Starting out with C# Notes



Chapter 2 – C# Fundamentals

* C# files end in .cs
* Don’t type .exe extension when executing a program
* A namespace is like a package for code that contains a variety of similar classes.
  + Activated with a using command, then <namespace> then a semicolon;
* public / private are keywords as well as access specifiers
* more than one class is allowed in a file, but only one public class is allowed.
* The public class must have the same name as the file name
* Framework Class Library (FCL) – collection of prewritten classes and methods
* GC – garbage collection (the heap)
* C# uses the .Net class libraries, all non-user defined classes in C# are .net FCL
* The text inside of parentheses of a method call are known as arguments
* Console.write continually streams text
* Console.writeline keeps the statement to one line
* \ - backslash
* / - forward slash
* Console.WriteLine (“The value is {0}”, number); The number inside the curly braces, takes the value of the variable at the position of it based on where it is located after the comma.
* a = a + b is the same thing as a += b.
* A cast operator manually coverts a value. Unary operators appearing as data type in parentheses before a variable. Ex. x = (int)number;
* Double piesPerPerson; int pies = 10; int people = 4;
* piesPerPerson = pies / people; Even though piesPerPerson is a double this will return an int.
* However, if we cast either pies or people, then piesPerPerson will return 2.5 instead of 2.
* piesPerson = (double)pies / people. -Or- piesPerPerson = pies / (double) people. 2.5.
* piesPerson = (double)(pies / people) is bad. This converts the int 2 to 2.0.
* for a constant variable like an interest rate use keyword const before data type, and use all uppercase letters for variable name
* The string type is an object for holding strings
* String literals are enclosed in double quotation marks
* C# uses System.String to represent strings.
* The key word string is used as an alias for the System.String class
* Objects are entities that contain attributes (variables) and methods (functions)
* Before an object is created (instanced), a programmer must design the class or blueprint for an object
* String variables are known as class type variables
* *Class type variables* don’t hold the actual data item, but the memory address of the data
* Class type variables when holding the address of an object is also known as referencing the object
* *Class type variables* are also known as *reference variables*
* Anytime you write a string literal in your program C# will create a string object in memory
* The string literal causes a string object to be created in memory with the value “<name>”
* Then, the assignment operator stores the address of that object in the variable name.
  + name = “Ryan Smith”
  + string object with the value of Ryan Smith is created.
  + The address of that string object is stored in the name variable
  + 2 separate software entities created from a single string
    - The object itself with a string value randomly somewhere in memory
    - The variable name that holds only the memory address of that object
* Because string is a class, it has multiple methods like (name.Length that returns an int)
* Reference variable is the name of a variable that references an object
* Method of that reference variable is the name of the method, and arguments
  + EX. ReferenceVariable.method(argument1, argument2, ect…)
* To retrieve user’s input interactively, call the Console.ReadLine method
* Convert.ToFloat – converts string to float
* Convert.ToInt32 – converts string to int

**Chapter 3 – Methods**

* Void method performs a task then terminates, WriteLine is a void method
* A *value returning method* not only performs a task, but sends a value back to the code that called it.
* A value returning method usually has a variable with an assignment statement that the return value of a that method is assigned to.
* Defining a method –
  + Method modifiers (ex. public, or static) modifies the method with specific properties
  + Return type – determines that type of value returning if any (void).
  + Method name – the identifying name the programmer specifies for that specific method
  + Parameter list – In the header, the parameters are the variable data types and name (usually different from the arguments name) declared to specify to the compiler what types of variables are acceptable for the arguments that are passed to the method during the actual function call.
  + Parameters – declared in header with data type, arguments actual data in method calls without the data types.
* Arguments: values sent into a method during method calls.
* Parameters: special variable that holds a value being passed into a method, created and declared in the header of a function/method definition.
* C# will not automatically convert an argument of a lower-ranking data type
  + From lowest to highest:
  + byte, short, int, long, float, double, decimal
  + char->int
* A data type is required before each parameter listing
* All arguments of the value data types are either *passed by value,* OR *passed by reference*
* *Pass by value* means only a copy of an argument’s value is passed into a parameter variable.
  + In pass by value, a method’s parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If/when parameter variables are changed within a method, it has no direct effect on the original arguments.
* *Passed by reference* means that a reference variable was defined as parameter in the function definition, and this reference variable allows for direct access to the original argument.
* A reference variable acts as a direct alias to the argument it’s accessing, and anything changed to the reference parameter variable inside the function, changes the original argument that is passed to it during a function call.
* Using a reference variable as a parameter, allows a function to change variable that is declared inside of another function, thus breaking scope
* Reference variables are declared the same way any other variable is declared except that the key word ***ref*** is placed in the front of it.
* String objects are immutable, meaning, they can’t be changed even though they are a reference variable by nature.
* A recursive function puts a call to itself inside of its function definition. There must be a way to control how many times it is called, usually place inside the definition with an if statement.
* Formatting numbers can be done inside of the curly braces of a statement (the placeholder for the position for which variable will be entered in the statement) by placing a colon, then the type of format (c for currency, d for decimal, e for scientific notation, f for fixed, g for general, n for number, p for percent, and x for hexadecimal), and lastly the number of places.
  + For ex. {0:C2} – would be the variable after the comma in the zero position, set to the currency type, and rounded to two decimal places.

