EIFFEL SOFTWARE

Eiffel Loops & Iteration

16 MAY 2018

SUMMARY

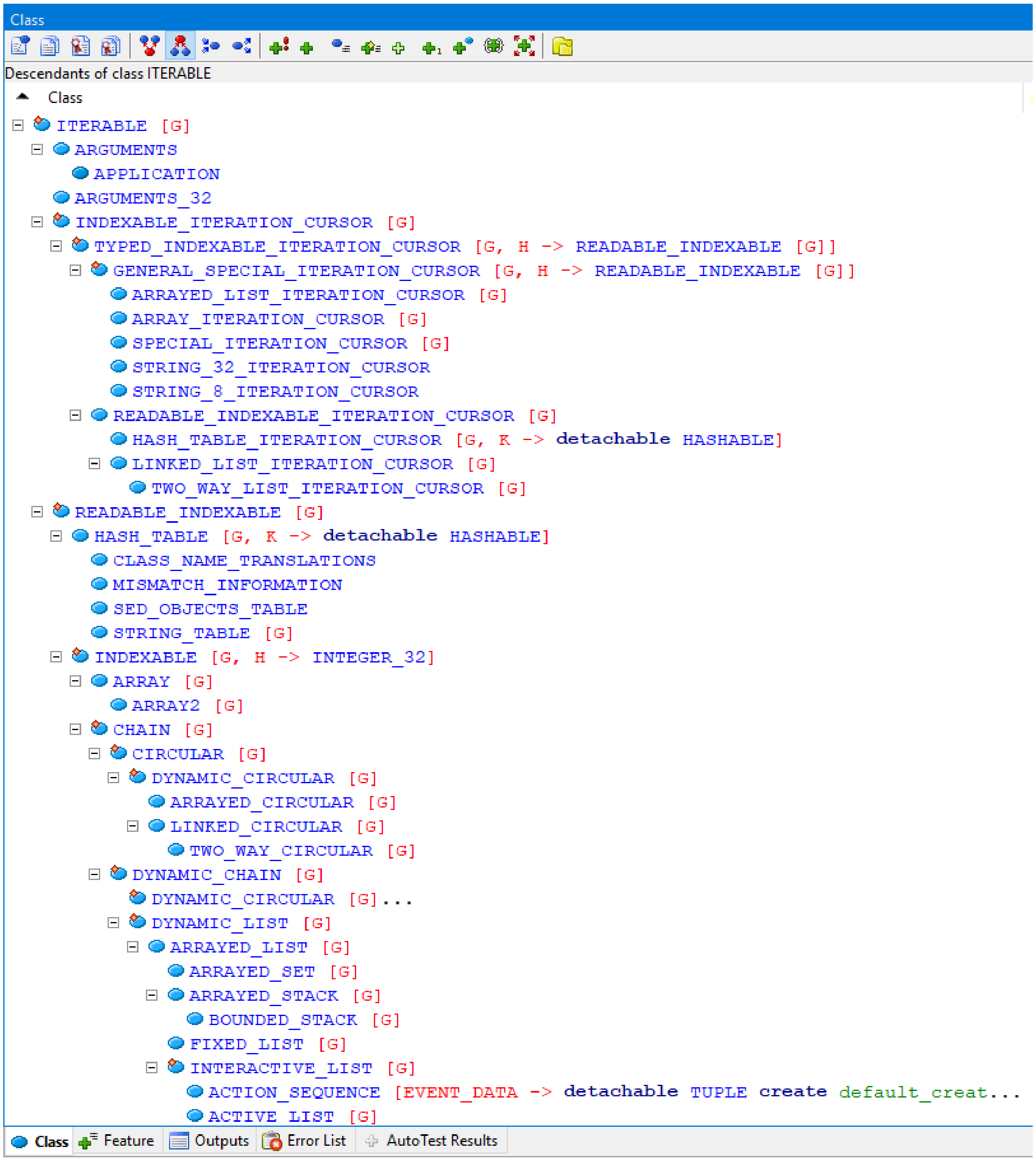
There are two basic looping mechanisms available in Eiffel:

- The **across** loop
- The **from** loop

We will look at various forms of the across loop first and then the from loop afterwards.

