Bilkent University

CS 315

Homework - 2

Report

Logically Controlled Loops

Dart, Javascript, Lua, Php

Python, Ruby, Rust

İrem Seven

21704269

Logically Controlled Loops

1 - Dart

while loop

In Dart language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
//while loop
    var flag = true;
    var count = 0;
    while( flag == true){
      if(count == 5){
        flag = false;
      }
      count++;
    }
   print("while loop terminated with flag, count: ${count}");
```

Output:

```
while loop terminated with flag, count: 6
```

break statement

Dart provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
//while loop with break
    flag = true;
    count = 0;
    while( flag == true){
      if(count == 5){
        break;
      }
      count++;
    }
  print("while loop terminated with break, count: ${count}\n");
```

Output:

```
while loop terminated with break, count: 5
```

user-located loop control mechanism

Loops can be labeled so that when a break is reached the loop that has the label specified will be terminated. Below is shown an example program that uses both labeled break and both not labeled break. The outer while loop is terminated only when both the k and i values reach the middle value, since when the *break outer;* is executed. On the other hand, the break without a label only terminates the loop that is closer to where the break is used, which is the inner loop.

```
//labeled break
   var i = 1;
   var k = 5;
   outer:
   while (i <= 5) {
         k = 5;
         while(k >= 1){
             print("i : ${i} k : ${k}");
             if (k == i ) {
                 if (i == 3) {
                     print("reached middle value");
                     break outer;
                 }
                 print("inner terminating with i : ${i} k : ${k}");
                 break ;
             }
             k--;
         }
      i++;
   }
```

Output:

```
i : 1 k : 5
i : 1 k : 4
i : 1 k : 3
i : 1 k : 2
i : 1 k : 1
inner terminating with i : 1 k : 1
```

```
i : 2 k : 5
i : 2 k : 4
i : 2 k : 3
i : 2 k : 2
inner terminating with i : 2 k : 2
i : 3 k : 5
i : 3 k : 4
i : 3 k : 3
reached middle value
```

continue statement
Dart provides continue statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below find the sum of even numbers. When the continue has executed the rest of the statement after the continue is not executed, the loop passes to the next iteration.
NOTE: continue statement can also be labeled as in the previous example.

```
//continue example
  var num = 1;
  var sum = 0;
  while(num <= 15) {
    if (num % 2 != 0) {
      num++;
      continue;  //pass the rest of the loop
    }
    sum = sum + num;
    num++;
  }
  print("\nsum of even values from 1 to 15 is ${sum}");
```

Output:

```
sum of even values from 1 to 15 is 56
```

logically controlled for loop
In Dart, the for loop can also be logically controlled. An example is shown below which does the same thing as in the while loop example.

```
//for loop
   flag = true;
```

```
  count = 0;
  for(;flag == true;){
    if(count == 5){
      flag = false;
    }
   count++;
  }
 print("for loop terminated with flag, count: ${count}");
```

Output:

```
for loop terminated with flag, count: 6
```

do while loop

In Dart the do while loop structure enables posttest control. This means that the condition is checked after the loop is executed for one time. Below is shown an example usage of do while loops.

```
 //do while loop
  flag = false;
  count = 0;
  do{
    if(count == 5){
      flag = false;
    }
    count++;
  }while( flag == true);
 print("do while loop terminated with flag, count: ${count}");
```

Output:

```
do while loop terminated with flag, count: 1
```

2 - Javascript

while loop

In Javascript language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
//while loop
var flag = true;
var count = 0;
while( flag == true){
    if(count == 5){
       flag = false;
     }
  count++;
}
document.write('<br>' + "while loop terminated with flag, count: " + count + '<br>');
```

Output:

```
while loop terminated with flag, count: 6
```

break statement

Javascript provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
//while loop with break
flag = true;
count = 0;
while( flag == true){
    if(count == 5){
        break;
    }
```

```
   count++;
}
document.write('<br>' + "while loop terminated with break, count: " + count + '<br>');
```

