

CS 315 HW2

BURAK YETİŞTİREN 21802608 SECTION-1



Table of Contents

<i>Dart</i>	2
1. What are the locations of tests? Pretest, posttest, or both?	2
2. What kind of user-located loop control mechanisms are provided?	3
<i>JavaScript</i>	4
1. What are the locations of tests? Pretest, posttest, or both?	4
2. What kind of user-located loop control mechanisms are provided?	5
<i>Lua</i>	6
1. What are the locations of tests? Pretest, posttest, or both?	6
2. What kind of user-located loop control mechanisms are provided?	7
<i>PHP</i>	8
1. What are the locations of tests? Pretest, posttest, or both?	8
2. What kind of user-located loop control mechanisms are provided?	9
<i>Python</i>	11
1. What are the locations of tests? Pretest, posttest, or both?	11
2. What kind of user-located loop control mechanisms are provided?	11
<i>Ruby</i>	13
1. What are the locations of tests? Pretest, posttest, or both?	13
2. What kind of user-located loop control mechanisms are provided?	14
<i>Rust</i>	15
1. What are the locations of tests? Pretest, posttest, or both?	15
2. What kind of user-located loop control mechanisms are provided?	16
Evaluation of the Languages in Terms of Readability & Writability	18
Readability	18
Writability	19
Learning Strategies That I Have Used	21
Compilers & Interpreters I Have Used	21
References	22

Dart

1. What are the locations of tests? Pretest, posttest, or both?

In the **Dart** language, there are three types of logically controlled loops [1]. They are the while, for, and do-while loops [1]. In **Dart** language the programmers are provided with logically controlled loops, in which the test is done both after and before the loop statements are executed, meaning the locations of the test are both pretest and posttest.

Input:

```
var flag = 1;

while( flag < 1) //PRETEST (this loop is not entered because the value
                //of the flag is '1' which is not less than 1.)
{
    print("In while loop");
}

for(; flag < 1;) //PRETEST (this loop is not entered because the value
                //of the flag is '1' which is not less than 1.)
    print("In for loop");

do
{
    print("In do-while loop");
}
while(flag < 1); //POSTTEST (this loop is entered ONLY ONCE because the value
                //of the flag is checked after the statements in loop are executed.)
```

Output:

```
Console

In do-while loop
```

In the example program segment provided above, the logically controlled loops are tested in terms of the location of the test. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. This is done because, every loop has the same logical statement as a test, here it checks if the variable flag is less than one. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the do-while loop is entered, which is a posttest loop in the **Dart** language. This means that the mechanisms for both the pretest and posttest are provided in the **Dart** language, and they are working properly.

2. What kind of user-located loop control mechanisms are provided?

In the **Dart** language, there are two user-located loop control mechanisms [2]. These are '*continue*', and '*break*' statements [2]. The '*continue*' statement is used under the scope of an '*if*' statement. If the '*if*' statement is entered, then the '*continue*' statement is executed. When it is executed it skips the current step of the loop, and the next step is started being executed, if possible. The '*break*' statement is used under the scope of an '*if*' statement, similar to the '*continue*' statement. This time, if the '*if*' statement is entered, then the '*break*' statement is executed, and when it is executed it terminates the loop. Additional to these statements, **Dart** language also provides the labels for the programmers to relate the user-controlled loop mechanisms to the loop they would like [3]. The labels are optional to use. The example below should further demonstrate this phenomenon.

Input:

```
outerloop: //label of the outer for loop
for(var i = 0; i < 3; i++)
{
  innerloop: //label of the inner for loop
  for(var j = 0; j < 3; j++)
  {
    if( j == 1)
    {
      continue innerloop; //If the value of j variable, is equal to '1', 'continue' statement skips
                          //to the beginning of the inner loop.
    }

    if( i == 2)
    {
      break outerloop; //If the value of i, is equal to '2', 'break' statement terminates
                      //the outer loop.
    }
    print("i : $i j: $j");
  }
}
```

Output:

```
Console

i : 0 j: 0
i : 0 j: 2
i : 1 j: 0
i : 1 j: 2
```

In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested for loops. The outer and inner loops are for loops, which basically count from 0 to 2. The '*continue*' statement is put in an '*if*' statement, which would be executed if the variable '*j*' is equal to 1. Similarly, the '*break*' statement is put in an '*if*' statement, which would be executed if the variable '*i*' is equal to 2. If we take a look at the results of this code segment, we can see indeed, that the '*continue*' and '*break*' statements are working, because indeed the step of the loop was skipped when '*j*' was equal to 1, that the value 1 for was not

printed for 'j', and the last value printed for 'i' was 1, meaning that the outer loop was terminated when the value of 'i' was equal to 2, before that value could be printed.

JavaScript

1. What are the locations of tests? Pretest, posttest, or both?

In **JavaScript** language, there are three types of logically controlled loops [4]. They are the while, for, and do-while loops [4]. In **JavaScript** language the programmers are provided with logically controlled loops, in which the test is done both after and before the loop statements are executed, meaning the locations of the test are both pretest and posttest.