GENERALLY ITERABLE THINGS

In Eiffel, many classes (and their objects) are ITERABLE [G]. Using the “Class tool” in EiffelStudio, a look at the Descendants of class ITERABLE [G] is revealing. We can get a sense of just how many things can be iterated over.


Class

Descendants of class ITERABLE

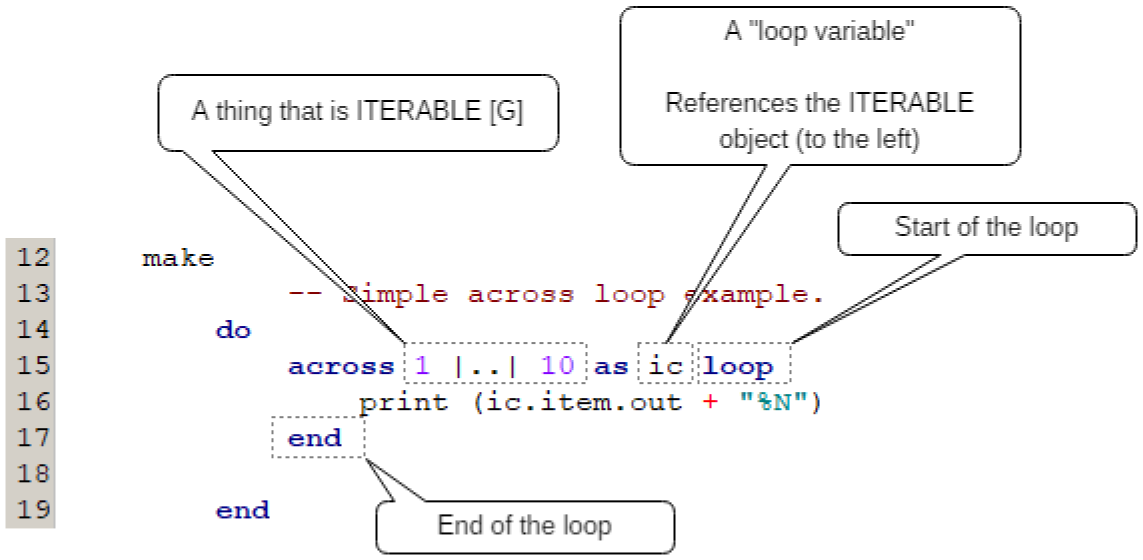
```
Class
⊟ ITERABLE [G]
  ⊟ ARGUMENTS
      APPLICATION
    ARGUMENTS_32
  ⊟ INDEXABLE_ITERATION_CURSOR [G]
    ⊟ TYPED_INDEXABLE_ITERATION_CURSOR [G, H -> READABLE_INDEXABLE [G]]
      ⊟ GENERAL_SPECIAL_ITERATION_CURSOR [G, H -> READABLE_INDEXABLE [G]]
          ARRAYED_LIST_ITERATION_CURSOR [G]
          ARRAY_ITERATION_CURSOR [G]
          SPECIAL_ITERATION_CURSOR [G]
          STRING_32_ITERATION_CURSOR
          STRING_8_ITERATION_CURSOR
      ⊟ READABLE_INDEXABLE_ITERATION_CURSOR [G]
          HASH_TABLE_ITERATION_CURSOR [G, K -> detachable HASHABLE]
        ⊟ LINKED_LIST_ITERATION_CURSOR [G]
            TWO_WAY_LIST_ITERATION_CURSOR [G]
  ⊟ READABLE_INDEXABLE [G]
    ⊟ HASH_TABLE [G, K -> detachable HASHABLE]
        CLASS_NAME_TRANSLATIONS
        MISMATCH_INFORMATION
        SED_OBJECTS_TABLE
        STRING_TABLE [G]
    ⊟ INDEXABLE [G, H -> INTEGER_32]
      ⊟ ARRAY [G]
          ARRAY2 [G]
      ⊟ CHAIN [G]
        ⊟ CIRCULAR [G]
          ⊟ DYNAMIC_CIRCULAR [G]
              ARRAYED_CIRCULAR [G]
            ⊟ LINKED_CIRCULAR [G]
                TWO_WAY_CIRCULAR [G]
        ⊟ DYNAMIC_CHAIN [G]
            DYNAMIC_CIRCULAR [G]...
          ⊟ DYNAMIC_LIST [G]
            ⊟ ARRAYED_LIST [G]
                ARRAYED_SET [G]
              ⊟ ARRAYED_STACK [G]
                  BOUNDED_STACK [G]
                FIXED_LIST [G]
              ⊟ INTERACTIVE_LIST [G]
                  ACTION_SEQUENCE [EVENT_DATA -> detachable TUPLE create default_creat...
                  ACTIVE_LIST [G]
```

NOTE: The [G] in ITERABLE [G] is referred to as a Generic. It represents the type of the objects in the container in the ITERABLE container.

