead run on an interpreter. Statically typed languages require a programmer to declare a variable's type, which cannot change, while dynamically typed languages let the type change as a program runs. Object-oriented languages provide substantial support for decomposing a program into objects. Markup languages don't execute, but describe formatting and other features (like HTML, the language of web pages).
* Programmers use libraries to improve productivity, by making use of pre-written functions.