Output:

```
while loop terminated with break, count: 5
```

user-located loop control mechanism

Loops can be labeled so that when a break is reached the loop that has the label specified will be terminated. Below is shown an example program that uses both labeled break and both not labeled break. The outer while loop is terminated only when both the k and i values reach the middle value, since when the *break outer;* is executed. On the other hand, the break without a label only terminates the loop that is closer to where the break is used, which is the inner loop.

```
//labeled break
var i = 1;
var k = 5;
outer:
while (i <= 5) {
    k = 5;
    while(k >= 1){
        print("i : ${i} k : ${k}");
        document.write('<br>' + "i: " + i + " k :" + k  + '<br>');
          if (k == i ) {
             if (i == 3) {
                 document.write("reached middle value" +'<br>');
                 break outer;
             }
             print("inner terminating with i : ${i} k : ${k}");
             document.write("inner terminating with i: " + i +  "k :" + k +'<br>');
           break ;
        }
          k--;
       }
    i++;
}
```

Output:

```
i: 1 k :5

i: 1 k :4

i: 1 k :3

i: 1 k :2

i: 1 k :1
inner terminating with i: 1k :1

i: 2 k :5

i: 2 k :4

i: 2 k :3

i: 2 k :2
inner terminating with i: 2k :2

i: 3 k :5

i: 3 k :4

i: 3 k :3
reached middle value
```

continue statement

Javascript provides the continue statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below finds the sum of even numbers. When the continue has executed the rest of the statement after the continue is not executed, the loop passes to the next iteration.

NOTE: continue statement can also be labeled as in the previous example.

```
//continue example
var num = 1;
var sum = 0;
while(num <= 15) {
   if (num % 2 != 0) {
      num++;
      continue;  //pass the rest of the loop
```

```
    }
   sum = sum + num;
   num++;
}
print("\nsum of even values from ${num} to ${limit} is ${sum}");
document.write('<br>' + "sum of even values from 1 to 15 is " + sum  + '<br>');
```

Output:

```
sum of even values from 1 to 15 is 56
```

logically controlled for loop

In Javascript, the for loop can also be logically controlled. An example is shown below which does the same thing as in the while loop example.

```
//for loop
flag = true;
count = 0;
for( ;flag == true;){
   if(count == 5){
      flag = false;
    }
  count++;
}
document.write('<br>' + "for loop terminated with flag, count: " + count + '<br>');
```

Output:

```
for loop terminated with flag, count: 6
```

do while loop

In Javascript the do while loop structure enables posttest control. This means that the condition is checked after the loop is executed for one time. Below is shown an example usage of do while loops.

```
//do while loop
flag = false;
count = 0;
{
    if(count == 5){
        flag = false;
    }
  count++;
}while( flag == true);
document.write('<br>' + "do while loop terminated with flag, count: " + count + '<br>');
```

Output:

```
do while loop terminated with flag, count: 1
```

3 - Lua

while loop

In Lua language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
--while loop
flag = true
count = 0
while( flag == true)
do
    if(count == 5)
    then
        flag = false
    end

count = count + 1
end
print("while loop terminated with flag, count: ",count)
```

```
while loop terminated with flag, count:        6
```

break statement

Lua provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
--while loop with break
flag = true;
count = 0;
while( flag == true)
do
    if(count == 5)
    then
      break
    end
count = count + 1
end

print("while loop terminated with break, count: ", count)
```

```
while loop terminated with break, count:        5
```

goto statement

There is no continue in Lua goto might be used instead. Also, the labeled structure can be achieved by the *goto* statement. An example usage is shown below. The example below finds the sum of even numbers.

```
--no continue in lua, goto might be used instead
  num = 1
  sum = 0
  ::redo::
  while(num <= 15)
  do
    if (num % 2 == 1)
    then
      num = num + 1
      goto redo
    end
    sum = sum + num
    num = num + 1
  end
  print("sum of even values from 1 to 15 is ", sum);
```

```
sum of even values from 1 to 15 is     56
```