Tables, arrays, cursors, lists, chains, and strings are among the many things we can iterate over. If you want to know if you can iterate over one of your objects, use the Class Tool to see if it inherits from ITERABLE [G].

ACROSS LOOP - BASICS

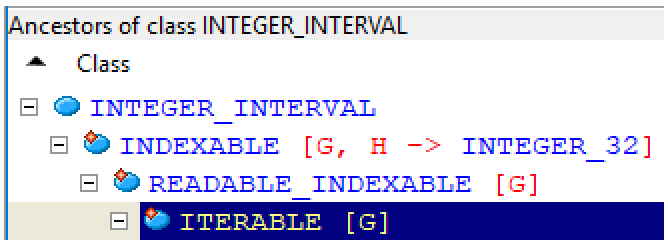
We want to iterate an INTEGER value from 1 to 10 and print the value to the console with each iteration. Refer to lines 15, 16, and 17 (the across loop) of the code below:


A thing that is ITERABLE [G]
A "loop variable"
References the ITERABLE object (to the left)
Start of the loop
End of the loop

```
12    make
13            -- Simple across loop example.
14        do
15            across 1 |..| 10 as ic loop
16                print (ic.item.out + "%N")
17            end
18
19        end
```

Let’s break this down so we can sufficiently understand what the Eiffel compiler “sees” (i.e. learn to “Think like our compiler”).

The **across** loop needs “something” to go “across” – that is – iterate over. The Eiffel compiler sees the **across** keyword and then looks for a “something” that is ITERABLE. In the example above, the Compiler sees the notation `1 |..| 10` as a type of INTEGER_INTERVAL, which is a type of ITERABLE [G] object (thanks to Multiple Inheritance).



Ancestors of class INTEGER_INTERVAL

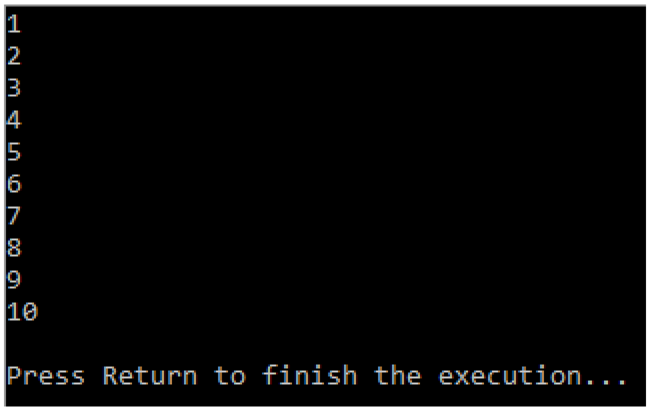
Class

```
⊟ INTEGER_INTERVAL
  ⊟ INDEXABLE [G, H -> INTEGER_32]
    ⊟ READABLE_INDEXABLE [G]
      ⊟ ITERABLE [G]
```

In this case, the cursor object will have ten INTEGER items with values 1 to 10. A reference to the object is held in the loop variable named “ic”.

The **loop** keyword marks the start of the loop cycle and the **end** keyword marks the end. Within the loop, we can reference the current item being iterated by referencing the object.item (e.g. ic.item will be 1,2,3 ... 10 as the loop advances).

The **across** loop code (above) will produce the following console results:



```
1
2
3
4
5
6
7
8
9
10

Press Return to finish the execution...
```

NOTE: With an **across** loop, there is no need to write code to manually advance from item to item. The Eiffel compiler creates code to advance automatically at the end of the loop.

Given the output above, we want to lastly understand the call to "print".

