EIFFEL SOFTWARE

Eiffel Loops & Iteration

16 MAY 2018

SUMMARY

GENERALLY ITERABLE THINGS

ACROSS LOOP - BASICS

ACROSS LOOP - INDEXING

ACROSS LOOP - REVERSING

ACROSS LOOP - SKIPPING

ACROSS LOOP - ARRAY

ACROSS LOOP - HASH TABLE

ACROSS & FROM TOGETHER

SUMMARY

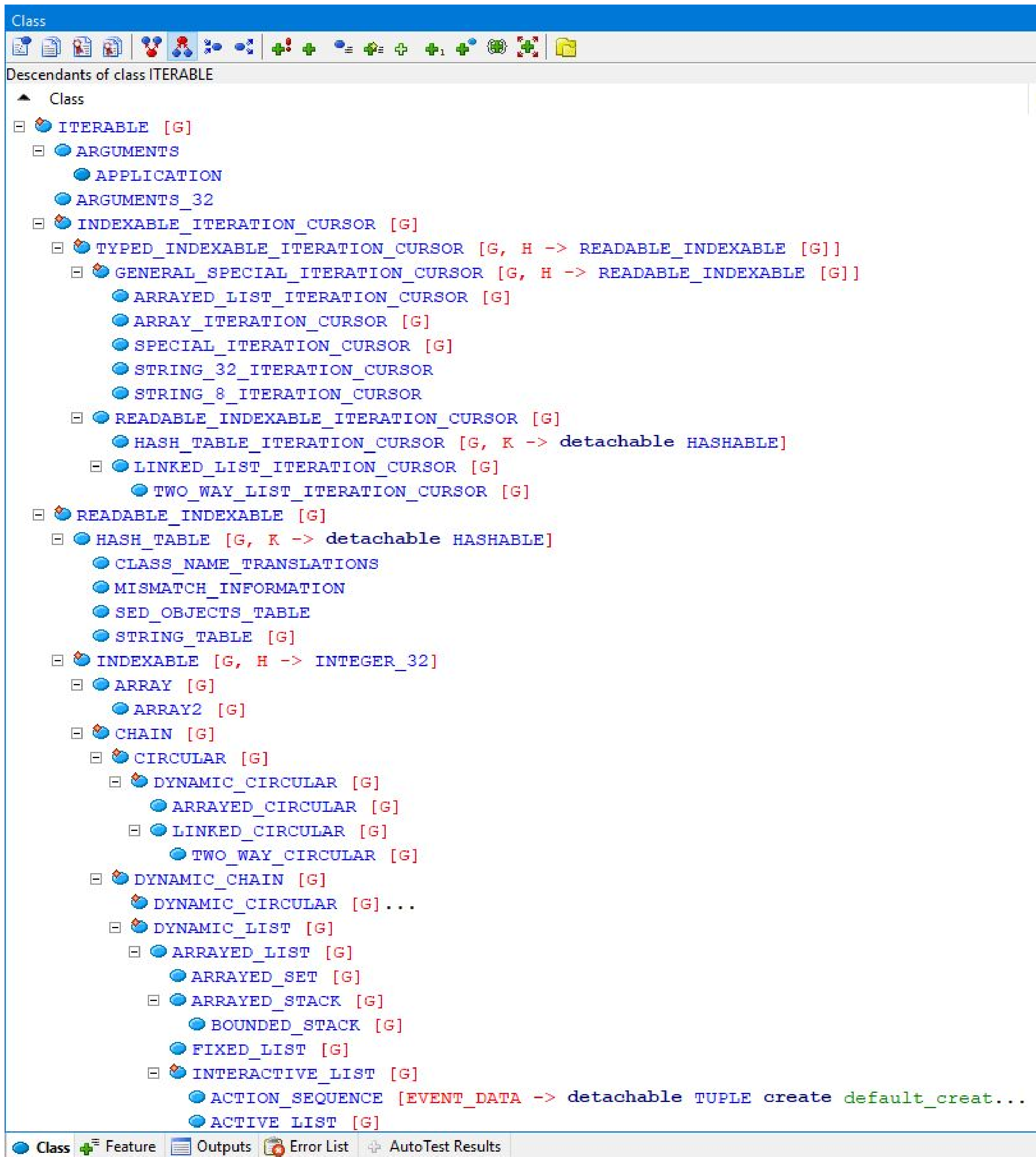
There are two basic looping mechanisms available in Eiffel:

- 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.