repeat until structure

In Lua, the repeat until loop structure enables posttest control. This means that the condition is checked after the loop is executed for one time. Unlike do while loop in other languages repeat until repeats loop until the condition is satisfied, so the condition should not be satisfied to loop to be executed. Below is shown an example usage.

```
--posttest loop
  flag = true
  count = 0
  repeat
      if(count == 5)
      then
        flag = false
      end
      count = count + 1
  until( flag == false) --repeats loop until the condition is statisfied
print("repeat loop terminated, count: ", count)
```

```
repeat loop terminated, count:          6
```

4 - Php

while loop

In Php language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
//while
$flag = true;
$count = 0;
while( $flag == true){
    if($count == 5){
      $flag = false;
    }
    $count++;
}
echo "while loop terminated with flag, count: $count <br>" ;
```

```
while loop terminated with flag, count: 6
```

break statement

Php provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
//break
$flag = true;
$count = 0;
while( $flag == true){
    if($count == 5){
      break;
    }
    $count++;
}
echo "while loop terminated with break, count: $count <br>" ;
```

```
while loop terminated with break, count: 5
```

user-located loop control mechanism

Statements such as break and continue can be numbered. Below is shown an example program that uses both numbered breaks. The outer while loop is terminated only when both the k and i values reach the middle value, since when the *break 2;* is executed. The reason is that the loop we want to terminate is the second loop relative to the break statement, where is written. On the other hand, the break numbered as one terminates the loop that is closer, the first one relative to the break, which is the inner loop.

```
//labels as numbered loops
        $i = 1;
   $k = 5;
   while ($i <= 5) {
       $k = 5;
       while($k >= 1){
           echo "i : $i k : $k <br>";
           if ($k == $i ) {
               if ($i == 3) {
                   echo "reached middle value <br>";
                   break 2;
               }
               echo "inner terminating with i : $i k : $k <br>";
               break 1;
           }
           $k--;
       }
     $i++;
   }
```

```
i : 1 k : 5
i : 1 k : 4
i : 1 k : 3
```

```
i : 1 k : 2
i : 1 k : 1
inner terminating with i : 1 k : 1
i : 2 k : 5
i : 2 k : 4
i : 2 k : 3
i : 2 k : 2
inner terminating with i : 2 k : 2
i : 3 k : 5
i : 3 k : 4
i : 3 k : 3
reached middle value
```

continue statement

Php provides the continue statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below finds the sum of even numbers. When the continue has executed the rest of the statement after the continue is not executed, the loop passes to the next iteration.

NOTE: continue statement can also be numbered as in the previous example.

```
//continue
  $num = 1;
  $sum = 0;
  while($num <= 15) {
    if ($num % 2 != 0) {
      $num++;
      continue;  //pass the rest of the loop
    }
    $sum = $sum + $num;
    $num++;
  }
  echo "<br> sum of even values from 1 to 15 is $sum <br>";
```

```
sum of even values from 1 to 15 is 56
```

logically controlled for loop

In Php, the for loop can also be logically controlled. An example is shown below which does the same thing as in the while loop example.

```
//for loop
            $flag = true;
    $count = 0;
    for(;$flag == true;){
      if($count == 5){
        $flag = false;
      }
     $count++;
    }
   echo "for loop terminated with flag, count: $count <br>";
```

```
for loop terminated with flag, count: 6
```

do while loop

In Php, the do while loop structure enables posttest control. This means that the condition is checked after the loop is executed for one time. Below is shown an example usage of do while loops.

```
//do while
            $flag = false;
    $count = 0;
    do{
      if($count == 5){
        $flag = false;
      }
      $count++;
    }while( $flag == true);
   echo "do while loop terminated with flag, count: $count";
```

```
do while loop terminated with flag, count: 1
```