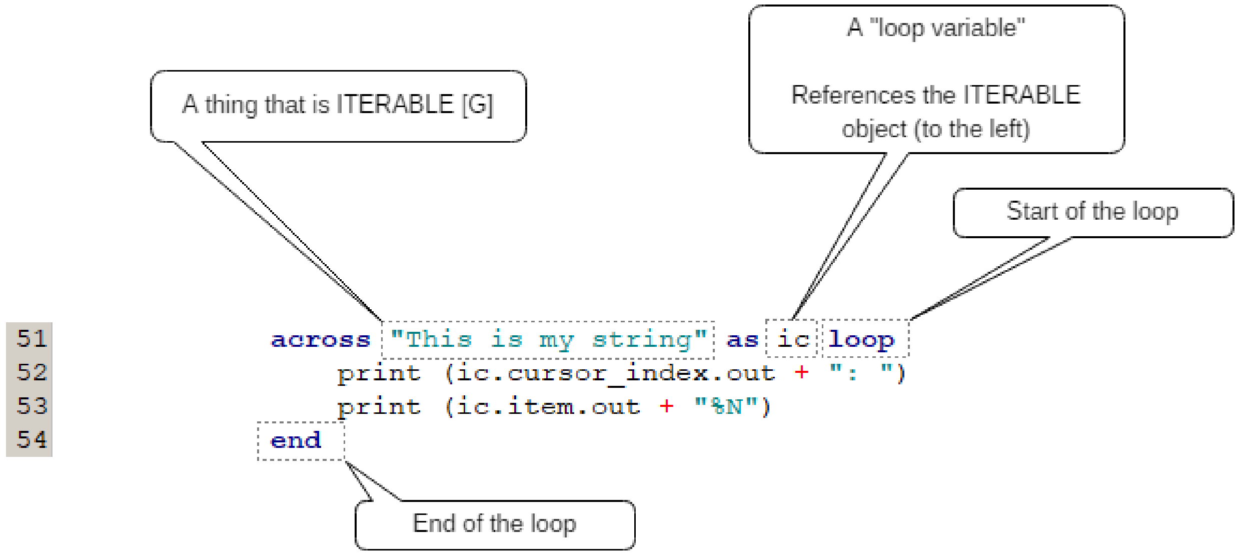
```
print (ic.item.out + "%N")
```

The print feature[1] takes a STRING object and outputs its contents to the console. The code "ic.item" references the current item being iterated in the loop (e.g. INTEGERs 1 to 10). The additional dot-call to "out" transforms (or casts) the INTEGER as a STRING and the `+ "%N"` concatenates a newline character to the end of the STRING.

ACROSS LOOP - INDEXING

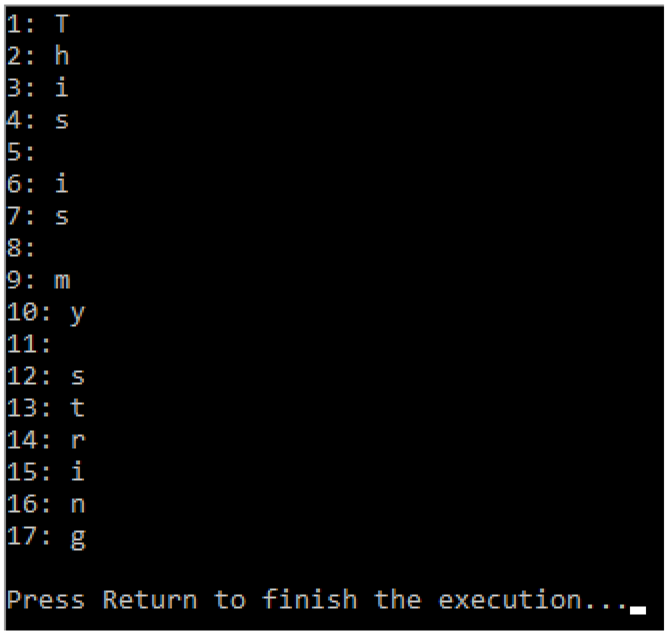
Because Eiffel is iterating over an ITERABLE object, we have access to a number of interesting features of this class as we iterate. One such feature is the "cursor_index" feature. In practice, it looks something like this (line #52):

[1] See the chart for class ANY, specifically the "print" feature.