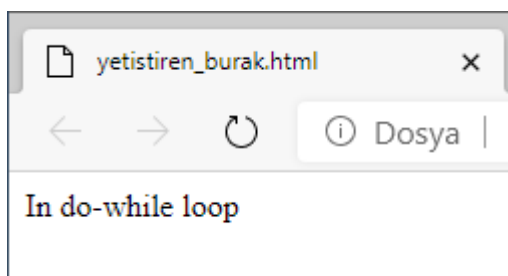
Input:

```
var flag = 1;
while( flag < 1) //PRETEST (this loop is not entered because the value
                //of the flag is '1' which is not less than 1.)
{
    document.write("In while loop" + "<br>");
}

for(; flag < 1;) //PRETEST (this loop is not entered because the value
                //of the flag is '1' which is not less than 1.)
    document.write("In for loop" + "<br>");

do
{
    document.write("In do-while loop" + "<br>");
}
while(flag < 1); //POSTTEST (this loop is entered ONLY ONCE because the value
                //of the flag is checked after the statements in loop are executed.)
```

Output:



In the example program segment provided above, the logically controlled loops are tested in terms of the location of the test. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. This is done because, every loop has the same logical statement as a test, here it checks if the variable flag is less than one. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the do-while loop is entered, which is a posttest loop in the **JavaScript** language. This means that the mechanisms for both the pretest and posttest are provided in the **JavaScript** language, and they are working properly.

2. What kind of user-located loop control mechanisms are provided?

In **JavaScript** language, there are two user-located loop control mechanisms [5]. These are '*continue*', and '*break*' statements [5]. The '*continue*' statement is used under the scope of an '*if*' statement. If the '*if*' statement is entered, then the '*continue*' statement is executed. When it is executed it skips the current step of the loop, and the next step is started being executed, if possible. The '*break*' statement is used under the scope of an '*if*' statement, similar to the '*continue*' statement. This time, if the '*if*' statement is entered, then the '*break*' statement is executed, and when it is executed it terminates the loop. Additional to these statements, **JavaScript** language also provides the labels for the programmers to relate the user-controlled loop mechanisms to the loop they would like [6]. The labels are optional to use. The example below should further demonstrate this phenomenon.

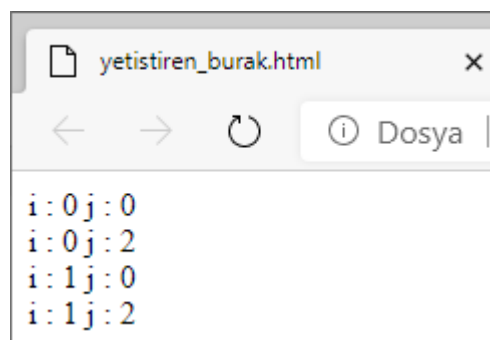
Input:

```
outerloop: //label of the outer for loop
for(var i = 0; i < 3; i++)
{
    innerloop: //label of the inner for loop
    for( var j = 0; j < 3; j++)
    {
        if( j == 1)
        {
            continue innerloop; //If the value of j variable, is equal to '1', 'continue' statement skips
                                //to the beginning of the inner loop.
        }

        if( i == 2)
        {
            break outerloop; //If the value of i, is equal to '2', 'break' statement terminates
                             //the outer loop.
        }

        document.write("i : " + i + " j : " + j + "<br>");
    }
}
```

Output:



```
i:0j:0
i:0j:2
i:1j:0
i:1j:2
```

In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested for loops. The outer and inner loops are for loops, which basically count from 0 to 2. The *'continue'* statement is put in an *'if'* statement, which would be executed if the variable *'j'* is equal to 1. Similarly, the *'break'* statement is put in an *'if'* statement, which would be executed if the variable *'i'* is equal to 2. If we take a look at the results of this code segment, we can see indeed, that the *'continue'* and *'break'* statements are working, because indeed the step of the loop was skipped when *'j'* was equal to 1, that the value 1 for was not printed for *'j'*, and the last value printed for *'i'* was 1, meaning that the outer loop was terminated when the value of *'i'* was equal to 2, before that value could be printed.

Lua

1. What are the locations of tests? Pretest, posttest, or both?

In the **Lua** language, there are two types of logically controlled loops [7]. Another type of loop, which is a *'for'* loop, is counter-controlled [8]. The logically controlled loops are the while and repeat-until loops. In the **Lua** language, the programmers are provided with logically controlled loops, in which the test is done both after and before the loop statements are executed, meaning the locations of the test are both pretest and posttest.

Input:

```
flag = 1;

while(flag < 1) --PRETEST (this loop is not entered because the value
                --of the flag is '1' which is not less than 1.)
do
    print("In while loop");
end

repeat
    print("In repeat-until loop")
    flag = flag + 1
until( flag > 1 ) --POSTTEST (this loop is entered ONLY ONCE because the value
                  --of the flag is checked after the statements in loop are executed, and in the
                  --loop the flag is incremented because this loop continues to execute while
                  --the boolean expression is FALSE.)
```

Output:

Result

CPU Time: 0.00 sec(s), Memory:

```
In repeat-until loop
```

In the example program segment provided above, the logically controlled loops are tested in terms of the location of the test. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially, unless in the repeat-until loop. The repeat-until loop is a special type of loop which keeps on executing, while the Boolean expression provided is false. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the repeat-until loop is entered, which is a posttest loop in the **Lua** language. This means that the mechanisms for both the pretest and posttest are provided in the **Lua** language, and they are working properly.

2. What kind of user-located loop control mechanisms are provided?

In the **Lua** language, there is only one user-located loop control mechanism, along with the labeling mechanism [9] [10]. The '*break*' statement is used under the scope of an '*if*' statement. If the '*if*' statement is entered, then the '*break*' statement is executed. When it is executed it terminates the loop. Additional to this statement, the **Lua** language also provides the labels for the programmers to relate the user-controlled loop mechanisms to the loop they would like [10]. The labels are optional to use. But if the programmer wants to simulate the effects of a '*continue*' statement, which is available in some languages, which allows the programmer to skip an iteration of a loop, s/he can use a label with a '*goto*'. The example below should further demonstrate this phenomenon.