5 - Python

while loop

In Python language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
# while
flag = True
count = 0
while( flag == True):
    if(count == 5):
        flag = False
    count = count + 1
print("while loop terminated with flag, count:",count)
```

```
while loop terminated with flag, count: 6
```

break statement

Python provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
flag = True
count = 0
while( flag == True):
    if(count == 5):
        break
    count = count + 1
print("while loop terminated with break, count:",count)
```

```
while loop terminated with break, count: 5
```

continue statement

Python provides the continue statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below finds the sum of even numbers. When the continue has executed the rest of the statement after the continue is not executed, the loop passes to the next iteration.

```
#continue
num = 1
sum = 0
while(num <= 15):
   if (num % 2 != 0):
      num = num + 1
      continue;
   sum = sum + num
   num = num + 1
print("\nsum of even values from 1 to 15 is" , sum)
```

```
sum of even values from 1 to 15 is 56
```

additional notes

→ There is no user-located loop control in Python.

→ The for loop in Python is not logically controlled.

→ There is no posttest loop structure in Python.

6 - Ruby

while loop

In Ruby language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
#while
```

```
$flag = true
$count = 0
while ( $flag == true) do
    if($count == 5)
      $flag = false
    end
    $count = $count + 1
end

puts("while loop terminated with flag, count: #$count")
```

```
while loop terminated with flag, count: 6
```

break statement

Ruby provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
#break
$flag = true
$count = 0
while( $flag == true) do
    if($count == 5)
      break
    end
    $count = $count + 1
end

puts("while loop terminated with break, count: #$count")
```

```
while loop terminated with break, count: 5
```

next statement

Ruby provides the next statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below finds the sum of even numbers. When the next has executed the rest of the statement after the next is not executed, the loop passes to the next iteration.

```
#next
$num = 1;
$sum = 0;
while($num <= 15) do
   if ($num % 2 != 0)
      $num = $num + 1
      next
   end
     $sum = $sum + $num
     $num = $num + 1
end

puts("sum of even values from 1 to 15 is #$sum");
```

```
sum of even values from 1 to 15 is 56
```

until loop

Ruby provides an alternative pretesting method which is until. Until works as while loop the difference is that it executes the loop when the condition is not satisfied. It terminates iteration only when the condition is not satisfied.

```
#until -- pretest
$flag = true
$count = 0
until $flag == false do
     if($count == 5)
       $flag = false
     end
     $count = $count + 1
end
puts("until end loop terminated with flag, count: #$count")
```

```
until end loop terminated with flag, count: 6
```

begin end loop

In Ruby, the begin end loop structure enables posttest control. This means that the condition is checked after the loop is executed for one time. Below is shown an example usage.

```
#begin end loop - posttest
$flag = false
$count = 0
begin
    if($count == 5)
      $flag = false
    end
    $count = $count + 1
end while $flag == true

puts("begin end loop terminated with flag, count: #$count")
```

```
begin end loop terminated with flag, count: 1
```

additional notes

→ There is no user-located loop control in Ruby.

→ The for loop in Ruby is not logically controlled.

7 - Rust

while loop

In Rust language, there is a while loop structure that enables to perform logic checking before entering inside the loop. In other words, the while loop is for pretesting the logic condition. Below is shown an example code segment for the while loop. It checks the condition in the parenthesis before entering the loop. Then, if the condition is satisfied it executes the loop until the condition becomes false. In this example the flag becomes false when the count becomes 5, then the counter is incremented again. The next time the loop checked it becomes false, thus the execution of the while loop is terminated.

```
// while loop
   let mut flag = true;
   let mut count = 0;
   while flag == true {
     if count == 5 {
       flag = false;
     }
     count = count + 1;
   }
   println!("while loop terminated with flag, count: {}", count);
```

```
while loop terminated with flag, count: 6
```