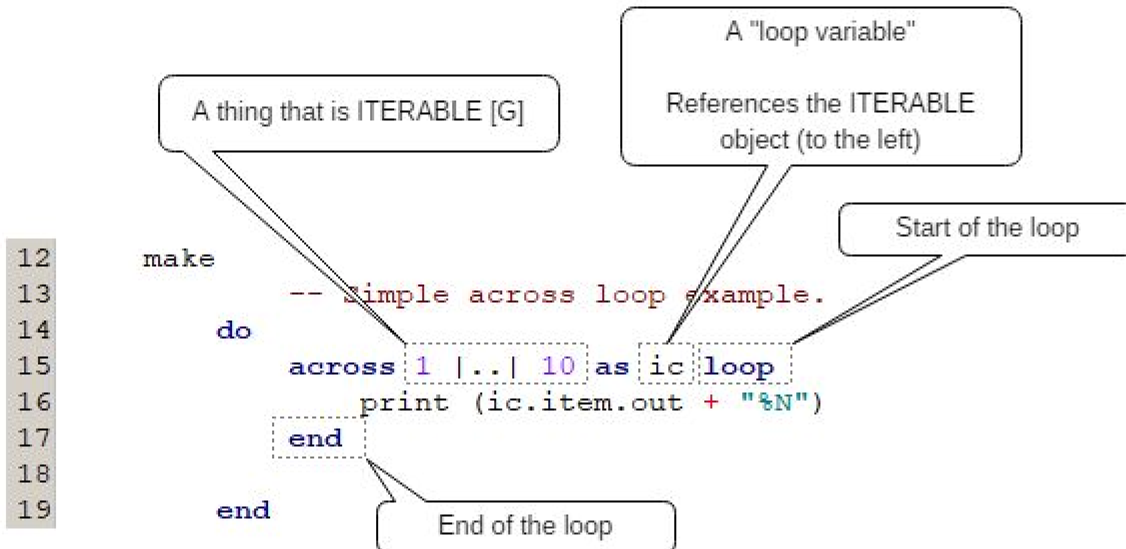


NOTE: The [G] in ITERABLE [G] is referred to as a Generic. It represents the type of the objects in the ITERABLE container.

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].

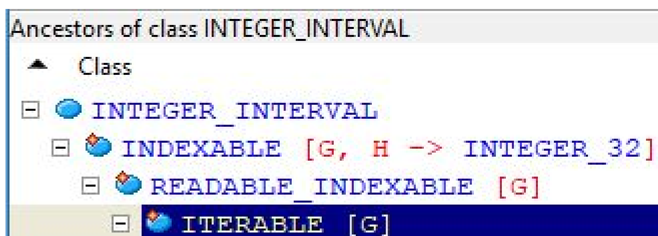
ACROSS LOOP - BASICS

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:



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).