Input:

```
i = 0;
while (i < 3)
do
    j = 0
    if i == 2
    then
        break --If the value of i, is equal to '2', 'break' statement terminates the loop
    end
    while(j < 3)
    do
        if j == 1 then
            goto continue
        end
        print("i : ", i, " j: ", j);
        ::continue:: --label to skip to the next iteration
        j = j + 1
    end
    i = i + 1
end
```

Output:

Result

CPU Time: 0.00 sec(s), Memory: 26;

i :	0	j :	0
i :	0	j :	2
i :	1	j :	0
i :	1	j :	2

In this code segment, the behavior of the `'goto'`, and `'break'` statements are demonstrated. There are two nested while loops. The outer and inner loops are while loops, which basically count from 0 to 2. The `'goto'`, and `'break'` statements are put into `'if'` statements, and if the logic expression of the `'if'` statements are satisfied, they are executed. In this code segment, the `'goto'` statement simulates the effects of the `'continue'` statement. As it can be seen in the output, `j = 1` is skipped each time, while counting from 0 to 2 for `j`, and the loop is terminated when `i = 2`.

PHP

1. What are the locations of tests? Pretest, posttest, or both?

In the **PHP** language, there are three types of logically controlled loops [11]. They are the while, for, and do-while loops [11]. In **PHP** language the programmers are provided with logically controlled loops, in which the test is done both after and before the loop statements are executed, meaning the locations of the test are both pretest and posttest.

Input:


```
$flag = 1;

while( $flag < 1) //PRETEST (this loop is not entered because the value
                  //of the flag is '1' which is not less than 1.)
{
    echo "In while loop <br>";
}

for(; $flag < 1;) //PRETEST (this loop is not entered because the value
                  //of the flag is '1' which is not less than 1.)
    echo "In for loop <br>";

do
{
    echo "In do-while loop <br>";
}
while($flag < 1); //POSTTEST (this loop is entered ONLY ONCE because the value
                  //of the flag is checked after the statements in loop are executed.)
```

Output:

 <https://SafePri>

In do-while loop

In the example program segment provided above, the logically controlled loops are tested in terms of the location of the test. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. This is done because, every loop has the same logical statement as a test, here it checks if the variable flag is less than one. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the do-while loop is entered, which is a posttest loop in the **PHP** language. This means that the mechanisms for both the pretest and posttest are provided in the **PHP** language, and they are working properly.

2. What kind of user-located loop control mechanisms are provided?


In the **PHP** language, there are three user-located loop control mechanisms [12] [13]. These are '*continue*', '*break*', and '*goto*' statements [13] [14] [15]. The '*continue*' statement is used under the scope of an '*if*' statement. If the '*if*' statement is entered, then the '*continue*' statement is executed. When it is executed it skips the current step of the loop, and the next step is started being executed, if possible. The '*break*' statement is used under the scope of an '*if*' statement, similar to the '*continue*' statement. This time, if the '*if*' statement is entered, then the '*break*' statement is executed, and when it is executed it terminates the loop. The '*goto*' statement is used under the scope of an '*if*' statement, similar to the previously explained statements. But this time, there is a difference, and it is, the '*goto*' statement in **PHP** must be used with labels [13]. These will be demonstrated in the two code segments below.

Input:

```
for($i = 0; $i < 3; $i++)
{
    for($j = 0; $j < 3; $j++)
    {
        if( $j == 1)
        {
            continue 1; //If the value of j variable, is equal to '1', 'continue' statement skips
                        //to the beginning of the inner loop, the '1' after the continue statement means
                        //that only one layer is skipped.
        }

        if( $i == 2)
        {
            break 2; //If the value of i, is equal to '2', 'break' statement terminates
                    //the outer loop, the '2' after the break statement means
                    //that two layers are terminated.
        }
        echo "i : $i j: $j <br>";
    }
}
```

Output:




```
i : 0 j: 0
i : 0 j: 2
i : 1 j: 0
i : 1 j: 2
```

In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested for loops. The outer and inner loops are for loops, which basically count from 0 to 2. The *'continue'* statement is put in an *'if'* statement, which would be executed if the variable *'j'* is equal to 1. After this *'continue'* statement, there is *'1'* which means the statement will affect only one layer of loops, that is the inner for loop. In this example *'1'* was unnecessary, because by default the *'continue'* statement, without an integer after it affects only one loop, but to demonstrate this functionality I have chosen to write *'1'* after the *'continue'* statement. Similarly, the *'break'* statement is put in an *'if'* statement, which would be executed if the variable *'i'* is equal to 2. Similar to the *'continue'* statement, the number after the *'break'* statement determines how many layers the statement is going to affect. If we take a look at the results of this code segment, we can see indeed, that the *'continue'* and *'break'* statements are working, because indeed the step of the loop was skipped when *'j'* was equal to 1, that the value 1 for was not printed for *'j'*, and the last value printed for *'i'* was 1, meaning that the outer loop was terminated when the value of *'i'* was equal to 2, before that value could be printed.

Input:

```
for($count = 0; $count < 5; $count++)
{
    if($count == 1)
    {
        goto skip;
    }
    if($count == 4)
    {
        goto terminate;
    }
    echo "count: $count <br>";
    skip: //label to skip to the next iteration
}
terminate: //label to indicate the end of the loop tp terminate
```

Output:

 <https://>

```
count: 0
count: 2
count: 3
```

In this code segment, the behavior of the *'goto'* statement is demonstrated. The statements are put into *'if'* statements, and if the logic expression of the *'if'* statements are satisfied, they are executed. In this code segment, the *'goto'* statements simulate the effects of the *'continue'*, and *'break'* statements. As it can be seen in the output, 1 is skipped while counting from 0 to 4, and the loop is terminated when the count is 4.