**Chapter 4 – Classes and Objects**

* *Encapsulation –* combing data and code into a single object
* *Data hiding –* an object’s ability to hide its data from code that is outside the object.
* Only objects methods can be accessed directly to make changes to the object’s data, but the data itself cannot be directly manipulated.
* *Access Specifier –* indicates how the class may be accessed.
* Set functions for classes are void, Get functions are a return type of the data being passed in.
* Avoid Stale data – In the case, where a field was created for area, when data is dependent on other data, like area, and the other data is changed in code, but box.GetArea() isn’t also called in the main function, then it will still have the old data in it.
  + When calculation functions (like GetArea) rely on fields that are set in other functions (ex. SetLength), then you always want to return the field calculation versus storing a variable that contains that data. Double area as a variable would be a bad practice. Return length \* width would be good practice in a GetArea function.
* Each instance of a class has its own set of fields
* Instance methods do not have the key word static in their headers
* A ***constructor*** is a method that is automatically called when an object is created
  + Constructor usually perform initialization or setup operations
  + Constructor object has the same name as the class
  + Constructor header doesn’t specify return type
* *Uninitialized reference variable* is an object that is declared but not initialized (aka the = new object isn’t added to the end of the declaration)
  + Local variables must be assigned a value before they can be used
  + Local reference variables must reference an object before it can be used
* A default constructor is always created when the programmer doesn’t create his own constructor
  + If a constructor is defined that requires parameters, then when a new object is instanced, then it needs to provide the exact number of parameters provided in constructor definition
  + Default constructor sets numeric fields to 0, Booleans to false, and if any object fields are reference variables, those variables are set to null.
* A constructor that doesn’t accept arguments is a *no-arg constructor*
* A class can have multiple constructors of the same name as long as their parameter lists are different. This is known as *overloading.*
* *Binding* is calling the correct overloaded method or constructor based on the parameters
* C# uses a method’s signature to distinguish it from other methods of the same name
  + *Method signature* consists of method’s name, and data types of the method parameters in the order that they appear in the parameter list.
  + The return type of a method is not considered part of its signature.
* If an instanced field is declared with the public access specifier, it can be accessed by code outside of the class
* Cannot give a parameter variable and a local variable in the same method the same name.
* However, you can have a local variable or a parameter variable with the same name as a field.
  + This is known as *shadowing*, and is generally a dangerous practice.
* Set methods are known as *mutator methods. (setters)*
* *Get* methods are known as accessor methods. (getters)
* Properties can be used to define a class instead of methods for getters and setters.
  + <modifier><return type><Propertyname>, then open braces for whole property
  + Inside the property, there is a get property, followed by braces, inside that is the return <variable> followed by a semicolon
  + Same thing for set, except the instance field is equal to value.
* For Length

public double Length

{

get

{

return length;

}

set

{

length = value;

}

}

* You can use this property notation instead of defining separate mutator and accessor methods for each instanced field of a class.
* Instance fields that use properties to get and set data are accessed via outside code differently
  + box.Length = 5.0; instead of box.SetLength(5.0);
  + there is no () – parentheses used when accessing field via class properties
* **Common Errors**
  + Putting a semicolon at the end of a method header
  + Declaring a variable to reference an object, but forgetting to use the **new** keyword to create the object
  + Forgetting the parentheses that must appear after the class name, which appears after the new keyword
  + Forgetting to provide arguments when a constructor requires them
  + Inserting a space before or after the period in a method call
  + Trying to overload methods by giving them different return types
  + Forgetting to write a default constructor for a class that you want to be able to create instance of without passing arguments to the constructor
  + In a method, unintentionally declaring a local variable with the same name as a field of the same class

**Chapter 5 – Decision Structures**

* *If* statement causes one or more statements to execute only when a Boolean expression is true
* The keyword *bool* is short for Boolean
* Relation operators can be used to test character data
* In Unicode, uppercase letters come before lowercase letters
* *If-else* statements execute one group of statements if its Boolean expression is true, or another group if its Boolean expression is false.
* *If, else-if* statements are a chain of *if* statements. Each statement in the chain performs its test until one of the tests is found to be true.
  + If, else-if statements are interconnected, and each statement is analyzed top down, and as soon an else if statement becomes true, all else if statements below that are ignored.
  + The block ends at the last else-if, versus if statements, which by themselves are never ignored.
  + If statements are single statements, else-if statements are interconnected
* *Nested ifs* is an if statement in the conditionally executed code of another if statement
* **Comparing Strings**
  + *==* . if (name1 == name2), overloading the equal sign, double equals
  + Use string’s class *Equals method* to see if the objects contents are the same
    - StringReference1.Equals(StringReference2)
    - If (name1.Equals(name2)
    - Comparing objects to string literals is also legal
    - If (name1.Equals(“Mark”) – equals
    - If(!name1.Equals(“Mark”) – not equals
  + *CompareTo* can be used to determine whether one string is greater than, equal to, or less than another string.
    - StringReference.CompareTo(OtherString)
      * If method returns -1, StringReference is < OtherString
      * If method returns 0, the two strings are equal
      * If method returns 1, then StringReference is > otherString
      * Compare(str1, str2, ignoreCase) – ignores case sensitivity
* The scope of the variable is limited to the block in which it is declared
* **Conditional operator**:
  + Expression1 ? Expression 2 : Expression 3;
  + Something always changes in a conditional operator, ternary operator
  + If expression 1 is true, then expression 2 gets assigned or executed, but if expression 1 is false, then expression 3 gets assigned or executed instead
  + number = x < 0 ? 20 : 50;
    - if x is less than 0, than number = 20, if x is not less than 0, than number = 50.
* **Switch operator**:

switch (SwitchExpression)

