

START User Manual

Names: Niall Kelly (20461772) and Adam Gray (20364103)

Project Title: START - Easy to Learn Programming Language

Date Finished: 16/2/2023

Contents

- Introduction - Page 1
- Using the Language - Page 1

Introduction

START is a programming language specifically designed to introduce beginner programmers to programming languages. START focuses on reducing abbreviated syntax found in many programming languages, and replacing it with syntax that is more natural to the user in order to make the code more understandable and memorable. START is a helpful stepping stone to help you learn common programming concepts before tackling them in more complex languages.

The following guide should help you get to grips with using START. It includes a step-by-step installation instructions as well as guide to the functionality of START.

Using the Language

Creating your first Program

To create our first program, we'll need to open up either a command line text editor such as Nano, a text editor like NotePad++ or an IDE such as Visual Studio Code, in this tutorial, we will be using Visual Studio Code.

Open Visual Studio code (or your equivalent) and create a new file in the folder where you want to save your work, with the file extensions being `.st`, in this example, our file will be called `hello.st`.

Once you have created the file, open the file and write the following code:

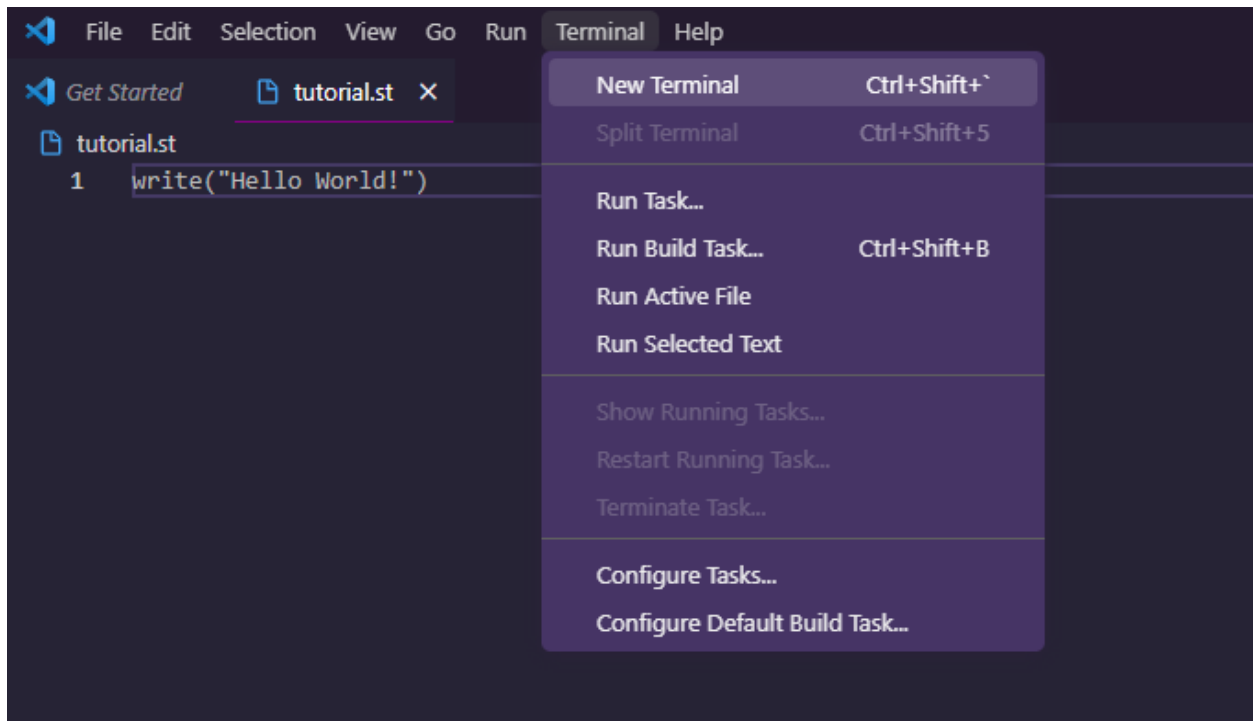
```
write("Hello World!")
```

You have successfully written a Hello World program in START, now lets see how to run it.