Python

1. What are the locations of tests? Pretest, posttest, or both?

In **Python** language, there is one type of logically controlled loop, and it is the while loop [1]. The other loops are for loops, and they are categorized as counter-controlled loops. In the while loop in **Python** language test is done before the loop statements are executed, meaning the location of the test is pretest. There are no loops in **Python** for posttest loop control, but it can be simulated. This can be seen in the following code segment.

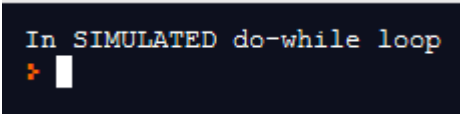
Input:

```
flag = 1

while flag < 1: #PRETEST (this loop is not entered because the value
                #of the flag is '1' which is not less than 1.)
    print("In while loop")

while True: #simulating do-while (posttest) (this loop is entered ONLY ONCE because
            #the value of the flag is checked after the statements in loop are executed.)
    print("In SIMULATED do-while loop")
    if(flag <= 1):
        break
```

Output:



```
In SIMULATED do-while loop
>
```

In the example program segment provided above, the logically controlled while loop and the simulated do-while loop are tested. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. The simulated do-while loop is provided to be able to see the effects of the while loop. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the simulated do-while loop is entered, which is a simulated posttest loop. **A POSTTEST LOOP DOES NOT EXIST IN THE PYTHON LANGUAGE.** The only mechanism for pretest is provided in the **Python** language, which is working properly.

2. What kind of user-located loop control mechanisms are provided?

In **Python** language, there are two user-located loop control mechanisms [16]. These are 'continue', and 'break' statements [16]. The 'continue' statement is used under the scope of an 'if' statement. If the 'if' statement is entered, then the 'continue' statement is executed. When it is executed it skips the current step of the loop, and the next step is started being executed, if

possible. The *'break'* statement is used under the scope of an *'if'* statement, similar to the *'continue'* statement. This time, if the *'if'* statement is entered, then the *'break'* statement is executed, and when it is executed it terminates the loop. The *'break'* and *'continue'* statements can be used in the counter-controlled loops in **Python** too. In **Python** language, there are no labels provided, which the programmers can use to jump around in the program.

Input:

```
i = 0
while i < 3:
    j = 0
    if i == 2:
        break #If the value of i, is equal to '2', 'break' statement terminates
              #the outer loop.
    while j < 3:
        if j == 1:
            j = j + 1
            continue #If the value of j variable, is equal to '1', 'continue' statement skips
                    #to the beginning of the inner loop.
        print("i: ", i, " j: ", j)
        j = j + 1
    i = i + 1
```

Output:

```
i: 0 j: 0
i: 0 j: 2
i: 1 j: 0
i: 1 j: 2
✖
```

In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested while loops. The outer and inner loops are while loops, which basically count from 0 to 2. The *'continue'* statement is put in an *'if'* statement, which would be executed if the variable *'j'* is equal to 1. Similarly, the *'break'* statement is put in an *'if'* statement, which would be executed if the variable *'i'* is equal to 2. If we take a look at the results of this code segment, we can see indeed, that the *'continue'* and *'break'* statements are working, because indeed the step of the loop was skipped when *'j'* was equal to 1, that the value 1 for was not printed for *'j'*, and the last value printed for *'i'* was 1, meaning that the outer loop was terminated when the value of *'i'* was equal to 2, before that value could be printed.

Ruby

1. What are the locations of tests? Pretest, posttest, or both?

In the **Ruby** language, there are four types of logically controlled loops. They are the pretest 'while', and 'until' loops, with their posttest variants [17]. As explained, in **Ruby** language the programmers are provided with logically controlled loops, in which the test is done both after and before the loop statements are executed, meaning the locations of the test are both pretest and posttest.

Input:

```
$flag = 1

while $flag < 1 do #PRETEST (this loop is not entered because the value
                  #of the flag is '1' which is not less than 1.)
  puts "In while loop"
end

begin
  puts("In posttest while loop" )
end while $flag < 1 #POSTTEST (this loop is entered ONLY ONCE because the value
                  #of the flag is checked after the statements in loop are executed.)

until $flag > 1 do #PRETEST (this loop is entered because the value
                  #of the flag is '1' which is not less than 1 THIS LOOP ITERATES AS LONG AS
                  #THE BOOLEAN EXPRESSION IS FALSE.)
  puts("In until loop" )
  $flag +=1;
end

$flag = 1 #We made sure that the value of $flag is 1
          #because the until loop is entered

begin
  puts("In posttest until loop" )
  $flag +=1;
end until $flag > 1 #POSTTEST (this loop is entered ONLY ONCE because the value
                  #of the flag is incremented in the loop and the value is checked after the loop statements
                  #THIS LOOP ITERATES AS LONG AS THE BOOLEAN EXPRESSION IS FALSE.)
```

Output:

```
❖ ruby main.rb
In posttest while loop
In until loop
In posttest until loop
❖
```

In the example program segment provided above, the logically controlled loops are tested in terms of the location of the test. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the 'while' loop is **not** entered, which is a pretest loop in the **Ruby** language. Only the 'while' loop is not entered, because two out of the other loops have posttest control mechanisms, meaning that the loops will at least execute once. The 'until' loop is entered because this special type of loop keeps running as long as the Boolean expression controlling the loop is false.