{

case CaseExpression:

// body of code

break;

case CaseExpression:

// body of code

break;

default:

break;

}

Example –

Int number;

number = Convert.ToInt32(Console.ReadLine());

switch (number)

{

case 1:

Console.WriteLine(“You entered 1.”)

break;

case 2:

Console.WriteLine(“You entered 2.”)

break;

default:

Console.WriteLine(“You didn’t enter 1, 2, or 3);

break;

}

* There always needs to be a break after the default switch statement
* Switch statement is exited once it reaches a break
* char characters can be used
* each case can be overloaded, for example
* case ‘a’:
* case’A’:
* Can be written on succeeding lines without any code in between them, which means if either a, or A is entered then it will execute the block of code for either entry.

**Common Errors**

* Using = instead of == to compare primitive values. = is the assignment operator
* Forgetting to enclose an if statement’s Boolean expression in parentheses
* Writing a semicolon at the end of an if clause
* Forgetting to enclose multiple conditionally executed statements in braces
* Omitting the trailing else in an if-else-if statement. While not a syntax error, can quickly cause logic errors
* Not writing complete Boolean expressions on both side of a logical && or || operator
  + X > 0 && < 10 is bad, x > 0 && x < 10 is good. Include the variable on both sides
* Using a Switch Expressions that is not an integral data type (aka use integers or chars)
* Using a CaseExpression that is not a literal or a final variable.
* Forgetting to write a colon at the end of a case statement
* Failing to write a break statement in a case section
* Omitting a break in the default section of a switch statement
* Reversing the ? and : in a conditional operator

**Chapter 6 – Loops and Files**

* The difference between prefix and postfix
  + Postfix mode causes the increment to happen after the value of variable is used
  + Prefix modes causes the increment to happen before the value of variable is used
* While loop is a pretest loop, which tests its expression before each iteration.
  + Commonly used for validating data
* Do-while loop is a posttest loop, which means it always runs at least once, and then the expression is tested at the end of the code. Same thing as a while loop, except it has to run at least once
* For loop is a user-controlled loop, most ideal loop when number of iterations is known
  + For (variable initialization; Test expression; increment or decrement variable)
* A sentinel value signals the end of the input or list when the number of iterations is knows
  + Ex. 30 days of sales, run report for each day. 30 is the sentinel value.
* Nested loops are loops inside other loops
* The break statement (optional) in a loop causes a loop to terminate early.
* The continue statement in a loop causes a loop to stop its current iteration and begin the next one

**Files**

* To write data to a file, you use the *StreamWriter* class.
* To read data to a file, you use the *StreamReader* class.
* **Using.System.IO;** must be placed at the top of your program to use those two classes
* *Delimiter* is an item that separates other items
* StreamWriter <reference name> = new StreamWriter (<output file name>);
* StreamWriter <reference name> = new StreamWriter (<output file name>, true); append file
* StreamReader inputFile = new StreamReader (filename);
  + The string that is returned from the ReadLine method will not contain the newline character

**Common Errors**

* Using the increment or decrement operator in the wrong mode
* Forgetting to enclose the Boolean expression in a while loop or a do-while loop inside parentheses
* Placing a semicolon at the end of a while or for loop’s header
* Forgetting to write the semicolon at the end of the do-while loop.
* Using commas instead of semicolons to separate the initialization, test, and update expressions in a for loop
* Forgetting to write code in the body of a while or do-while loop that modifies the loop control variable.
* Using a sentinel value that can also be a valid data value
* Forgetting to initialize an accumulator to zero