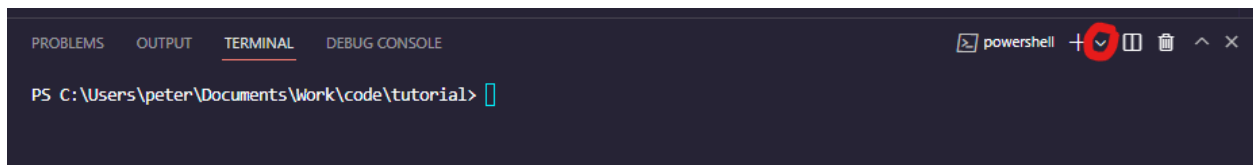
How to run Programs

To run your program there are two paths you can take:

1. if you're using an IDE like VSCode
 2. if you're using a command line text editor or regular text editor like NotePad++
-
1. If using VSCode (or similar), navigate to the top of the software and click on terminal, and then click new terminal:



Once the terminal is open, ensure you are using a PowerShell terminal, if not, simply hit the drop down arrow located beside the terminal type, and select PowerShell from the options:



2. If you are using the alternatives, simply navigate to the Windows search bar and type Windows PowerShell and select the app, and a new PowerShell window will open up. Once inside either Terminal, simply navigate to the correct folders using the `cd` command followed the the folders name you want to move into (to see all available folders to move into, use the `ls` command to list the folders), for more about PowerShell commands see [here](#).

Now that you're in the correct folder, simply run the following command:

```
start1 hello.st
```

You should see “Hello World!” returned to you, congratulations on running your first START program. (**Note:** to run any START program simply run `start1` followed by the name of the file you wish to run).

Below are the features available in START, as well as some simple code to help you to get a grasp on the language, feel free to use it as your starting point for any program you are trying to write.

Variables and Writing

The following code shows simple variable assignment as well as how to use the built in write function:

```
name is "John"
age is 25
height is 6.0
male is true
married is false
children is null

write(name, "is", age, "years old,", "he is", height, "feet tall") nl

write("Is", name, "male:", male) nl

write("His children are:", children)
```

As seen above, variables are assigned using a given name for the variable (e.g. name) followed by the `is` keyword as well as the value of the assignment. The value of a variable can be an Integer, Float, String, Boolean, Null or a List (we will see this one later).

When it comes to outputting to the terminal, we use the write function, this function can take multiple parameters separated by a comma, this automatically separates them via a space in the output. As well as this, we can an optional `nl` to the end of a write statement, this adds a newline to the end of the statement.

Operators and simple operations

The simple mathematical operators in START are as follows:

- add - adds two numbers together (alt is +)
- sub - subtracts the second number from the first (alt is -)
- mult - multiplies two numbers together (alt is *)
- div - divides first number by the second (alt is /)
- pow - puts the first number to the power of the second (alt is ^)
- mod - returns the remainder after division by second number (alt is %)

The following code shows these operators in action

```
a is 10
b is 5

sum is a add b
dif is a sub b
pro is a mult b
quo is a div b

power is a pow b
leftOver is a mod b

write("sum is:", sum, " dif is:", dif, " pro is:", pro, " quo is:", quo) nl

write("power is:", power, " leftOver is:", leftOver)
```

Next we have the comparison operators, these are used to check if a statement evaluates to true or false, there are six of these and they are as follows:

- equals - checks if two numbers are equal (alt is ==)
- not equals - checks if two numbers are not equal (alt is ≠)
- greater than - checks if the first number is larger than the second (alt is >)
- less than - checks if the first number is smaller than the second (alt is <)
- greater than or equal to - checks if the first number is larger or equal to the second (alt is ≥)

- less than or equal to - checks if the first number is smaller or equal to the second (alt is \leq)