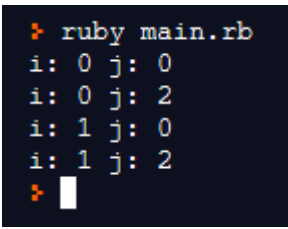
2. What kind of user-located loop control mechanisms are provided?

In **Ruby** language, the programmers are provided with the '*break*', '*next*', and '*redo*' statements to control the loops [18]. The '*next*' statement is used under the scope of an '*if*' statement. If the '*if*' statement is entered, then the '*next*' statement is executed. When it is executed it skips the current step of the loop, and the next step is started being executed, if possible. The '*break*' statement is used under the scope of an '*if*' statement, similar to the '*next*' statement. This time, if the '*if*' statement is entered, then the '*break*' statement is executed, and when it is executed it terminates the loop. Additional to these statements, there is the '*redo*' statement in the **Ruby** language, as explained previously. Whenever the '*redo*' statement is executed in a loop, the current iteration of the loop is started from the beginning. The examples below should further demonstrate these phenomena.

Input:

```
$i = 0
while $i < 3 do
  $j = 0
  if $i == 2 then
    break #If the value of i, is equal to '2', 'break' statement terminates
          #the outer loop.
  end
  while $j < 3 do
    if $j == 1 then
      $j = $j + 1
      next #If the value of j variable, is equal to '1', 'next' statement skips
           #to the beginning of the inner loop.
    end
    puts "i: #{ $i } j: #{ $j }"
    $j = $j + 1
  end
  $i = $i + 1
end
```

Output:



```
> ruby main.rb
i: 0 j: 0
i: 0 j: 2
i: 1 j: 0
i: 1 j: 2
>
```

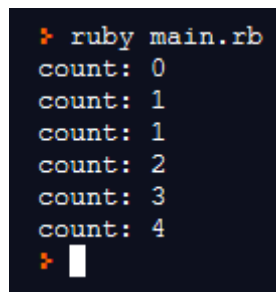
In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested while loops. The outer and inner loops are while loops, which basically count from 0 to 2. The '*next*' statement is put in an '*if*' statement, which would be executed if the variable '*j*' is equal to 1. Similarly, the '*break*' statement is put in an '*if*' statement, which would be executed if the variable '*i*' is equal to 2. If we take a look at the results of this code segment, we can see indeed, that the '*next*' and '*break*' statements are working, because indeed the step of the loop was skipped when '*j*' was equal to 1, that the value 1 for was not

printed for 'j', and the last value printed for 'i' was 1, meaning that the outer loop was terminated when the value of 'i' was equal to 2, before that value could be printed.

Input:

```
$control = false
$count = 0
while $count < 5 do
  puts "count: #{$count}"
  if $count == 1 then
    if !$control then
      $control = true
      redo #If the value of count is 1 and the boolean variable control is false,
          #the redo statement is executed, which repeats the current iteration.
    end
  end
  $count = $count + 1
end
```

Output:



```
➤ ruby main.rb
count: 0
count: 1
count: 1
count: 2
count: 3
count: 4
➤
```

In this example, the behavior of the 'goto' statement is demonstrated. The 'while' loop basically counts from 0 to 4. In the 'while' loop, there exists a nested 'if' statement, which cumulatively checks if the count is equal to 1, and the Boolean variable control is false. If the conditions are satisfied, the 'redo' statement is executed. This phenomenon can be verified if we take a look at the output of this code segment. Indeed, we see the value 1 for the count variable, meaning that for count = 1, the loop statements were executed twice.

Rust

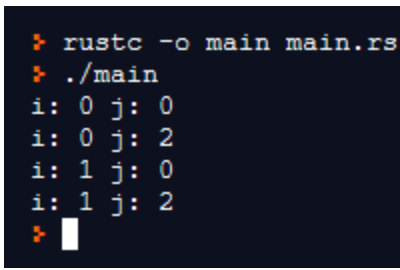
1. What are the locations of tests? Pretest, posttest, or both?

In **Rust** language, there is one type of logically controlled loop, and it is the 'while' loop [19]. The other loops are the 'for' loop and a loop mechanism, which is to be controlled by the user [19]. The 'for' loop can be categorized as a counter-controlled loop. In the 'while' loop in **Rust** language test is done before the loop statements are executed, meaning the location of the test is pretest. There are no loops in **Rust** for posttest loop control, but it can be simulated with the second mechanism stated above. This can be seen in the following code segment.

Input:

```
let mut i = 0;
while i < 3
{
    let mut j = 0;
    if i == 2
    {
        break; //If the value of i, is equal to '2', 'break' statement terminates
                //the outer loop.
    }
    while j < 3
    {
        if j == 1
        {
            j = j + 1;
            continue; //If the value of j variable, is equal to '1', 'continue' statement skips
                      //to the beginning of the inner loop.
        }
        println!("i: {} j: {}", i, j);
        j = j + 1;
    }
    i = i + 1;
}
```

Output:



```
> rustc -o main main.rs
> ./main
i: 0 j: 0
i: 0 j: 2
i: 1 j: 0
i: 1 j: 2
>
```

In the example program segment provided above, the logically controlled ‘*while*’ loop and the simulated posttest loop are tested. Initially, a variable called flag is assigned to 1. In the following parts of the program, it is made sure that the variable flag carried the same variable as it was assigned initially. The simulated posttest loop is provided to be able to see the effects of the ‘*while*’ loop. If the loops are entered they print a string indicating that a specific loop is entered, as it can be seen from the code segment too.

If we look at the output of this code segment, we can see that only the simulated loop is entered, which is a simulated posttest loop. **A POSTTEST LOOP DOES NOT EXIST IN THE RUST LANGUAGE.** The only mechanism for pretest is provided in the **Rust** language, which is working properly.