**Chapter 7 – Arrays**

* An array is an object that can store a group of values, all of the same type.
* Using arrays means declaring a reference variable, and new key word to create an instance of the array in memory
* Int [] numbers; - creates an array reference variable only, a variable that reserves memory space
* Int [] numbers = new int[6]; creates an array reference variable, and an array object
* The number inside the brackets I the array’s size *declarator*.
* Declarator is the number of elements the array can hold, can be a literal value, or another variable. Declaring a separate integer constant (const) value and using that as the declarator in an array is common practice.
* Brackets must come after the data type of the array, and not after the name of the array.
* *Subscript* each element of the array is assigned a number, known as that element’s subscript.
* The first subscript is always 0. Element 0 is the start of any array.
* The last subscript is always one less than the total number of elements. Declarator – 1.
* numbers[0] is pronounced numbers sub zero.
* numbers[3] = 30; is pronounced numbers sub three is assigned 30.
* By default, C# initializes all elements of an array to the value of zero.
* An array subscript must have an int value - numbers[2.5] is illegal
* It is possible to reassign an array reference variable to a different array
  + Int[] numbers = new int[10];
  + Int[] numbers = new int[5];
* The array and the reference for an array are two separate entities.
* Int[] array1 = array2; *Shallow copy*, only the addresses are copied. Now both array 1 and array 2 are referencing the memory addresses from array 2.
* *Deep Copy* copies the contents itself, usually using a for loop. Both arrays will maintain separate references after a deep copy, keeping each one unique.
  + Int[] firstArray = {5, 10, 15, 20, 25};
  + Int[] secondArray = new int[5];
  + for (int i = 0; i < firstArray.Length; i++)
    - secondArray[i] = firstArray[i]
* *foreach loop*
  + double[] temperatures = {51.2, 55.6, 46.7, 49.8, 50.2, 53.7, 56};
  + foreach (int temperature in temperatures)
    - Console.WriteLine(temperature);
* To pass an array as an argument to a method, you pass the variable that references the array
* Individual elements of arrays can be passed to a method.
  + In the case of an int[] number array with int elements
  + The method would accept int as the parameter, not the whole array
* Entire Array objects can also be passed as arguments as well (passed by reference)
  + Arrays are passed just like objects, in that the variable that references the actual array object is passed, not the object itself. It is passed in as a reference, allowing for direct manipulation of the original object.
* An array can be returned from the method call, but the variable reference that is going to store the array has to be of the same data type from the array manipulated inside the method.
* BankAccount [] accounts = new BankAccount[5]; - This is an array of objects
  + All elements of an objects array are initialized to null
  + You must individually create the object for each element in the array
  + for (int i = 0; I accounts.Length; i++)
    - accounts[i] = new BankAccount();
* each element of the bank account object will have individual references
* An array of objects is essentially and array of references
* Elements of an object array are accessed via subscripts like any other data type of array
  + accounts[2].SetBalance(2500.0);
* *Popular search* algorithms are the sequential and binary search
* *Selection sort* is the most common sorting algorithm
* ***Selection sort***takes the smallest value in the array and finds out where it is located. Then, that element is moved to element 0 (first array element).
  + *Pseudocode:*
  + *For() startScan equals each subscript in array 0 through next to last subscript*
  + *Set index variable to startScan*
  + *Set minIndex variable to startScan*
  + *Set minValue variable to array [startScan]*
  + *For() index is each subscript in array from (startScan + 1) through last subscript*
    - *If array[index] is less than minValue*
      * *Set min Value to array[index]*
      * *Set minIndex to index*
      * *End if*
    - *Increment the index*
  + *End inner for loop*
  + *Set array[minIndex] to array [startScan]*
  + *Set array[startScan] to minValue*
  + *End outer for loop*
* ***Binary Search*** is a search algorithm that (a) requires the array to be in ascending order, then (b) cuts the list in half, and compares the middle number to the requested number, then (c) if the number found is less than what is requested, the binary search uses the larger half remaining, and then splits that one, and (d) repeats this process until it finds a match.
* There are two types of Multidimensional arrays: rectangular and ragged.
* ***Multidimensional array*** – double[,] scores = new double [3, 4]; - array of arrays
  + Each two dimensional array has two subscripts, one for its row, and another for its column
* To get the total length of a row – scores.GetLength(0);
* To get the total length of a column – scores.GetLength(1);
* Passing a 2 dimensional array – public static void ShowArray(int[,] array)
* To access data from a 2 dimensional array, use a nested for loop, and for the inner for loop be sure to include both variables. Ex. scores[row, col] – this will retrieve an individual column value one at a time. Starting at the top row, left to right, top to bottom.
* *Ragged array –* when a two dimensional array has a different number of rows vs columns
* Int[][] ragged = new int [4][];
  + ragged[0] = new int[3]; // Row 0 has 3 columns
  + ragged[1] = new int[4] // Row 1 has 4 columns ect….
* For (int I = 0; I < ragged.Length; i++)
  + Console.WriteLine(“The number of columns in row {0} is {1}”, I, ragged[I}.Length);
* Multidimensional arrays (beyond 2)
  + double[,,] seats = new double [3, 5, 8]; - 3 sets (or planes) of 5 rows and 8 columns.

**COMMON ERRORS**

* Using an invalid subscript
* Confusing the contents of an integer array with the elements subscript
* Causing an off by one error
* Using the = to copy an array
* Using == to compare two arrays
* Reversing the row and column subscripts when processing a two-dimensional array