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")
```

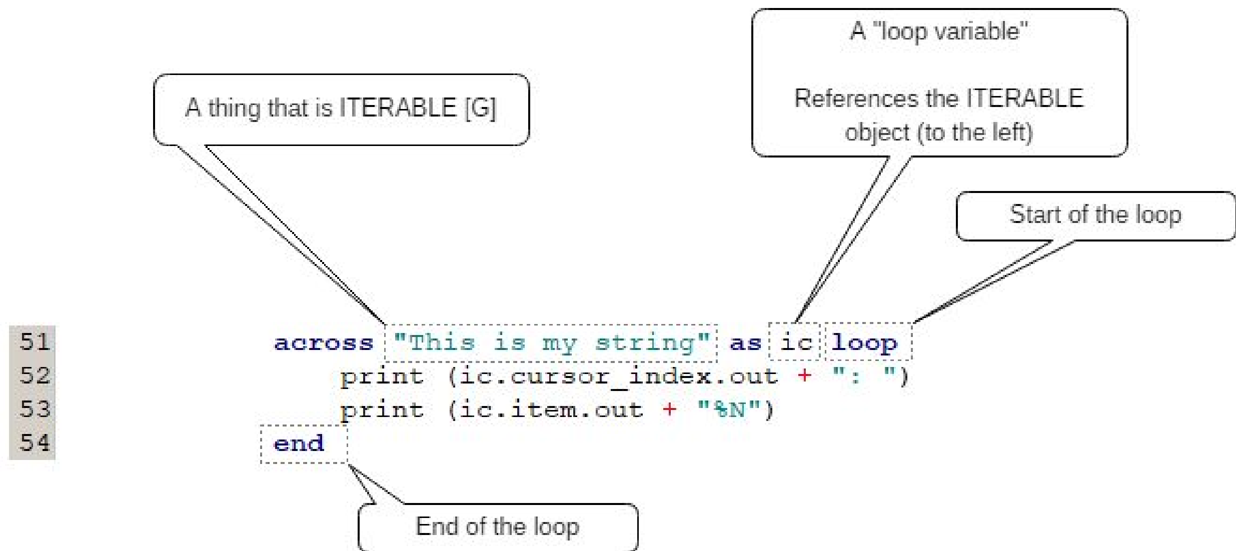
The print feature[1] takes a STRING object and outputs its contents to the console. The code “ic.item” references the current item being iterated in the loop (e.g. INTEGERs 1 to 10). The additional dot-call to “out” transforms (or casts) the INTEGER as a STRING and the `+ "%N"` concatenates a newline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



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:

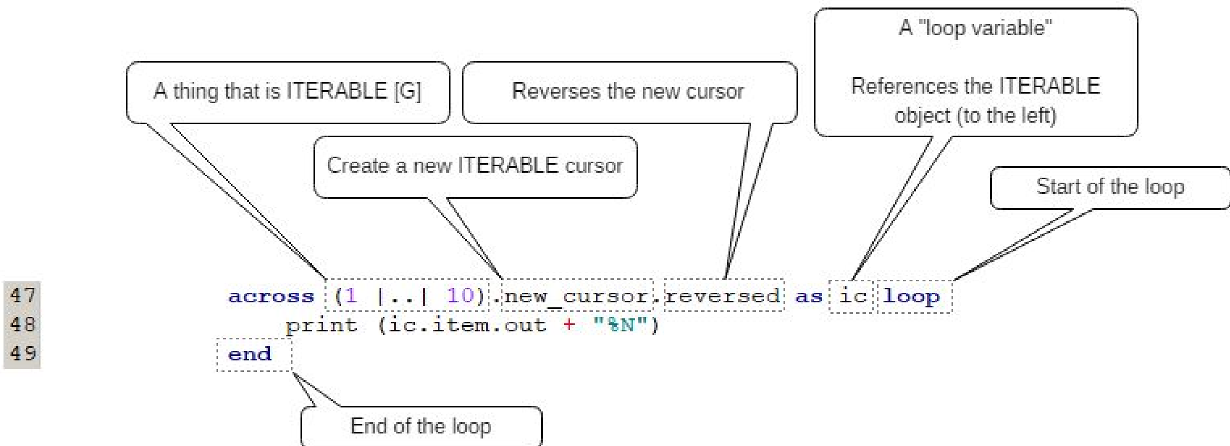
```
1: T
2: h
3: i
4: s
5:
6: i
7: s
8:
9: m
10: y
11:
12: s
13: t
14: r
15: i
16: n
17: g

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:



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

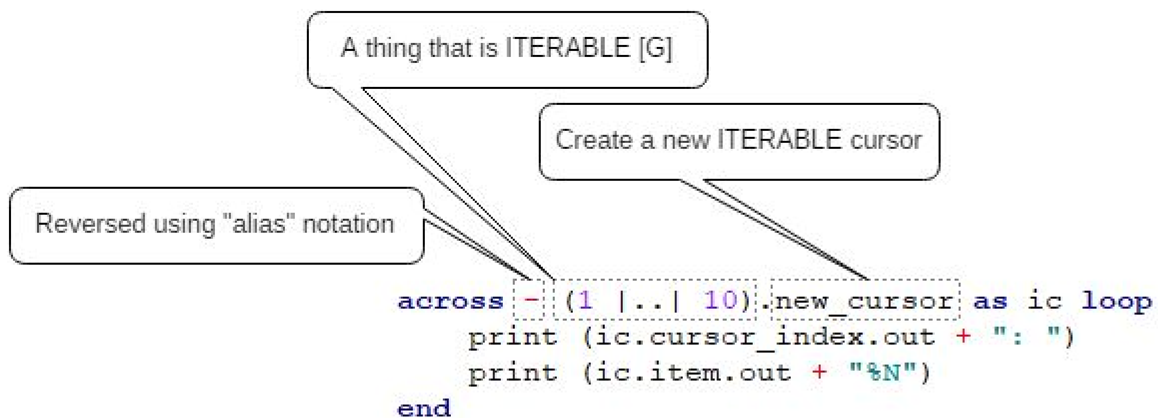
- Enclose the construct in parenthesis. This tells the editor that we are now dealing with the “1 |..| 10” item as a object reference and we can now perform dot-calls with auto-complete.
- Append a call to “.new_cursor” which creates a brand new cursor that we can reverse.
- Append a call to “.reversed”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

We may also reverse the cursor using the alias notion of the “reversed” feature. The code looks like:



This is based on the alias notation in INDEXABLE_ITERATION_CURSOR.reversed

```
reversed alias "-": INDEXABLE_ITERATION_CURSOR [G]
        -- Reversed cursor of the iteration.
```

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

```
47        across (1 |..| 10).new_cursor + 2 as ic loop
48            print (ic.item.out + "%N")
49        end
50
51        acros              _cursor.reversed + 2 as ic loop
52            p              + "%N")
53        end
54
55        across ("This is my string").new_cursor + 2 as ic loop
56            print (ic.cursor_index.out + ": ")
57            print (ic.item.out + "%N")
58        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”.

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

An alias reference to ...

READABLE_INDEXABLE_ITERATION_CURSOR