2. What kind of user-located loop control mechanisms are provided?

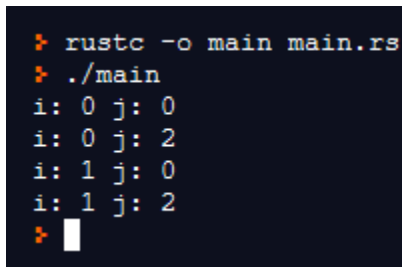
In **Rust** language, there are two user-located loop control mechanisms. These are ‘*continue*’, and ‘*break*’ statements [20] [21]. The ‘*continue*’ statement is used under the scope of an ‘*if*’ statement. If the ‘*if*’ statement is entered, then the ‘*continue*’ statement is executed. When it is

executed it skips the current step of the loop, and the next step is started being executed, if possible. The `'break'` statement is used under the scope of an `'if'` statement, similar to the `'continue'` statement. This time, if the `'if'` statement is entered, then the `'break'` statement is executed, and when it is executed it terminates the loop. The `'break'` and `'continue'` statements can be used in the counter-controlled loops in **Rust** too. In **Rust** language, there are no labels provided, which the programmers can use to jump around in the program.

Input:

```
let mut i = 0;
while i < 3
{
    let mut j = 0;
    if i == 2
    {
        break; //If the value of i, is equal to '2', 'break' statement terminates
                //the outer loop.
    }
    while j < 3
    {
        if j == 1
        {
            j = j + 1;
            continue; //If the value of j variable, is equal to '1', 'continue' statement skips
                      //to the beginning of the inner loop.
        }
        println!("i: {} j: {}", i, j);
        j = j + 1;
    }
    i = i + 1;
}
```

Output:



```
> rustc -o main main.rs
> ./main
i: 0 j: 0
i: 0 j: 2
i: 1 j: 0
i: 1 j: 2
> █
```

In the example program segment provided above, the user-located loop control mechanisms are tested. There are two nested while loops. The outer and inner loops are while loops, which basically count from 0 to 2. The `'continue'` statement is put in an `'if'` statement, which would be executed if the variable `'j'` is equal to 1. Similarly, the `'break'` statement is put in an `'if'` statement, which would be executed if the variable `'i'` is equal to 2. If we take a look at the results of this code segment, we can see indeed, that the `'continue'` and `'break'` statements are working, because indeed the step of the loop was skipped when `'j'` was equal to 1, that the value 1 for was not printed for `'j'`, and the last value printed for `'i'` was 1, meaning that the outer loop was terminated when the value of `'i'` was equal to 2, before that value could be printed.

Evaluation of the Languages in Terms of Readability & Writability

Readability

When the languages are categorized in terms of the readability in the scope of logically controlled loops, I can group them, readable, middle-ground, and less readable as follows:

<i>Readable</i>	<i>Middle-ground</i>	<i>Less Readable</i>
Dart JavaScript PHP	Lua Ruby	Python Rust

Dart: ‘while’, ‘for’, and ‘do-while’ loops options for logically controlled loops introduce more readability to the language. ‘while’ loops are usually not readable, as the operations to control the loop is done inside the loop (increment operations, etc.). This kind of practice reduces readability, but because the language has the ‘for’ loop alternative the readability is improved. The user control mechanisms are readable, the names chosen for the statements are close to the English natural language.

JavaScript: The arguments given for the **Dart** language are valid for **JavaScript** too:

‘while’, ‘for’, and ‘do-while’ loops options for logically controlled loops introduce more readability to the language. ‘while’ loops are usually not readable, as the operations to control the loop is done inside the loop (increment operations, etc.). This kind of practice reduces readability, but because the language has the ‘for’ loop alternative the readability is improved. The user control mechanisms are readable, the names chosen for the statements are close to the English natural language.

Lua: This language is categorized as middle-ground, in terms of readability, in the scope of the logically controlled loops. Language is less readable in contrast to the readable ones, because in **Lua** there exists no logically controlled ‘for’ loops, that the aforementioned loop operations (increment, etc.) must be done in the loop statements, which makes **Lua** less readable in the scope of the logically controlled loops. Additionally, the absence of a statement like ‘continue’, which when executed skips to the next iteration of the loop, is again a criterion, which makes **Lua** less readable, because to simulate the effects of such a statement, some labels and the ‘goto’ statement is used, which gives the program a complicated view. On the other hand, the presence of a posttest loop mechanism, with a name close to the English natural language (‘repeat-until’) eliminates the complex instructions, which would be otherwise used to simulate the effects of a posttest loop, increases the readability of **Lua** language. Additionally, the ‘goto’ and ‘break’ statements are close to English, and their effects are easily understandable, which makes **Lua** more readable.

PHP: The arguments given for the **Dart**, and **JavaScript** languages are valid for **PHP** too:

‘while’, ‘for’, and ‘do-while’ loops options for logically controlled loops introduce more readability to the language. ‘while’ loops are usually not readable, as the operations to control the loop is done inside the loop (increment operations, etc.). This kind of practice reduces readability, but because the language has the ‘for’ loop alternative the readability is improved. The user control mechanisms are readable, the names chosen for the statements are close to the English natural language.

Additionally, there is the *'goto'* statement is available in **PHP**, which is close to English, and its effects are easily understandable, which makes **PHP** more readable.