**Chapter 8 – Classes and Object part 2**

* A static class member belongs to the class, not objects instantiated from the class.
* Both instanced fields and instanced methods are associated with a specific instance of a class, and they cannot be used until an instance of the class is created.
* Fields or methods that don’t belong to any instance of a class are known as *static fields or static methods.*
* Static methods can only operate on static fields.
* Static methods or static fields belong to the class directly, instead of an instance of that class.
* The keyword static is used after the access specifier and in front of the data type to make the variable static.
  + Only one copy in memory of that static variable, regardless of how many instances
  + Initialization of a static takes place only once no matter how many instances there are
  + C# initialized all uninitialized static variables to zero.
* A static method can be called without any instances of the class being in existence.
* To use a static method, you simply write the name of the class before the dot operator
* Kilometers = Metric.MilesToKilometers(10.0); Where Metric is the name of the class
* Static methods are most often used to create utility classes that perform operations but don’t store data. A converter class would be helpful for something like this.
* The only limitation that static methods have is that they cannot refer to non-static members of the class.
  + Any method called from a static method must also be static
  + If a static method uses any of the class’s fields, they must be static as well.
  + You can use static methods on instanced data, but an object will have to be included in the call.
* Classes benefit greatly from ToString and Equals methods
  + Equals is used to compare the contents of objects
  + Sometimes an object must have methods that perform same class operations
  + Meaning, methods accept, as arguments, objects of the same class as itself, and or returns objects of the same class as itself.
  + This is also known as same class operation
* The Equals method from the string class accepts a reference to a string object as an argument.
* Aggregation occurs when an instance of a class is a field in another class.
* Be careful of aggregation because when a class has a field that is an object, it is possible to create a ‘security hold’ by having a method that returns a reference to the private field.
  + When a method returns a reference to a field, any variable outside the object that receives the reference can provide access to the field.
  + To prevent this security hole, avoid writing methods that return references to objects that are private fields.
  + Best to use the properties instead of public methods when you need to return data from private fields.
* The keyword ***this*** is the name of a reference variable that an object can use to refer to itself.
  + Available to all non-static methods.
  + Commonly used to overcome shadowing a field name.
  + Also used to call one constructor from another constructor in the same class.
  + FeetInches(int f): this (f,0){}
* Namespaces allow access to other blocks of classes organized under one umbrella
  + Using <namespace>; at the beginning of your code to access the classes from that namespace
  + System – general classes of C# namespace
  + System.Drawing – classes used for drawing images and creating GUI
  + System.IO – for input and output
  + System.Net – classes for network communication
  + System.Text - for formatting text
  + System.Windows.Forms – support for windows application
  + System.XML – provides classes supporting XML
* **Common Errors**
  + Attempting to refer to an instance field or instance method in a static method or field
  + In a method that accepts an object as an argument, writing code that accidently modifies the object
  + Allowing null references to be used.

