# JIT121 Programming Principles Tutorial 2

## Learning Objectives

1. Conditions and repetition
2. Implementing code from algorithms
3. Understanding menu driven design

## Assumptions

You are already familiar with:

* + - * Lecture 1 & 2 and completed Tutorial 1
      * writing code, compiling and running the program code using Visual Studio

## Activities Overview

1. Key Concepts - Conditions
2. Conditions and Loops
3. Simple menu driven system
4. Debugging

### Activity 1: Key concepts

Then open **KeyConcepts\_IF.zip** and run the exe file contained within and complete the three exercises (one per tab on the GUI). Talk to me if you do not understand the correct answers.

### Activity 2: Conditions and Loops

In Visual Studio, copy the following algorithm into a new VS code file - as comments. Then implement the algorithm in C#. Make sure you comply with the C# Coding Style Guide.

**Algorithm:**

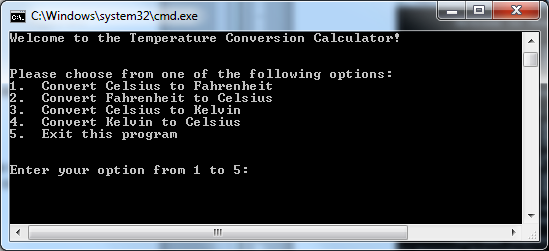
1. Ask the user to enter their name
2. Assign this string to a variable
3. Greet the user with their name
4. If the user's name is the same length as your name, tell them they have the same length name as you. (Hint: use the dot operator on a string to see what methods/properties are available. You should find one called **Length**.

(See: <https://msdn.microsoft.com/library/system.string.length%28v=vs.110%29.aspx>)

1. If the user's name is longer than your name, tell them they have a longer name.
2. If the user's name is shorter than your name, tell them they have a shorter name.
3. Ask the user to enter their favourite colour
4. Assign this **string** to a variable
5. If the colour is the same as your favourite colour, tell the user that you like this colour too
6. Otherwise, tell them what your favourite colour is.
7. Asks the user how many times they would like their name to appear on the screen
8. Assigns this number to a variable
9. If this number is >= 20, asks the user to enter another number smaller than 20 (*Hint: this should be a* **while** *loop!*)
10. If this number is smaller than 20, print the user's name that many times on the screen using a **while** loop.
11. Print an exit message.

Now modify this code to encapsulate steps 4-6 above in a method called **CheckName**, and steps 11-14 in a method called **PrintName**. Now ensure that the **Main** method calls those methods.

**Activity 3: A Simple Menu Driven System**

In this activity, you will develop an algorithm for a simple problem. The algorithm should be written in VS and appear as comments. Once you have developed the algorithm, you can implement a solution in C#. NB: Use the code from the lecture as the basis for your solution. Use ***stubs*** to develop the structure first and ensure that the menu system is working before adding the conversion logic.

The Problem:

Write a program to convert temperatures. The program will display a series of options. The user selects one of these and enters a temperature value. The converted temperature will be displayed on the screen. Then the menu is displayed again.

Here is an example of the menu:

You need to consider what input and output is required for your program. You also need to consider what information your program will need to store (i.e. variables).

An initial high-level algorithm could be:

Welcome message

Loop until user chooses to finish

display list of possible actions (menu)

input user value

perform required conversion

When developing an algorithm to solve a problem, you need to break the problem into a small number of steps. Then each of those steps can be broken down (or refined) into smaller steps and so on until you have simple enough steps that you can easily translate them into code (implement).

If it’s possible to misinterpret an algorithm in any way, then the algorithm is deficient.

Your goal is to express the algorithm in simple steps that correspond to operations in C# so that you can be sure that your strategy is correct before you write any code.

The algorithm must:

* be clear, accurate and concise;
* not to be language-specific, and contain no C# keywords or constructs;
* be understandable to a non-programmer.

Even if you do not yet know how some operation is done in C#, just write the strategy for what you want to happen clearly and precisely. The coding process (implementation) is merely translating the algorithm into C#.

Temperature Conversion Formulae:

Conversion 1 and 2: Fahrenheit and Celsius

The formula for converting Fahrenheit (f) to Celsius (c) is c = 5/9 (f - 32)

Rearrange that formula to convert from Celsius to Fahrenheit (i.e., solve for f)

Conversion 3 and 4: Celsius and Kelvin

The formula for converting Celsius to Kelvin is k = c + 273.15

Rearrange that formula to convert from Kelvin to Celsius (i.e., solve for c)

Code Quality

Ensure that you adhere to the C# Coding Style Guide for this exercise - refer to the key points in the previous activity.

Aim for a modularised solution with the **Main** method resembling a high level algorithm for the entire program, as well as a number of *single-purpose* methods (some void, and some value-returning).

**Activity 4: More VS Debugging**

### Part 1

See **Debugging Workshop Files\_Part 1.zip**.

These activities give you some experience in debugging frequent types of errors. You are given a description of each bug, and asked to fix it yourself. Learning to debug is very much a matter of experience. These bugs are relatively easy to fix, but ask for help if you get stuck.

### NullReferenceException

This activity shows you what it means when you get a **NullReferenceException** in your programs.

1. From the *FrequentErrors\_NullReferenceException* folder, open the solution file **FrequentErrors\_NullReferenceException.sln**.