Below is some code showing these operators:

```
a is 10
b is 5

write("They are equal:", a equals b) nl

write("They are not equal:", a not equals b) nl

write("a is larger:", a greater than b) nl

write("a is smaller:", a less than b) nl

write("a is less or same:", a less than or equal to b) nl

write("a is more or same:", a greater than or equal to b) nl
```

Finally we have Boolean operators, these are used to check multiple conditions at the same time, there are 3 of these and they are:

- and - checks if both conditions are true
- or - checks if either condition is true
- not - is true if the condition is not true

Below we have some Boolean operators in use:

```
a is 5
b is 10

write(a equals 5 and b equals 10) nl

write(a equals 100 or b equals 10) nl

write(not true)
```

Comments

Comments are a way of writing notes for you and others in your code. Comments are not treated as actual code. We can denote comments as anything within a pair of `//` like so:

```
a is 2 add 4           // addition operation //  
b is 4 mult 6         // multiplication //
```

Lists and List operations

Lists are a way of storing the same as well as different types together in one type. We can create a list by assigning it to a name as shown below:

```
numList is [1,2,3]  
anyList is [1, true, "hello", null, 5.0]
```

Once we have a list, we can use a feature called “List Indexing” to check what value is at what position of a list, remembering that the first position is position 0 not position 1, so to see the first item in `numList` above we would simply do:

```
first is numList[0]  
write(first)
```

We can also reassign what is at a list position by assigning that position in the list to a new value, as shown below:

```
numList[0] is 100
```

If we wanted to remove an item from a list, say the list `[1, 2, 3]` and we wanted to remove 2, we would simply do:

```
list is [1,2,3]  
remove 2 from list
```

If the list had multiple instances of 2, and we wanted to remove them all, we would do the following:

```
remove all 2 from list
```

If we had two lists, and we wanted to combine them into one list, we could use the `concat` operator, which would join the second list onto the end of the first, as shown below:

```
list1 is [1,2,3]
list2 is [4,5]

newList is list1 concat list2
```

If we wanted to add a number into a list, we use the add operator previously seen, but this time, it simply adds the number to list specified, as shown:

```
empty is []
empty is empty add 1
```

If we want to find out how many items are in the list, also known as the length of a list, we would do the following operation:

```
list3 is [100, 200, 300]
len is length of list3
```

Loops

Loops are a way of repeating a certain amount of code multiple times. In START there are two types of loops; for and while.

For loops can be used to repeat code based on the number of elements in a list or string. For example, if we wanted to iterate through a list and write out each element in it, we would write a for loop like so:


```
list is [1, 2, 3, 4, 5]

loop for each num in list{
    write(num) nl
}
```

Each time the loop repeats itself the `num` variable changes to the next item in the list. This means on the first iteration, `num` is 1, on the second iteration `num` is 2 and so on. The loop for each element in the given list. In this example the loop will end after 5 iterations as there are 5 elements in the list.

While loops are used to loop a portion of code while a given condition is true. The loop will end as soon as this condition becomes false. We can achieve the same result as in the for loop example above in a while loop as follows:

```
list is [1, 2, 3, 4, 5]

i is 0
loop while i less than (length of list){
    write(list[i]) nl
    i is i + 1
}
```

In the example, it is important to note that `i` is being increased by one at the end of each loop. This means `i` will be different at the beginning of each loop and will be compared to the length of list. This loop will end once `i` becomes equal to the length of the list. It is important to remember to change the variable you are comparing. If `i` did not change, the given condition would never be false, causing the loop to repeat infinitely.

If-Otherwise Statements

An `if` statement is a way of executing a block of code only if a defined condition is met, so we can say that if something in our code is true, then we can continue into the block

of code.

A simple if statement might look like:

```
a is 5
if a equals 5{
    write("a is 5")
}
```

We would see “a is 5” in our terminal as the condition in the if statement was true, so the block of code inside of the `{ }` was able to execute. But say we wanted to have another check to see if a was 10, what would we do?