A thing that is ITERABLE [G]
A "loop variable"
References the ITERABLE object (to the left)
Start of the loop
End of the loop

```
        across "This is my string" as ic loop
            print (ic.cursor_index.out + ": ")
            print (ic.item.out + "%N")
        end
```

In this example, we are iterating the CHARACTERs in the STRING. We want to print not only each CHARACTER, but what position that character holds as an INTEGER in the STRING. The console output will appear like this:



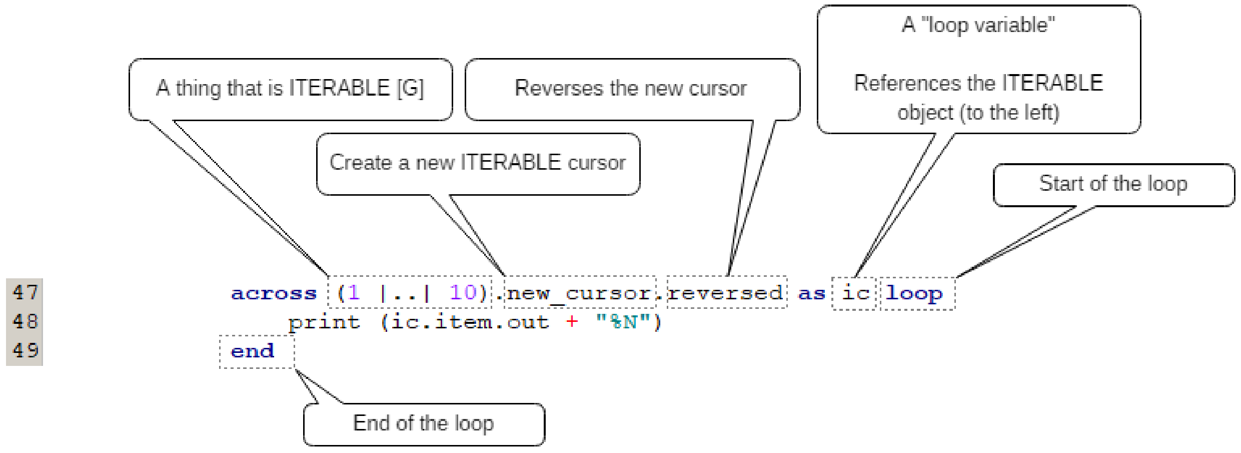
```
1: T
2: h
3: i
4: s
5:
6: i
7: s
8:
9: m
10: y
11:
12: s
13: t
14: r
15: i
16: n
17: g

Press Return to finish the execution...
```

Notice—as the loop iterates each CHARACTER, it is keeping track of an INTEGER index value. We reference this index value with a call to `ic.cursor_index` .

NOTE: The cursor_index feature may not be available on every item container. In the example above, we were able to access the feature because a STRING is a *Client* of INDEXABLE_ITERATION_CURSOR through STRING_8_ITERATION_CURSOR.

ACROSS LOOP - REVERSING

Many ITERABLE objects can be reversed (i.e. iterate them in reverse order). For example: We want to iterate from 10 to 1 instead of 1 to 10. A quick modification to our previous example will show how to do this:


A "loop variable"
References the ITERABLE object (to the left)
A thing that is ITERABLE [G]
Reverses the new cursor
Create a new ITERABLE cursor
Start of the loop
End of the loop

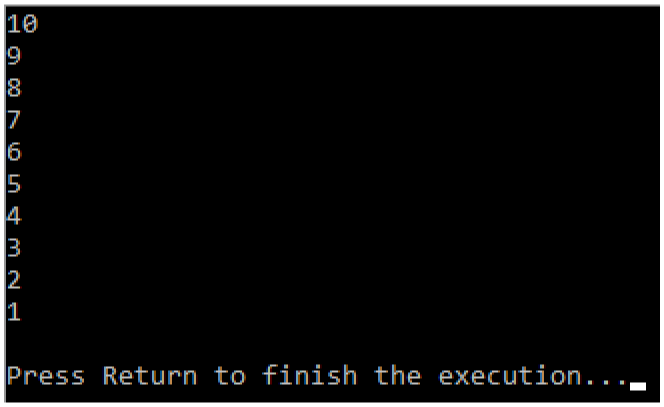
```
across (1 |..| 10).new_cursor.reversed as ic loop
    print (ic.item.out + "%N")
end
```

In this code, we still have the 1 |..| 10 construct. To reverse it, we do the following:

- Enclose the construct in parenthesis. This tells the editor that we are now dealing with the “1 |..| 10” item as a class reference and we can now perform dot-calls with auto-complete.
- Make a call to “.new_cursor” which creates a brand new cursor that we can reverse.
- Make a call to “.reversed” to reverse the order of the items in the resulting “new_cursor”.