Python: **Python** language is categorized as less readable in contrast to other languages, in the scope of the logically controlled loops. First of all, **Python** language does not have any *'for'* loop mechanism to support the phenomenon of the logically controlled loops. The programmers are only provided with the *'while'* loop. Additionally, there exist no loops, which have a posttest control mechanism. Therefore, if the programmers want to simulate the effects of these phenomena, they will have to use additional instructions to do so, which is less readable. The statements for user control in logically controlled loops are readable, but there exists only the *'break'* and *'continue'* statements, again this time it is less common to do so in contrast to the simulated loop mechanism explained above, if the programmers want to simulate the effects of the other statements available in some other programming languages, more instructions should be written, which makes the language again, less readable.

Ruby: **Ruby** language is categorized as middle-ground, in terms of readability, in the scope of the logically controlled loops. First of all, **Ruby** does not have any logically controlled *'for'* loops, that the aforementioned loop operations (increment, etc.) must be done in the loop statements, which makes **Lua** less readable in the scope of the logically controlled loops. Additionally, the unique way that Ruby allows the programmers to convert the pretest loops to posttest ones by writing the statements before the test with the same *'begin'*, *'end'* indicators is smart, but not readable, as it is easy to confuse whilst reading the program. On contrary, after all the language has a posttest control mechanism, which allows the programmers to escape from the impractical simulation way to write posttest loops. Additionally, the names of the statements for user control are convenient, and easy to understand, which increases the readability.

Rust: The arguments given for the **Python** language are valid for **Rust** too:

Rust language is categorized as less readable in contrast to other languages, in the scope of the logically controlled loops. First of all, **Rust** language does not have any *'for'* loop mechanism to support the phenomenon of the logically controlled loops. The programmers are only provided with the *'while'* loop. Additionally, there exist no loops, which have a posttest control mechanism. Therefore, if the programmers want to simulate the effects of these phenomena, they will have to use additional instructions to do so, which is less readable. The statements for user control in logically controlled loops are readable, but there exists only the *'break'* and *'continue'* statements, again this time it is less common to do so in contrast to the simulated loop mechanism explained above, if the programmers want to simulate the effects of the other statements available in some other programming languages, more instructions should be written, which makes the language again, less readable.

Writability

When the languages are categorized in terms of the writability in the scope of logically controlled loops, I can group them, writable, and less writable as follows:

<i>Writable</i>	<i>Less Writable</i>
PHP JavaScript Dart	Lua Rust Python Ruby

Dart: **Dart** language is considered writable, in the scope of the logically controlled loops. In **Dart**, there exist mechanisms for both pretest and posttest controls. The conventions are not complicated, logic operations are similar to the ones in mathematics. The signatures of the loops are simple as well. In the user control mechanisms, the statements are again convenient so put. There are no differentiated conventions for the same operation. Hence, the **Dart** language is writable, in the scope of the logically controlled loops.

JavaScript: The arguments given for the **Dart** language are valid for **JavaScript** too:

JavaScript language is considered writable, in the scope of the logically controlled loops. In **JavaScript**, there exist mechanisms for both pretest and posttest controls. The conventions are not complicated, logic operations are similar to the ones in mathematics. The signatures of the loops are simple as well. In the user control mechanisms, the statements are again convenient so put. There are no differentiated conventions for the same operation. Hence, the **JavaScript** language is writable, in the scope of the logically controlled loops.

Lua: **Lua** language is considered less writable, in the scope of the logically controlled loops, contrasted to other programming languages. In **Lua**, there exists no '*for*' loops for logically controlled loops. This means the operations to control the loop must be done in the loop statements in a '*while*' statement. This reduces writability. In the posttest loop control mechanism, an approach is used, so that the loop is executed until the Boolean expression is false. This approach is contrary to common sense, which suggests that the loop is executed as long as the Boolean expression is true, which speaks more to common sense. In this regard, the writability is again reduced. For the user control for the loops, there exists no functionality for skipping to the next iteration of a loop, instead, the programmers are left to simulate the effects of this by using labels and '*goto*' statements, which reduce the writability drastically.

PHP: The arguments given for the **Dart**, and **JavaScript** languages are valid for **PHP** too:

PHP language is considered writable, in the scope of the logically controlled loops. In **PHP**, there exist mechanisms for both pretest and posttest controls. The conventions are not complicated, logic operations are similar to the ones in mathematics. The signatures of the loops are simple as well. In the user control mechanisms, the statements are again convenient so put. There are no differentiated conventions for the same operation. Hence, the **PHP** language is writable, in the scope of the logically controlled loops.

Additionally, there is the mechanism in **PHP** in user control mechanisms for controlling logically controlled loops, in which the programmers can skip the current iteration or terminate the loop, if there are nested loops the programmers can specify the layer of the loop that they want to perform the operation to. This way of functionality increases the writability of the language.

Python: **Python** language is considered less writable, in the scope of the logically controlled loops, contrasted to other programming languages. In **Python**, there exists no '*for*' loops for logically controlled loops. This means the operations to control the loop must be done in the loop statements in a '*while*' statement. Additionally, there exists no posttest control mechanism in **Python** for logically controlled loops. This means the operations for the posttest control mechanism must be simulated, which again reduces the writability.

Ruby: **Ruby** language is considered less writable, in the scope of the logically controlled loops, contrasted to other programming languages. The reason behind this is that firstly there exists '*for*' loops for logically controlled loops. This means the operations to control the loop must be done in the loop statements in a '*while*' statement. This reduces writability. Secondly, there are

many operations for programmers to memorize, which in the end service for the same purpose. This again reduces writability. Lastly, an approach is used, so that the loop is executed until the Boolean expression is false. This approach is contrary to common sense, which suggests that the loop is executed as long as the Boolean expression is true, which speaks more to common sense. Overall, with the stated reasons, the writability is reduced.