break statement

Rust provides break statement as a user-located loop control mechanism. It simply terminated the executed loop. The example below does the same thing as the previous example, except it uses break statement rather than setting the flag value. This way it immediately terminates the loop, so the counter is not incremented after break is reached.

```
//break
   flag = true;
   count = 0;
   while flag == true {
     if count == 5 {
          break;
     }
     count = count + 1;
   }
   println!("while loop terminated with break, count: {}", count);
```

```
while loop terminated with break, count: 5
```

user-located loop control mechanism

Loops can be labeled so that when a break is reached the loop that has the label specified will be terminated. Below is shown an example program that uses both labeled break and both not labeled break. The outer while loop is terminated only when both the k and i values reach the middle value, since when the *break outer;* is executed. On the other hand, the break without a label only terminates the loop that is closer to where the break is used, which is the inner loop.

```
//labeled break
   let mut i = 1;
   let mut k = 5;
   'outer:
   while i <= 5 {
        k = 5;
        while k >= 1 {
            println!("i : {} k : {}", i , k);
            if k == i  {
                if i == 3 {
                    println!("reached middle value");
                    break 'outer;
                }
                println!("inner terminating with i : {} k : {}", i, k);
                break ;
            }
            k = k - 1;
        }
      i = i + 1;
   }
```

```
i : 1 k : 5
i : 1 k : 4
i : 1 k : 3
i : 1 k : 2
i : 1 k : 1
inner terminating with i : 1 k : 1
i : 2 k : 5
i : 2 k : 4
i : 2 k : 3
i : 2 k : 2
inner terminating with i : 2 k : 2
i : 3 k : 5
```

```
i : 3 k : 4
i : 3 k : 3
reached middle value
```

continue statement
Rust provides continue statement as another user-located loop control mechanism. It dismisses the rest of the loop and executes the next loop state. The example below find the sum of even numbers. When the continue has executed the rest of the statement after the continue is not executed, the loop passes to the next iteration.
NOTE: continue statement can also be labeled as in the previous example.

```
//continue
   let mut num = 1;
   let mut sum = 0;
  while num <= 15 {
    if num % 2 != 0 {
       num = num + 1;
       continue;  //pass the rest of the loop
    }
    sum = sum + num;
    num = num + 1;
  }
  println!("sum of even values from 1 to 15 is {}", sum);
```

```
sum of even values from 1 to 15 is 56
```

additional notes
→ The for loop in Rust is not logically controlled.
→ There is no posttest loop structure in Rust.

Evaluation of Languages

Python is the best language in terms of readability and writability since it does not allow labels which adds complexity to these design issues. However, it decreases the flexibility for programming. It provides simple but useful features. Also, it does not allow posttest which makes the language more complicated, since the same features can be implemented without a structure as do while.

Learning Strategy

To learn logically controlled structures in seven different languages, I mostly used internet sources. I have read manuals from the official websites of the languages. Apart from official manuals, I have used blogs such as GeeksforGeeks for examples of code snippets to discover the syntax and working principles of the languages. Since my task was on logically controlled structures, I have focused specifically on learning logically controlled loop structures in these languages rather than focusing on the whole language design. Then I tested the given examples using online compilers. The examples generally were focused on one point and they were so simple. For other features, I had to do research on the internet separately. Finally, I started creating example programs myself by combining the information I have gained. During this process, there were many problems I encountered and I was able to solve them by searching on the internet. Stack Overflow, for example, has been very useful for this purpose. I decided to use the same examples for each language since I thought it would be more beneficial to readers. That way the differences and similarities between the languages became easier to understand.

4 Compilers/Interpreters

The online interpreters and compilers used for testing are given below.

Dart: https://dartpad.dev/

Javascript: https://js.do/

Lua: https://www.tutorialspoint.com/execute_lua_online.php

PHP: https://www.w3schools.com/php/phptryit.asp?filename=tryphp_compiler

Python: https://www.programiz.com/python-programming/online-compiler/

Ruby: https://www.tutorialspoint.com/execute_ruby_online.php

Rust: https://play.rust-lang.org