From Visual Studio’s Solution Explorer window, open the main source file, **FrequentErrors\_NullReferenceException.cs**.

1. Have a quick look at the program code, then press *F5 (Start Debugging)* to run it.

You will get a *NullReferenceException* dialogue box. It should also say *Object reference not set to an instance of an object.*

Click the *Close* button in the top right-hand corner. Stay in the Debugger - don’t stop your program yet.

Hover your mouse over myString. The Debugger will show that myString has a **null** value. 

What does this mean? Remember that a string is a *reference type*, i.e. each **string** object normally points to another part of memory where its data is stored. For example:

name

Barbara Doyle

But in this program, the variable myString is not initialised before displayStringLength() is called. So it is like this:

myString

The Debugger displays this as **null**, but it means uninitialised. Technically, it is sometimes called a **Null Reference**. When the program tries to use **myString.Length**, the system realises that myString is not initialised – so it can’t have a length – and throws a **NullReferenceException**.

By the way, if you happen to know about **null** values in databases, be warned that this is not the same thing. In databases, **null** usually means unknown. In C#, it means uninitialised.

Press **Shift+F5** to stop the Debugger.

1. There are several other ways in which the same problem occurs. Sometimes the cause can be quite complex. Here, we’ll look at a simplified version of one of these.

In the **Main** method, comment-out the call to **displayStringLength()**, and then press **F5** to run the program again. See that a similar exception occurs, and then close the *NullReferenceException* dialogue box.

Hover your mouse over the result variable. In this case, it seems that this variable is being initialised by the following line, but is it?

**string** result = returnsString(**false**);

When you look at the code inside the returnsString() method, you should be able to see why there’s a problem.

You might think that it’s wrong for this method to **return null** – and sometimes that’s true – but it’s fairly common for methods that return **strings** (or other reference types) to return **null** as a way of indicating that they can’t do what they were asked to do.

Press **Shift+F5** to stop the Debugger, and then close Visual Studio.

### Operator Precedence

From the **FrequentErrors\_OperatorPrecedence** folder, open the solution file **FrequentErrors\_OperatorPrecedence.sln**.

From Visual Studio’s Solution Explorer window, open the main source file **FrequentErrors\_OperatorPrecedence.cs**.

The program attempts to calculate the average speed of a car trip (kilometresPerHour) by subtracting the starting distance (startKilometres) from the stopping distance (stopKilometres) and dividing by the time taken (hoursTaken). For the values given in the program, the distance travelled is 800km (= 1800 – 1000), and the time is 10 hours. So the average speed should be 80 km/hour.

Run the program. It outputs a different value. Why? Fix the bug so that the program calculates the correct value. Then run the program to see if your fix is correct. **Hint**: only one line needs to be changed.

### If assignment

From the **FrequentErrors\_IfAssignment** folder, open the solution file **FrequentErrors\_IfAssignment.sln**.

From Visual Studio’s Solution Explorer window, open the main source file **FrequentErrors\_IfAssignment.cs**.

As you can see, the program code declares a Boolean variable named **done** and sets its value to **false**. So, in the following **if** statement, you might expect that the following line will be executed:

Console.WriteLine("done is false");

Run the program. It outputs “done is true”, showing that the **if**-part is being executed rather than the **else**-part. Why?

**Hint 1**: what is the relational operator for testing for equality (in C#)?

**Hint 2**: if you have a green, wavy line (as shown below), hover your mouse over the variable name, done, in this line and read the warning that Visual Studio displays. If you don’t have this green, wavy line, try closing this .cs file and opening it again.



Once you have fixed this bug, please note that there is a much better way to fix it. Rather than saying:



It is clearer and less error-prone to simply say:



Try it and see.

Delete each of the two lines containing the sentence: “When the instructions tell you to, remove this entire line (only).” Delete each of those two lines entirely, but **don’t delete the lines between those two.**

You’ll find that your program no longer compiles. There is an error in the line shown here.



It’s essentially the same bug that you saw above. But because we are using an **int** variable here, the compiler gives you an error message rather than a warning. And because that error message is hard to understand – **Cannot implicitly convert type 'int' to 'bool'** – we’re telling you about it so you’ll recognise it in the future. Hint 1 also applies here, so you should be able to fix this problem easily.

### Infinite Loop

From the **FrequentErrors\_InfiniteLoop** folder, open the solution file **FrequentErrors\_InfiniteLoop.sln**.

From Visual Studio’s Solution Explorer window, open the main source file **FrequentErrors\_InfiniteLoop.cs**.

As you can see from the code, the program prompts the user for a “y” or “n” response. Run the program, and try entering some responses with at least one “y” or “n”. What happens?

To stop the program, the easiest way is to just close the console window. If you run the program via the Visual Studio Debugger **(F5 – Start Debugging)**, then pressing **Ctrl+C** (or **Ctrl+Break**) in the console window will also stop the program. Another way is to click on Visual Studio and then use **Shift+F5 (Stop Debugging)**.

Try to fix the bug. This one is a bit harder than the earlier bugs.

**Hint 1**: the **while**-condition uses the **or** (**||**) logical operator. Remember that this operator is **true** when either of its sides is **true**.

**Hint 2**: think of the user making a particular response (e.g. “y”), and then think about the value of the **while**-condition, for that particular response. Is it what it should be?

**Part 2**

### There is a separate document for this part of the Debugging activity. See Debugging Activity\_Part 2.pdf and follow the instructions.