That’s it! Our code now traverses the items 1 to 10 in new cursor where the items are 10 to 1 instead.

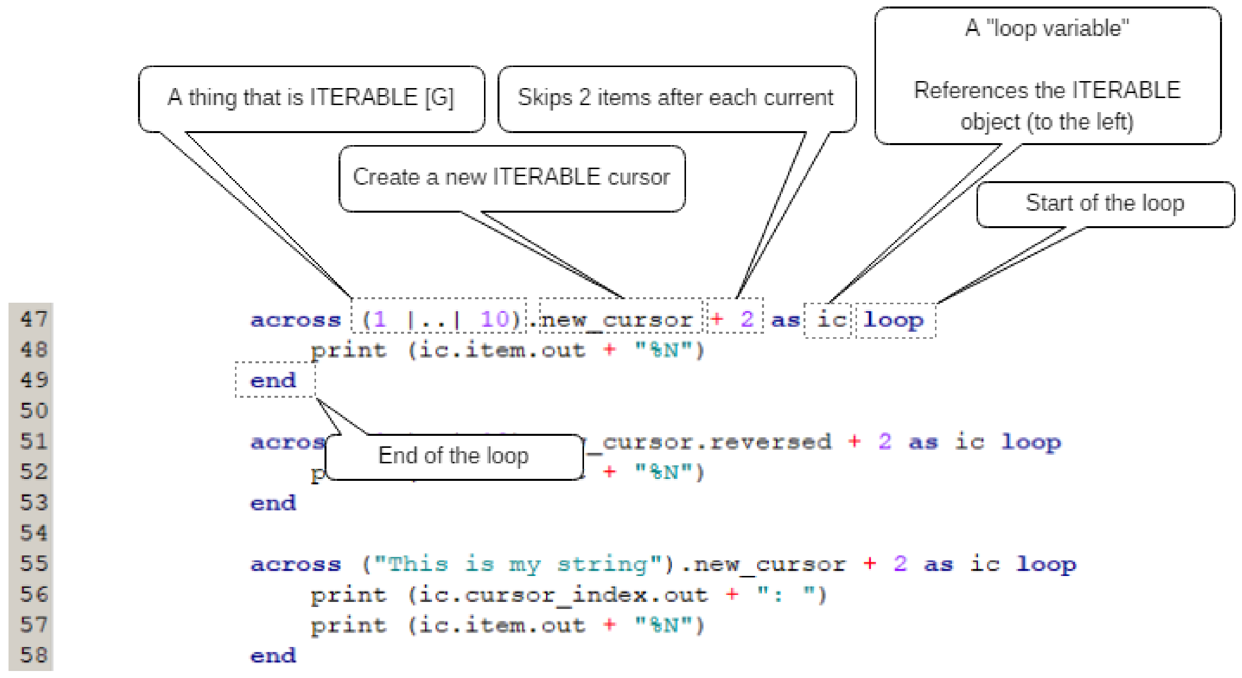
The resulting console output looks as one expects:



```
10
9
8
7
6
5
4
3
2
1

Press Return to finish the execution...
```

