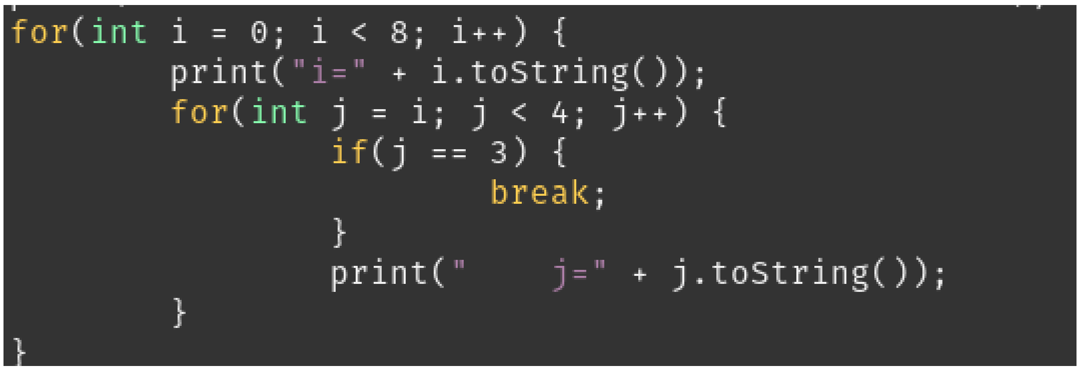
## 1) Code Explanations

**Dart:**

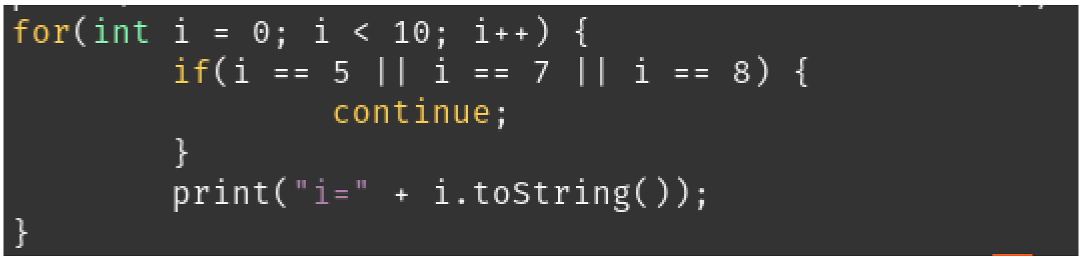
In Dart, both unconditional and labeled exits exist.



```
for(int i = 0; i < 8; i++) {
        print("i=" + i.toString());
        for(int j = i; j < 4; j++) {
                if(j == 3) {
                        break;
                }
                print("    j=" + j.toString());
        }
}
```

*Figure 1: Existence of Break in Dart*

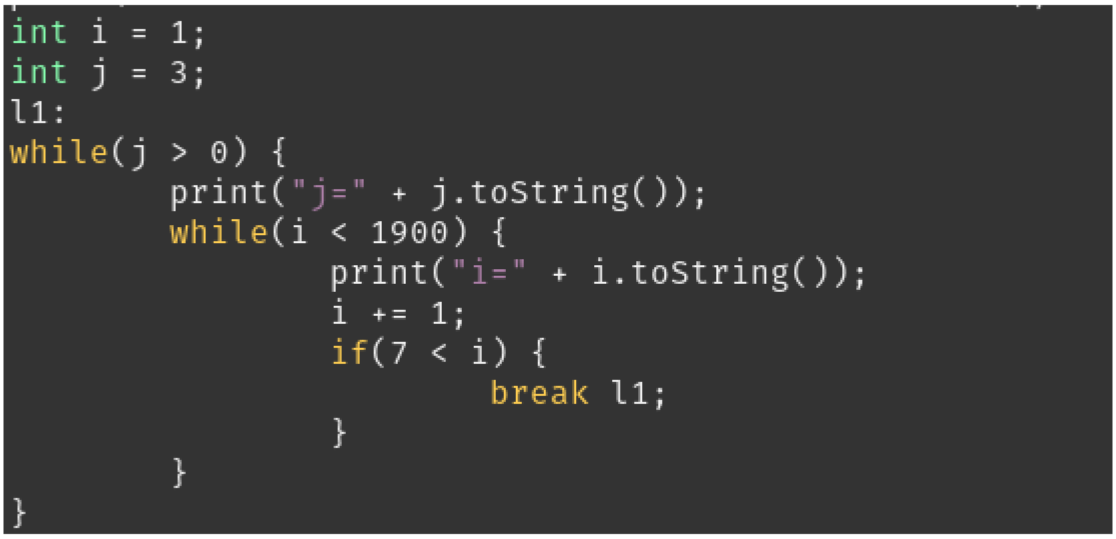
In Figure 1, the break statement terminates the inner loop. For i values less than 4, j gets integer value 3 eventually. Thus, for i values less than 4, the inner loop will be terminated when j reaches 3 for sure. The outer loops iteration is not effected by this break statement, but the inner loop is.



```
for(int i = 0; i < 10; i++) {
        if(i == 5 || i == 7 || i == 8) {
                continue;
        }
        print("i=" + i.toString());
}
```