```
74      incremented alias "+" (n: like step): like Current
75              -- <Precursor>
76          do
77              Result := twin
78              Result.set_step (step + n)
79          end
```

ACROSS LOOP - ARRAY

A typical application of the **across** loop is to apply it to an ARRAY or ARRAYED_LIST. In the example below, we will compute an average score from a list of scores (tests, games, or whatever).

What we iterate "across"

Manifest ARRAY of "scores"

Each score = ic.item

```
make
    note
        goal: "[
            Compute an average score from a list.
            ]"
    local
        l_scores: ARRAY [INTEGER]
        l_sum: INTEGER
        l_average_score: REAL_64
    do
        l_scores := <<3, 4, 5, 7, 9>>
        across l_scores as ic loop
            print (ic.cursor_index.out + ": ")
            print (ic.item.out + "%N")
            l_sum := l_sum + ic.item
        end
        l_average_score := l_sum / l_scores.count
        print ("Sum: " + l_sum.out + "%N")
        print ("Average: " + l_average_score.out + "%N")
    end
```

We first generate a list of individual “scores” from which we compute an average. We use an Eiffel “manifest array” to create an ARRAY [INTEGER] object.

The “l_scores” local object variable is given to the **across** loop to iterate over. Once the loop is complete, we output both the computed sum and average. The output tells the entire story:

```
1: 3
2: 4
3: 5
4: 7
5: 9
Sum: 28
Average: 5.5999999999999996

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

ACROSS LOOP - HASH TABLE

Another common application of the **across** loop is applied to HASH_TABLE [G, K]. The HASH_TABLE stores key-value pairs that we can access easily in an **across** loop.

```
12      make
13          note
14              goal: "[
15                  Generate a list of employees names and numbers.
16                  ]"
17          local
18              l_employees: HASH_TABLE [STRING, INTEGER]
19          do
20              create l_employees.make (3)
21              l_employees.put ("FRANK", 1001)
22              l_employees.put ("FRED", 1002)
23              l_employees.put ("FRITA", 1003)
24              across l_employees as ic loop
25                  print (ic.cursor_index.out + ": ")
                    print (ic.item.out + ", ")
                    print (ic.key.out + "%N")
28              end
29          end
```

Create and load the hash table

What we iterate "across"

The items "key" (i.e. employee number)

The items "value" (i.e. employee name)

The first task is to create and load the HASH_TABLE with values and keys. Once the table is loaded, we can then traverse **across** the table, accessing the keys (i.e. ic.key) and values (ic.item). The console output is as we expect:

```
1: FRANK, 1001
2: FRED, 1002
3: FRITA, 1003

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

ACROSS & FROM TOGETHER

The notation of **across** and **from** can be combined for extra readability. For either the **across** or the **from**, the "loop" construct is defined by the **loop** and **end** keywords. Only the code between **loop** and **end** will be executed for each iteration. The **across** and **from** keywords simply provide the compiler with direction on how to construct the loop in generated C/C++.

In the configuration below, the **from** clause is being used to clearly state preparation of certain variables having to do with the **loop** before the **loop** is executed. This is a "setup phase" in preparation to execute the **loop**.

Also of note are the **invariant** and **variant** clauses.

```
make
    note
        goal: "[
            Compute an average score from a list.
            ]"
    local
        l_scores: ARRAY [INTEGER]
        l_sum, v: INTEGER
        l_average_score: REAL_64
    do
        l_scores := <<3, 4, 5, 7, 9>>
        across
            l_scores as ic
        from
            l_sum := 0
            v := l_scores.count
        invariant
            positive_item: (ic.cursor_index <= ic.last_index) implies ic.item > 0
        loop
            print (ic.cursor_index.out + ": ")
            print (ic.item.out + "%N")
            l_sum := l_sum + ic.item
            v := v - 1
        variant
            v
        end
        l_average_score := l_sum / l_scores.count
        print ("Sum: " + l_sum.out + "%N")
        print ("Average: " + l_average_score.out + "%N")
    end
```

Local variable for "variant"

Sets the variant like a "count-down" value (in this example).

The "from" clause can be used to "prepare" variables associated with the loop before the loop executes. This is for readability.

States the condition that must hold True for each iteration of the loop.

Code to decrement the variant. This can be whatever you want.

Provides a mechanism against "endless loops" (e.g. v cannot go below 0).

The invariant clause (or “loop invariant”) is a ***Design by Contract*** mechanism which provides a set of BOOLEAN predicates stating what must hold **True** before and after each iteration over the **loop**.