ACROSS LOOP - SKIPPING

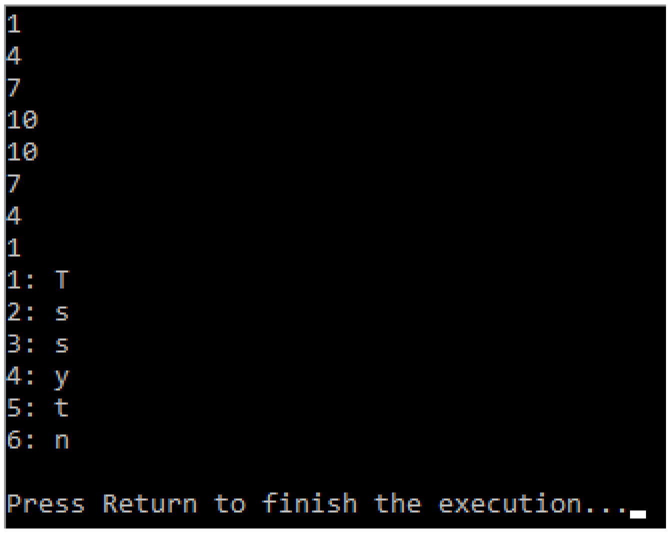
The across loop is simple and elegant. We can iterate forward and in reverse. We can also skip over objects. For example: We might want to print out every 3rd item. To do this, we simple add a “+ value” to our ITERABLE thing, like this:



A thing that is ITERABLE [G]
Skips 2 items after each current
A "loop variable" References the ITERABLE object (to the left)
Create a new ITERABLE cursor
Start of the loop
End of the loop

```
across (1 |..| 10).new_cursor + 2 as ic loop
    print (ic.item.out + "%N")
end

acros[illegible]_cursor.reversed + 2 as ic loop
    p[illegible] + "%N")
end

across ("This is my string").new_cursor + 2 as ic loop
    print (ic.cursor_index.out + ": ")
    print (ic.item.out + "%N")
end
```

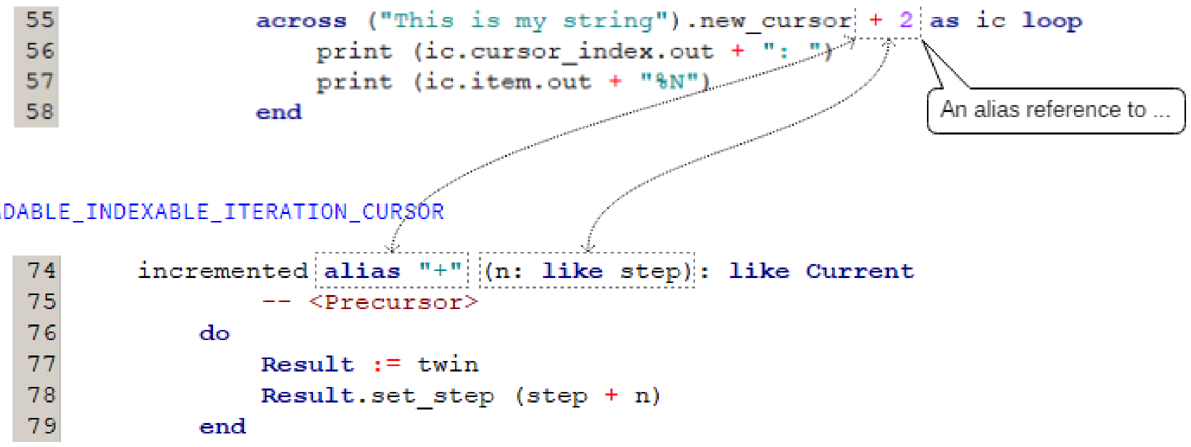
The resulting console output is:



```
1
4
7
10
10
7
4
1
1: T
2: s
3: s
4: y
5: t
6: n

Press Return to finish the execution...
```

Notice—in each across loop (above), we declare the ITERABLE thing (e.g. 1 |..| 10) and then reference a call to “.new_cursor”. The notation of “+ 2” is then applied to the result of new_cursor, causing that ITERABLE thing to start on an item, skip 2, and land on the next item (e.g. 1 .. 4 .. 7 .. 10).

Not only can we “increment” (e.g. “+ n”), we may also “decrement” (e.g. “– n”). In the case of READABLE_INDEXABLE_ITERATION_CURSOR objects, we can use the “+” and “–” notation as an “**alias**” for calls to “incremented” and “decremented”.



```
            across ("This is my string").new_cursor + 2 as ic loop
                print (ic.cursor_index.out + ": ")
                print (ic.item.out + "%N")
            end
```

READABLE_INDEXABLE_ITERATION_CURSOR

```
    incremented alias "+" (n: like step): like Current
            -- <Precursor>
        do
            Result := twin
            Result.set_step (step + n)
        end
```