*Figure 2: Existence of Continue in Dart*

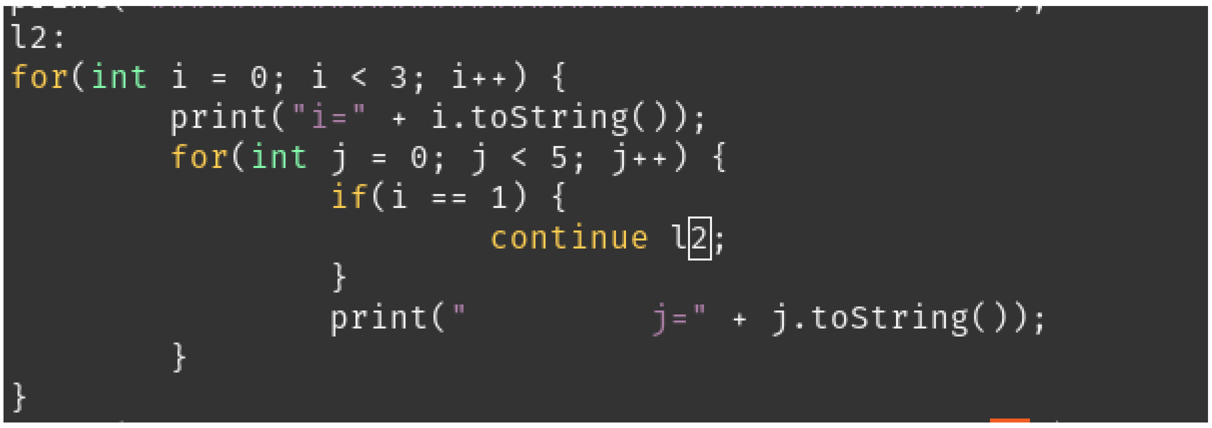
In Figure 2, the continue statement terminates the for loops iterations if i == 5, 7, or 8. Hence, in these iterations, we do not see the i values printed as the iteration terminates before it reaches to print statement.



```
int i = 1;
int j = 3;
l1:
while(j > 0) {
        print("j=" + j.toString());
        while(i < 1900) {
                print("i=" + i.toString());
                i += 1;
                if(7 < i) {
                        break l1;
                }
        }
}
```