**Chapter 9 – Text Processing, Exceptions, and More Files**

* Learn: methods of Char class, String class
* Learn: difference between string object and string builder object
* Using the split method to extract tokens from a string
* Learn: what an exception is
* The Char class is an alias for the char data type. <uses system namespace>
  + Bool IsDigit(char ch)
  + Bool IsLetter(char ch)
  + Bool IsLetterOrDigit(char ch)
  + Bool IsLower(char ch)
  + Bool IsUpper(char ch)
  + Bool IsWhiteSpace(char ch)
* *Substring* a string that is part of another string
  + Bool StartsWith(string str) – returns true if the string starts with substring provided
  + Bool EndsWith(string str) – “” ends “”
* String methods for getting a character or substring’s location:
  + Int IndexOf(char ch) – searches a string for the char passed in and returns first occurrence, otherwise returns a -1.
  + Int IndexOf(char ch, int start) – search a string starting at the int start position of the string and returns the first occurrence of the char passed in, otherwise returns -1.
  + Int IndexOf(string str) – searches a string for the string passed in and returns first occurrence of that string, otherwise returns – 1.
  + Int IndexOf(string str, int start) – same as above, just from the starting point specified of the second argument, start.
  + Int LastIndexOf – same set of 4 methods above, except the keyword Last is put in front of the method name, and it returns the last occurrence, instead of the first occurrence.
* String class allows you to retrieve a substring from a string:
  + string Substring(int start) – returns a copy of the substring that begins at start and goes to the end of the calling object’s string.
    - string fullName = “Cynthia Susan Lee”;
    - string lastName = fullName.Substring(14);
  + string Substring(int start, int end) – returns a copy of the substring from start to end positions.
  + char[] ToCharArray(int start, int end) – method returns all of the characters in the calling object as a char array. Same as above instead the return type is a char array instead of a string.
* Methods that return a modified copy of a string object
  + string Concat(string str) – method returns a copy of the calling string object with the contents of t str concatenated to it.
    - String fullName, firstName = “Timothy “, lastName = “Haynes”;
    - fullName = string.Concat(firstname, lastName); - this can also be written as
    - fullName = firstName + lastName;
  + string Replace (char oldChar, char newChar) – returns a copy of the calling string object, in wich all occurences of the character passed into oldChar have been replaced by the character passed into new Char.
    - string str1 = “Tom Talbert Tried Trains”;
    - string str2 = str1.Replace(‘T’, ‘D’);
  + string Trim() – returns a copy of the calling string object in which all leading and trailing whitespace characters have been deleted.
* StringBuilder class is similar to a string class, except that you can change the contents of a StringBuilder class, and provides additional methods.
  + String objects are immutable, once you set the contents of a string object, you cannot change the string value that it holds.
    - The previous string methods were for taking the contents of a strings and copying them into a new object, not changing the actual contents
    - You can reference another string with the same variable name
      * string name;
      * name = “George”;
      * name = “Sally”;
    - The second statement creates a string object containing “George”, and it assigns its address to name.
    - Although we cannot change the contents of the string object, we can make the reference variable ‘name’ point to a difference string object. This makes the string object George to be de-referenced and collected by the G.C
    - G.C. = garbage collector. Aka the ‘heap’. Automated overhead that keeps your programs from having memory leaks.
* StringBuilder objects on the other hand have methods that allow you to modify their contents without having to create a copy (or new) object in memory.
  + StringBuilder should be used when you have a lot of string objects that need a lot of changes made to them.
  + StringBuilder() – The constructor accepts no arguments. It gives an object enough storage space for 16 characters, but no characters are stored in it initially.
  + StringBuilder(int length) – constructor gives the object enough storage space to hold *length* characters, but not characters are stored in it.
  + StringBuilder(string str) – constructor initializes the object with the string in str. The object will have at least enough storage space to hold the string in str.
* StringBuilder objects can be passed to Write and Read console commands the same as strings.
* ***YOU CANNOT*** use the same shortcut notation for stringbuilder objects that you do for strings
  + StringBuilder city = “Charleston”; -- Will NOT compile
  + StingBuilder city = new StringBuilder(“Charlestown”); -- Will compile
* StringBuilder Append: - add the data to the end of the string
  + General form: object.Append(item);
  + Primitive data types, character arrays, and other strings are accepted as arguments
* StringBuilder Insert:
  + General form: object.Insert(start, item);
  + Int start specifies the position in the calling object’s string where the insertion begins
  + The value inserted can be primitives, char arrays, and other strings
* StringBuilder Replace:
  + General form: object.Replace(start, str);
  + While string’s replace method replaces the occurences of one character with another character, the StringBuilder class’s Replace method replaceds a specific substring with a string.
* StringBuilder Remove:
  + General form: object.Remove(int start, int end)
  + Used to delete a character or substring from a StringBuilder object.
* The Split method:
  + The String class Split method breaks a string down into its components, which are called *tokens*
  + Whitespace that separates words in a string phrase is known as the *delimiter*.
    - These individual words that are part of a string phrase separated by delimiters are known as *tokens*
  + The process of breaking a string into tokens is known as *tokenizing.*
  + Token delimiters need to be declared in a character array before you can run a split method.
  + string strToken = “One Two Three”;
  + char[] separate = {‘ ‘, ‘\n’};
  + string[] result = strToken.Split(separate);
    - foreach(string token in result)
    - Console.WriteLine(“Values: “ + token);
* Trim then Split:
  + string name = “ one;two;three “;
  + string temp;
  + char[] separate = {‘;’};
  + temp = name.Trim();
  + string[] valueArray = temp.Split(separate);
  + foreach(string delims in valueArray)
    - Console.WriteLine(delims);
* ***Exception***is an object that is generated as a result of an error of an unexpected event.