Rust: **Rust** language is considered less writable, in the scope of the logically controlled loops, contrasted to other programming languages. In **Rust**, there exists no '*for*' loops for logically controlled loops. This means the operations to control the loop must be done in the loop statements in a '*while*' statement. This reduces writability. Additionally, there exists no posttest control mechanism in **Rust** for logically controlled loops. This means the operations for the posttest control mechanism must be simulated in an unfamiliar structure: "loop{}", which again reduces the writability.

Learning Strategies That I Have Used

I had some prior knowledge of the languages, from the lectures, and the previous homework. Whenever it was sufficient I have tried using this knowledge. In other cases, I have used the information available online. The language definitions and the answers to the questions to other programmers were helpful, while I gained knowledge for this homework.

I have mostly used the method of trial & error, when I failed I have tried other methods, until I had a working program, though having in mind that I have gained previous knowledge about the problem I was working on. The examples of the other people in the sources I have checked were different than mine, after gaining knowledge from their implementations, I have used the trial & error method on my problem.

More technically, in the trial & error method, the output statements were the most helpful tools that I have used. This homework was about loops; therefore it is obvious, that I have gone into many infinite loops. Without the output statements, it would be impossible for me to have any idea what the loops were doing, in my trial & error method.

The sources I used whilst learning, which could be further seen in detail in the references part of the report below, included language definitions, online forums, and the tutorial sites. Also, the lecture slides were helpful.

Compilers & Interpreters I Have Used

Dart: <https://repl.it/languages/dart?v2=1>

JavaScript: Microsoft Edge web browser in my PC

Lua: <https://repl.it/languages/lua?v2=1>

PHP: <https://repl.it/@buraketistiren/RadiantVainDoom#index.php> (*I had to sign up to use this, hence the link contains my username*)

Python: PyCharm IDE in my PC

Ruby: <https://repl.it/languages/ruby>

Rust: <https://repl.it/languages/rust?v2=1>

References

- [1] W3 Adda, "Dart Loops," n.d.. [Online]. Available: <https://www.w3adda.com/dart-tutorial/dart-loops>. [Accessed 9 December 2020].
- [2] Geeks for Geeks, "Dart – Loop Control Statements (Break and Continue)," 18 June 2020. [Online]. Available: <https://www.geeksforgeeks.org/dart-loop-control-statements-break-and-continue/>. [Accessed 9 December 2020].
- [3] Tutorials Point, "Using Labels to Control the Flow," n.d.. [Online]. Available: https://www.tutorialspoint.com/dart_programming/dart_programming_loops.htm. [Accessed 9 December 2020].
- [4] Mozilla, "Loops and iteration," 18 July 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration. [Accessed 9 December 2020].
- [5] W3 Schools, "JavaScript Break and Continue," n.d.. [Online]. Available: https://www.w3schools.com/js/js_break.asp. [Accessed 9 December 2020].
- [6] Mozilla, "label," n.d.. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label>. [Accessed 9 December 2020].
- [7] Tutorials Point, "Lua - Loops," n.d.. [Online]. Available: https://www.tutorialspoint.com/lua/lua_loops.htm. [Accessed 9 December 2020].
- [8] Tutorials Point, "Lua - for Loop," n.d.. [Online]. Available: https://www.tutorialspoint.com/lua/lua_for_loop.htm. [Accessed 9 December 2020].
- [9] Tutorials Point, "Lua - break Statement," n.d.. [Online]. Available: https://www.tutorialspoint.com/lua/lua_break_statement.htm. [Accessed 9 December 2020].
- [10] lua-users, "Goto Statement," 2 February 2014. [Online]. Available: <http://lua-users.org/wiki/GotoStatement>. [Accessed 9 December 2020].
- [11] W3 Schools, "PHP Loops," n.d.. [Online]. Available: https://www.w3schools.com/php/php_looping.asp. [Accessed 9 December 2020].
- [12] W3 Schools, "PHP Break and Continue," n.d.. [Online]. Available: https://www.w3schools.com/php/php_looping_break.asp. [Accessed 9 December 2020].
- [13] The PHP Group, "goto," n.d.. [Online]. Available: <https://www.php.net/manual/en/control-structures.goto.php>. [Accessed 9 December 2020].
- [14] The PHP Group, "continue," n.d.. [Online]. Available: <https://www.php.net/manual/en/control-structures.continue.php>. [Accessed 9 December 2020].
- [15] The PHP Group, "break," n.d.. [Online]. Available: <https://www.php.net/manual/en/control-structures.break.php#control-structures.break>. [Accessed 9 December 2020].

- [16] Programiz, "Python break and continue," n.d.. [Online]. Available: <https://www.programiz.com/python-programming/break-continue>. [Accessed 10 December 2020].
- [17] Tutorials Point, "Ruby - Loops," n.d.. [Online]. Available: https://www.tutorialspoint.com/ruby/ruby_loops.htm. [Accessed 11 December 2020].
- [18] J. Castello, "Understanding The Ruby Next & Break Keywords," September 2020. [Online]. Available: <https://www.rubyguides.com/2019/09/ruby-next-break-keywords/>. [Accessed 11 December 2020].
- [19] The Rust Reference, "Loops," n.d.. [Online]. Available: <https://doc.rust-lang.org/reference/expressions/loop-expr.html#iterator-loops>. [Accessed 11 December 2020].
- [20] Rust, "Keyword break," n.d.. [Online]. Available: <https://doc.rust-lang.org/std/keyword.break.html>. [Accessed 11 December 2020].
- [21] Rust, "Keyword continue," n.d.. [Online]. Available: <https://doc.rust-lang.org/std/keyword.continue.html>. [Accessed 11 December 2020].