*Figure 3: Labeled Break in Dart*

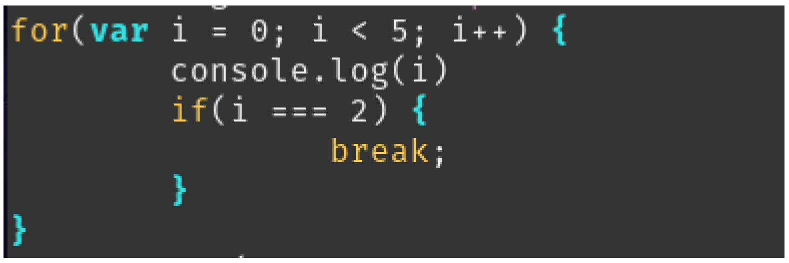
In Figure 3, the outer while loop has a label: l1. We only see one j value, 3, being printed, as the outer while loop is terminated when i > 7. This condition is satisfied in the first iteration of the outer while loop.


```
l2:
for(int i = 0; i < 3; i++) {
        print("i=" + i.toString());
        for(int j = 0; j < 5; j++) {
                if(i == 1) {
                        continue l2;
                }
                print("        j=" + j.toString());
        }
}
```
*Figure 4: Labeled Continue in Dart*

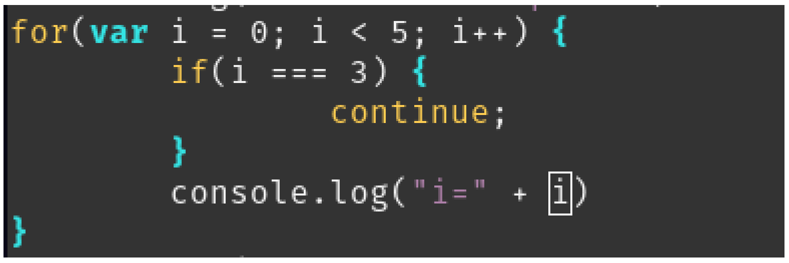
In Figure 4, the outer loop is labeled as l2. What this loop does is it prints j values from 0 to 4 for i values 0, 1, and 2. We do not see any j value being printed for i === 1, as we skip that iteration of the outer loop.

## Javascript:

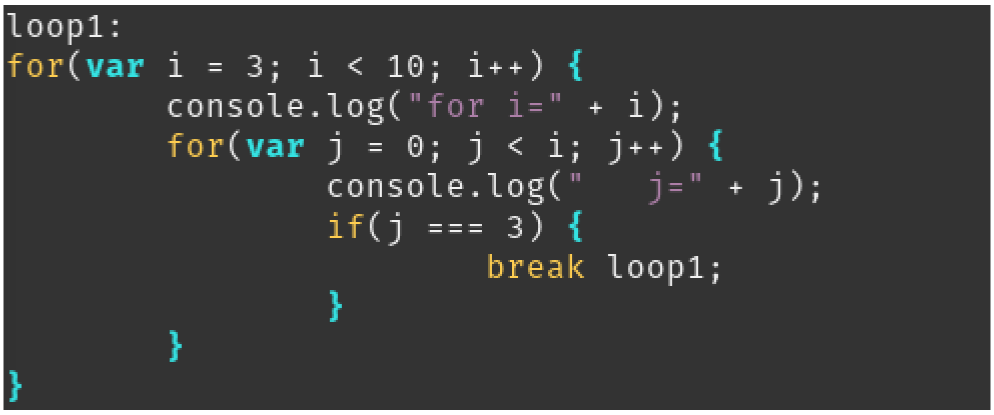
In Javascript, both unconditional and labeled exits exist.


```
for(var i = 0; i < 5; i++) {
        console.log(i)
        if(i === 2) {
                break;
        }
}
```
*Figure 5: Break in Javascript*

In Figure 5, the for loop will not print values greater than 2, as when i === 2, it is terminated with a break statement.


```
for(var i = 0; i < 5; i++) {
        if(i === 3) {
                continue;
        }
        console.log("i=" + i)
}
```
*Figure 6: Continue in Javascript*