* To prevent exceptions from crashing your program, write code that *throws* an exception.
* *Exception handler* is a section of code that gracefully responds to exceptions when thrown.
* if your code hasn’t been manually written to look for exceptions, the *default exception handler* deals with it, and prints an error message, and crashes the program.
* Exception objects are created from classes in C# CLR.
  + ArithmeticException | IO.IOException -> ApplicationException | SystemException -> Exception -> Object (general hierarchy)
  + All classes in the hierarchy inherit from the Exception class in System namespace
* All exceptions that you will handle are instances of classes that inherit from SystemException
* IOException serves as a base class for exceptions that are related to input and output operations
* ArithmeticException serves as a base class for exceptions that result from arithmetic errors. /0
* ***Handling an Exception:***
  + Use a *try* statement:
    - try
    - {
      * (try block statements)
    - }
    - catch (ExceptionType ParameterName)
    - {
      * (catch block statements)
    - }
* First, the key word ‘*try’* appears.
* The block of code after this is known as the *try block*.
  + One or more statements that are executed and can potentially throw an exception
  + Try block is protected because application will not halt if it throws an exception
* After try block, the catch clause appears which begins with the key word *catch*
  + Followed by the code (Exceptiontype parameterName) – parameter variable declaration, where ExceptionType is the name of an exception *class* and parameterName is a variable name.
  + If code in the try block finds (‘throws’) an exception of the ExceptionType class, then the parameter variable will reference the exception object
  + And…the code that immediately follows catch clause is executed. The ‘*catch block*’.
* Example of trying to open a MyFile.txt with the StreamReader class:
  + try
  + {
    - StreamReader freader = new StreamReader(“MyFile.text”);
  + }
  + catch (FileNotFoundException e)
  + {
    - Console.WriteLine(e.Message);
  + }
  + If this code throws exception, the C# CLR searches for a catch clause that can deal with exception.
  + In order for a catch to deal with an exception, its parameter must be of a type that is compatible with the exception’s type. (aka the type of exception thrown in the try block, needs output a compatible class type that the FileNotFoundException is able to read)
    - Different try blocks find different Exception types. The class Exception type declared in the catch cause must match up with the type of exception found in the try block.
* For many cases the code in the try block will be capable of throwing more than one type of exception.
  + In a case for *multiple catch exceptions* you need to write a catch clause for each type of exception that could potentially be thrown
* **Exception Handlers to Recover from Errors**
* When in a try statement handling multiple exception and some of the exceptions are related to each other through ***Inheritance,*** then the more specialized exception classes should be handled before the more general exception classes.
* **The *finally* Clause**
* The try statement has an optional *finally* clause which must appear after all of the catch clauses
  + - try
    - {
      * (try block statements)
    - }
    - catch (ExceptionType ParameterName)
    - {
      * (catch block statements)
    - }
    - finally
    - {
      * (finally block statements)
    - }
* The *finally* block is one or more statements that are always executed after the try block has executed, and after any catch blocks have executed ***if*** an exception was thrown.
* The statements in a finally block occur regardless of an exception being thrown or not.
* *Customized* message strings can be thrown:
  + General form: throw new ExceptionType (MessageString);
    - MessageString is an optional string argument
  + throw new System.ExceptionType (“Cannot divide by zero.”);
* Hierarchy for exception handling:
  + When an exception is thrown, it must be handled by the program or by the default exception handler
  + First, a compatible exception handler inside the method itself (if manually programmed)
  + Second, the code is passed to the previous method in the call stack (*the offending method)*, and checks to see if there is a try, catch block in that method.
  + Third, if the 2nd method (or the first offending method) doesn’t have a try, catch block, then it keeps going up the chain, until it reaches the Main method.
  + If the Main method does not handle the exception, then the program is halted, and the default exception handler, handles the exception.
* **Advanced File Operations**
  + ***Binary Files*** are files consisting of raw binary data (1s and 0s)
  + ***Random Access Files:*** are files that allow a program to read data from any location within the file, or write data to any location within the file.
* Binary files cannot be opened in a text editor. Fewer conversions in a binary file.
* Writing data to a binary file
  + Create an object from the BinaryWriter class <contained in the System.IO namespace>
  + BinaryWriter outputFile = new
  + BinaryWriter(File.Open(“MyInfo.data”, FileMode.Truncate));
* New BinaryWriter(File.Open(filename, FileMode.OpenOrCreate));
* To instantiate a binary writer object, I need call the File.Open method in the new declaration of it, and pass the appropriate 2 arguments into File.Open in order to create the binary file.
* **Random Access Files –** a file where a program may immediately jump to any location in the file
* To work with RAF, you use the FileStream class in the System.IO namespace
* FileStream randomFile = new FileStream(filename, FileMode.open, FileAccess.Write);
* A file that is opened or created with FileStream is treated as a binary file.
* The FileStream class treats a file as a stream of bytes.
* Internally, the FileStream class keeps a long integer value known as the *file pointer.*
* **The file pointer** holds the byte number of a location in a file.
* When a file is first opened, the file pointer is set to 0.
* When an item is read from the file, it is read from the byte that the file pointer points to.
* Writing also occurs at the location the file pointer is pointed at.
* If the file pointer points to a byte somewhere in the middle of the file where data is already stored, the write operation will cause data to be written over the existing data at that location.
* Filestream class allows you to move the file pointer.
* To use the **file pointer,** use the **Seek** method.
* randomFile.Seek(6, SeekOrigin.Begin);
* // Read the character stored at this location and display it. Should be the letter f.
* ch = (char) randomFile.ReadByte();
* Console.WriteLine(ch);

