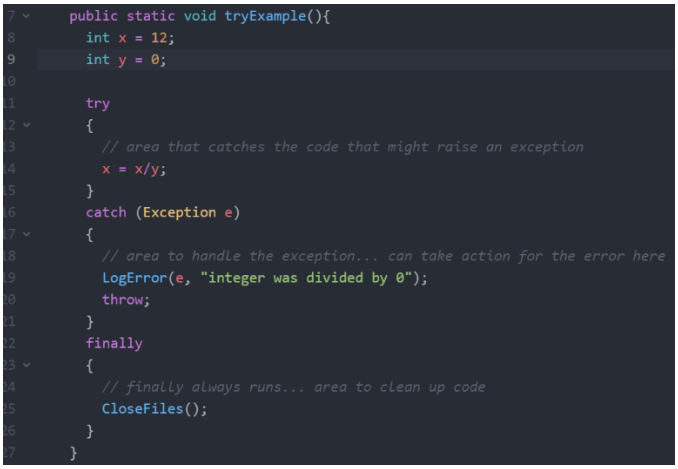
Basic Detail II

The programming language C# is defined by many details, some being unique to itself while other details taking inspiration from other languages. The previous essay covered details about scoping, type system, and memory management structure. C# has more details that further define its language. C#'s control flow structure has many selection, iteration, and jumping statements that can be used. Functions and parameter passing are important details to understand in creating an efficient program in C#. These details help further define C# as a programming language.

Control flow structures allow programmers to manage the order in which statements are executed. These structures provide better control and allow for more complex logic. C# utilizes selection statements, looping statements, and jumping statements to handle control flow. Some recognizable selection statements that can be used in C# are if-else statements, nested if-else statements, and select-case statements. One selection statement that is different than the standard ones is the try-catch-finally statement. In Microsoft Learn, Decision Statements (Visual Basics), Kathleen Dollard says, "Try…Catch…Finally constructions let you run a set of statements under an environment that retains control if any one of your statements causes an exception". Here is an example of what a try-catch-finally statement would look like:



```
public static void tryExample(){
  int x = 12;
  int y = 0;

  try
  {
    // area that catches the code that might raise an exception
    x = x/y;
  }
  catch (Exception e)
  {
    // area to handle the exception... can take action for the error here
    LogError(e, "integer was divided by 0");
    throw;
  }
  finally
  {
    // finally always runs... area to clean up code
    CloseFiles();
  }
}
```

In this example, the try-catch-finally statement has three blocks. The try block contains the code that could potentially raise an exception. If an exception is caught, then the program jumps to the corresponding catch block. There can also be multiple catch blocks for different exception types, not just one. The catch block handles the exception from the try block. In the catch block, a programmer can take actions such as logging an error. In this code, the LogError() function is used as a placeholder. In this code, the throw expression is used to re-throw the same exception that is handled by the catch block. After the catch block, the finally block is useful for cleanup tasks and always runs regardless of any exception raised.

Iteration and looping statements in C# are all generally recognizable and used in other programming languages. C# utilizes for loops, foreach loops, while loops, and do-while loops. In addition to iteration and looping statements, C# utilizes recognizable jumping statements in its control flow structure. These include break, continue, return, and goto statements. These are some of the examples of how control flow is structured in C#. The selection/ exception-handling statement try-catch-finally is a not as often seen statement in control flow structures. When comparing the C#'s control structure to Java's control structure, there are many similarities that can be found.

C# allows users to utilize a handful of different functions and function modifiers. In C#, users can create regular methods and local functions. A regular method can be declared by using this syntax: <visibility> <return type> <name> (<parameters>). This declaration is common in many other programming languages, such as Java. According to Bill Wagner in Microsoft Learn, Local functions(C# Programming Guide), local functions are methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared using this syntax: <modifiers> <return type> <name> <parameter-list>. Local

functions are declared similar to regular methods, but they are nested within another member and can have modifiers. The modifiers that can be used with local functions are “aysnc”, “unsafe”, “static”, and “extern”.

The “async” modifier is used to specify that the local function is asynchronous. As stated earlier in the case study, asynchronous programming allows the user to write non-blocking code, which is useful when dealing with operations that take more time to complete. Here is a simple example where the “async” modifier would be used on a local function:

```
30      static async Task Main(){
31        // contents...
32
33        int result = await exampleFunction();
34
35        // contents...
36      }
37
38      private static async Task<int> exampleFunction(){
39        string name = "Saul Ulfilas";
40        return name.Length;
41      }
42
```

In this example, the integer result calls exampleFunction() within the Main() method. The Main() method runs synchronously until the first “await” expression. The expression “await” goes hand in hand with the “async” keyword. At this point, the Main() method pauses until the task is complete. Then, exampleFunction() runs asynchronously until it returns. Once it returns, the Main() function can continue to run synchronously. This is a very simple example of “async” and asynchronous programming, and it can be utilized better in more in-depth programs.

The “unsafe” modifier is used to denote a local function as unsafe. This modifier is primarily used when a local function uses pointers. In this case, the programmer will always have to denote the local function as “unsafe”. The “extern” modifier is used to declare a local

function that is implemented externally. For this case, the “extern” modifier can be used only if the local function is static. The “static” modifier is used in a wide number of other languages and is used to declare a local function to belong to the type itself rather than to a specific object.

C# handles parameter passing for its functions and methods in various different ways. Parameters can be passed by using pass by value or pass by reference. In Microsoft Learn, Method Parameters, Bill Wagner says, “For value (struct) types, a copy of the value is passed to the method. For reference (class) types, a copy of the reference is passed to the method”. Since a struct is a value type, the method uses a copy of the struct argument that is passed. On the other hand, class instances are reference types. If a reference type is used for pass by value, a copy of the reference is passed to the method. In order to make a method parameter be pass by reference, the keyword “ref” must be used. Using this keyword, the method receives full access to the actual variable. Here is an example of the keyword “ref” being used in a function header: public void fun1(ref Dog d, int age).

These different details all come together to help create and define the programming language C#. There are details that are unique to C#, and there are some that are taken from other languages. The control flow structure of C# comes with various different statements that allow for different flow between statements. Along with control flow, regular methods and local functions are key in creating an efficient and effective C# program. Understanding parameters and parameter passing is important as well.