Simple, we would use what is known as an `otherwise if` statement. This follows the same rule as the previous if statement, except we say otherwise before the if keyword, as we are checking this case after the original, so say we now wanted to also see if a was 10, we would have:

```
a is 10
if a equals 5{
    write("a is 5")
}
otherwise if a equals 10{
    write("a is 10")
}
```

We can have as many `otherwise if` statements following an if statement as we like. But, now say if neither of these are true, we wanted to just write the value of a, we wouldn't make an otherwise if for every case, that would take too long, instead, we would simply use the otherwise statement, which would encompass all other not defined statements, so now we would have:

```
a is 15
if a equals 5{
    write("a is 5")
}
otherwise if a equals 10{
    write("a is 10")
}
otherwise{
    write(a)
}
```

Summed up, if the if statement or otherwise if statements are met, the block of code connected to them will execute, if none of these are true, our otherwise case will be executed. Also if we had a lone if statement that was not true, such as:

```
b is 10
if b equals 20{
    write("b is 20")
}
```

The statement would not execute and any code after would be executed instead.

Functions and Function Calls

A function is a specific user written block of code that can be executed anytime the user chooses. To do this, the user must first define a function above where it might be called, to do this we use the function keyword, followed by the name of the function, then any parameters the user wants to give to the function. Parameters are numbers that are going to be used when the function is called, an example function to add 1 to any number the function receives is shown below:

```
function addOne(number){
    number is number add 1
    return number
}
```

As we can see at the bottom of the function, we have a return keyword followed by the new number we have created. What this return does is it gives this number back to where the function is called below, so if look at the following code:

```
function addOne(number){
    number is number add 1
    return number
}

newNum is addOne(5)
write(newNum)
```

We can see we are defining a `newNum` which is the return value of the `addOne` function. If we write this number we can see it is 6. This is because we pass the number 5 as a parameter to the function to use.

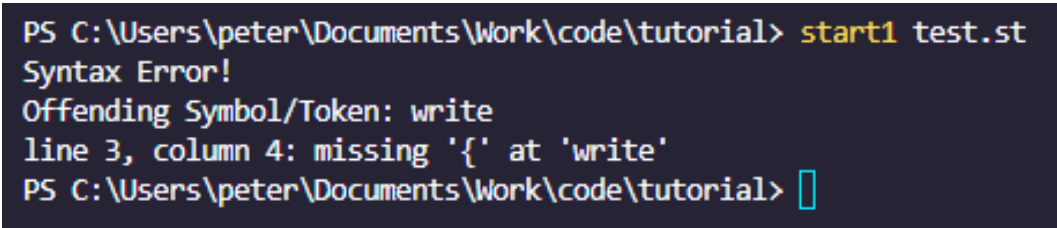
Note: If a function does not have a return statement it will write null as default, remember if you don't need to return, you don't need a function, this helps keep your code more concise.

Reading Error Messages

If you accidentally make a typo or use the wrong symbol, don't worry, it happens to everyone, look at the output in your terminal and read the provided error message, it would give you a good indication of what the problem is. For example, in the following code, I have omitted the opening bracket needed for our if statement:

```
n is 5
if n equals 5
    write("n is 5")
}
```

And below is the outputted error message produced:

A terminal window with a dark background. The prompt is 'PS C:\Users\peter\Documents\Work\code\tutorial>'. The user has entered 'start1 test.st'. The output shows a 'Syntax Error!' followed by 'Offending Symbol/Token: write' and 'line 3, column 4: missing '{' at 'write''. The prompt is then shown again with a cursor.

```
PS C:\Users\peter\Documents\Work\code\tutorial> start1 test.st
Syntax Error!
Offending Symbol/Token: write
line 3, column 4: missing '{' at 'write'
PS C:\Users\peter\Documents\Work\code\tutorial> 
```

As we can see, we are given some information, telling us we have made a syntax error, and at what token (word/variable) that it occurred at. In this case our file errored on `write`. It then tells us that, on line 3 of our code, and at the 4th character, we are missing an opening bracket and instead it found write. So now we know where to find our error and how to correct it to make our code work again.

It is important to read the error messages carefully as they are an important part of the learning process when programming for the first time.