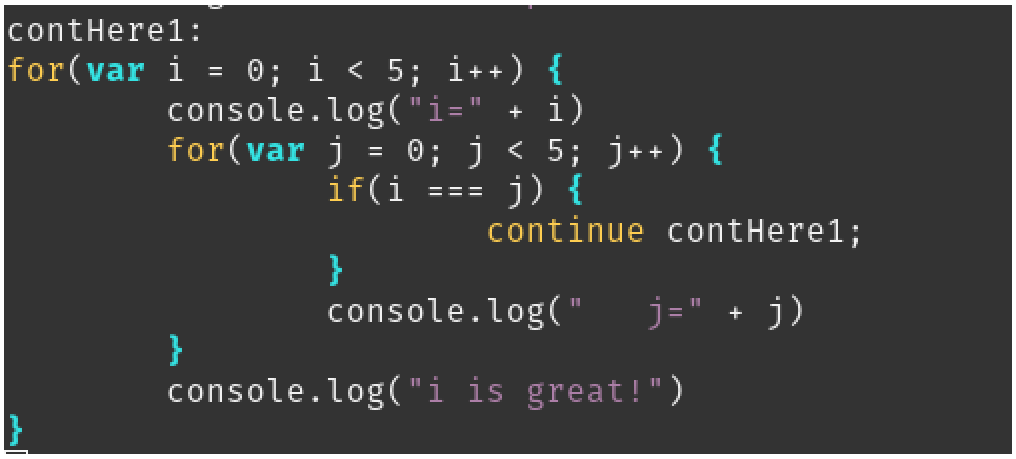
In Figure 6, the loop will not print “i=3”, as the iteration is terminated when i === 3.



```
loop1:
for(var i = 3; i < 10; i++) {
        console.log("for i=" + i);
        for(var j = 0; j < i; j++) {
                console.log("    j=" + j);
                if(j === 3) {
                        break loop1;
                }
        }
}
```

*Figure 7: Labeled Break in Javascript*

In Figure 7, the first iteration of the outer for loop proceeds normally, but in the second iteration, j does take value 3 at some point and thus, the outer for loop is terminated. As a result, we do not see i values greater than 4 being printed.



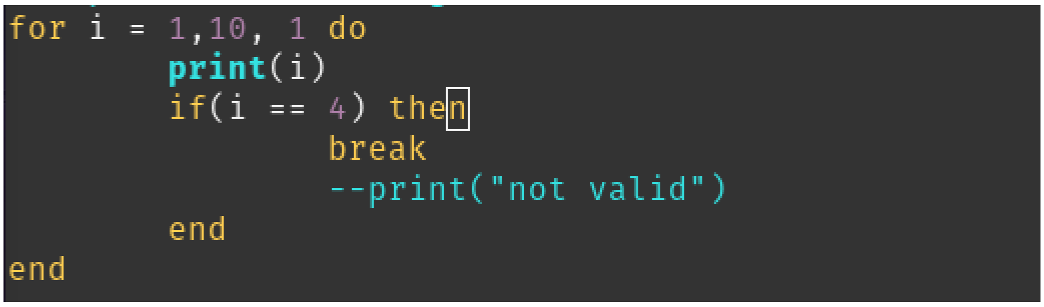
```
contHere1:
for(var i = 0; i < 5; i++) {
        console.log("i=" + i)
        for(var j = 0; j < 5; j++) {
                if(i === j) {
                        continue contHere1;
                }
                console.log("    j=" + j)
        }
        console.log("i is great!")
}
```

*Figure 8: Labeled Continue in Javascript*

In Figure 8, ve never see j values greater or equal to i values, as whenever i === j, we terminate the iteration of the outer loop.

## Lua:

In Lua, the “break” statement exists, but continue doesn’t. Also, it is impossible to use labels to terminate outer loops in nested loops.



```
for i = 1,10, 1 do
        print(i)
        if(i == 4) then
                break
                --print("not valid")
        end
end
```