**COMMON ERRORS**

* Trying to use string comparison methods such as StartsWith and EndsWith for case insensitive comparisons
* Thinking of the first position of a string as 1
* Assuming that all statements inside a try block will execute.
* Getting the try, catch, and finally clauses out of order.
* Writing two catch clauses that handle the same exception in the same try statement.
* When catching multiple exceptions that are related to one another through inheritance, listing the more general exceptions first.

**Chapter 10 – Inheritance**

***Inheritance –*** allows a new class to be based on an existing class. The new class inherits the members of the class it is based on.

* When an object is a specialized version of another object, there is an *is-a relationship* between them. Inheritance creates an is-a relationship between classes.
* *Base class –* general class (top level) (superclasses)
* *Derived class –* specialized version of the base class. Specialized class. (subclasses)
* A colon is used to indicate that a class inherits from a base class
  + Public class FinalExam : GradedActivity
  + <derived class > : <base class> - whatever class the file refers to will come after the keyword ‘class’, and if it inherits from another class, that class will be written last after the colon. The declared class (derived class) will always come first, then base class in header, with a colon in between them.
* Private members of a base class cannot be accessed directly by the derived class.
  + Only the base class’s methods can modify those private values
  + However, the private variable will still exist in memory for the derived class
* Derived class variables can be passed into base class methods that are inherited.
* Any public method in a base class is automatically inherited in the derived class
  + For example, score is a private field in the GradesActivity class
  + No method can be written in the derived class (Exam) that can access that variable
  + However, if there exists a public method in the base class that permits access to that private field, then the derived class can use that method as if it was its own to manipulate the private field score.
* **Constructor Inheritance**
* The base class constructor always executes before the derived class constructor.
* Base class cannot call methods or use fields from a derived class.
* The ‘*base’* keyword refers to an object’s base class.
  + You can use the ‘base’ keyword to call a base class constructor.
* In the case where there doesn’t exist a no-arg or default constructor in the base class, the derived class must use the ‘base’ key word to explicitly call a base class constructor.
  + Public class Subclass2 : SuperClass2
    - //constructor
    - Public Subclass2() : base(10)
    - {
    - }
    - In this case SuperClass has a constructor with an int value as a parameter.
  + Base class constructors can only be called in the derived class constructor’s header.
  + If not explicitly defined in the derived class’s constructor header that it will be calling an overloaded constructor from the base class, than the default or no arg constructor is automatically called in the constructor of the derived class : base()
* If a base class does not have a default or no arg constructor (in the case where only a single constructor was written with parameter values being passed), then the derived class must call a constructor that the base class does have.
* A derived class may have a method with the same signature as a base class method. In such a case, the derived class method overrides the base class method.
* To **override a base class’s method**
  + public virtual void <method name> must be changed in the base class’s method
  + public override void <same name as base method> must be declared in derived class
* **Overloading vs. Overriding**
  + Overloading a method occurs when a method has the same name as another, but a different parameter list.
  + Overriding occurs when they have the same name, and the same signature list.
  + Overloading and Overriding occur in inheritance.
    - A method in a derived class can overload a method in a base class.
  + Overriding can only occur through inheritance, and not on itself within a single class.
* **Protected members**
  + Protected members of a class may be accessed by methods in a derived class.
  + Protected members are a third access specifier that aren’t quite private or public
  + They aren’t private because derived classes can access the protected member directly
  + They aren’t quite public because access to them are restricted to their class or derived classes.
* When an access specifier isn’t defined, C# defaults it to private
* C# FCL has a class named Object, which all classes directly or indirectly are derived from.
* **POLYMORPHISM**
* A base class reference variable can reference objects of a derived class.
* A reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are derived classes of its type.
* GradedActivity exam; (Graded activity is the base class from this chapter)
* exam = new GradedActivity(); exam variable is used to reference a GradedActivity object
* GradedActivity exam = new FinalExam(50, 7);
  + Because FinalExam is not just a FinalExam object, but also a GradedActivity object through inheritance, we can use a GradedActivity reference variable to reference a Final Exam object!
* Although a GradedActivity variable can reference objects of any class that extends GradedActivity, since it is still GradedActivity variable, it only has access to the original 3 methods in its base class, regardless of what class it references.
* The reason you use Polymorphism is for overriding methods in derived classes
  + When a polymorphic reference variable is of a base class type but references a derived class, and they both share an overridden method, the referenced object overridden method gets called instead of the base class method of the same name and signature.
    - This is the only time a base class variable type can access methods from within a derived class.
    - This process of the CLR determining at runtime which overridden methods gets call is known as *dynamic binding, or late binding.*
* Parameter object references can also be used polymorphically.
* Polymorphism is possible top down, but not bottom up
  + GradedActivity activity = new GradedActivity();
  + FinalExam exam = activity; XXX Bad – this will cause the compiler to yell at you
  + You can’t assign a lower tier (derived class) variable to reference a top tier (base class) object.
* **Abstract Classes and Methods**
* An abstract class is not instantiated, but other classes extend it.
* An abstract method has no body and must be overridden in a derived class.
  + An abstract method is a method that appears in a base class but expects to be overridden in a derived class
  + *Access Specifier* abstract *ReturnType MethodName* (ParameterList);
  + The keyword ‘*abstract’* appears in the header, and the header ends in a semicolon.
  + Ex: public abstract void SetValue(int value);
  + Abstract methods are used to ensure a derived class implements a method
  + *When a class contains an abstract method, you cannot create an instance of the class.*
  + Abstract methods are most common in abstract classes.
* An **Abstract Class** is not instantiated itself but serves as a base class for other classes.
  + Public abstract class *ClassName*
* An **Interface** specifies behavior for a class.
* An **Interface** sets a blueprint for a class to follow.
* It doesn’t actually have any code in it, but it has methods and signatures that need to be defined exactly as has been defined in the interfaces method signature.
* It’s kind of like a reminder for creating a class, that a class is implementing an interface that has these methods with these variables types that need to be defined in the class.
* Usually the name of the interface, and the variables give the programmer a clue as to the intent of the methods defined in the interface.
* Interfaces act like base classes (even though they aren’t, they are interfaces) for other ‘real’ base classes. Difference being that an actual base class can implement multiple interfaces, whereas there is only one true ‘base’ class.
* A generic method within main can pass in an interface item which would allow for multiple types of objects or classes to access that method directly (granted that they both implemented that interface in their class definition, and properly defined the interface method inside the class file).
  + An interface is very similar to an abstract class in that it cannot be instantiated
  + An interface intends to serve as a base class
  + An interface has ***all*** abstract methods
  + ***All*** methods written in an interface must be written elsewhere
    - public interface *InterfaceName*
    - *{*
    - *(Method Headers..); note:* they end in a semicolon
    - *}*
    - public class FinalExam3 : GradedActivity, Relatable
  + Interfaces can contain methods or properties, but they are abstract
* Difference between an abstract class and an Interface?
  + An abstract class can extend only one base class, where a class can implement multiple interfaces
  + public class MyClass : Interface1, Interface2, Interface3