*Figure 9: Break in Lua*

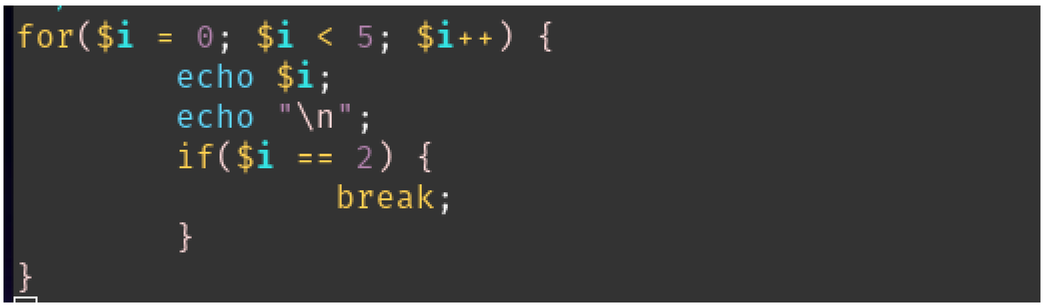
Figure 9 shows how to use break statement in Lua. First think that is noteworthy is that it is not allowed to put any other statements after break statement in the same block. The reason is justifiable: If it is never reached, why would it be allowed? This is a feature of Lua that helps with readability but harms writability.

The “continue” statement is invalid in Lua. The developers of Lua state that **“continue was only one of a number of possible new control flow mechanisms”**[1]**.** It seems like they didn’t put it into the language since there were other ways to do it, like goto statements.

In Lua, labels do exist but can’t be used with “break”. The reason is the same as not putting “continue” in the language. It is possible to use goto statements with labels to perform the same functionality of labeled “break”.

## PHP:

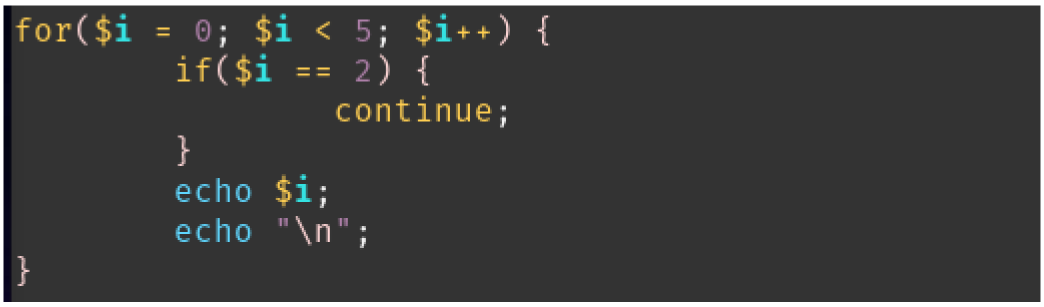
Both unconditional and labeled exits exist in PHP. Labeled exit is made without using labels. It is done by using scope level.



```
for($i = 0; $i < 5; $i++) {
        echo $i;
        echo "\n";
        if($i == 2) {
                break;
        }
}
```

*Figure 10: Break in PHP*

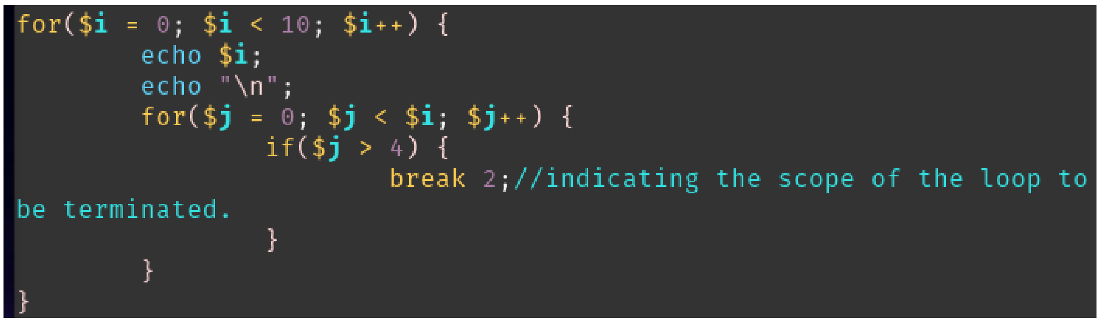
In Figure 10, we do not see values greater than 2 being printed, as the loop is terminated when $i == 2.



```
for($i = 0; $i < 5; $i++) {
        if($i == 2) {
                continue;
        }
        echo $i;
        echo "\n";
}
```

*Figure 11: Continue in PHP*

In Figure 11, we do not see 2 being printed, as the iteration is terminated before it reaches to echo statement when $i == 2.

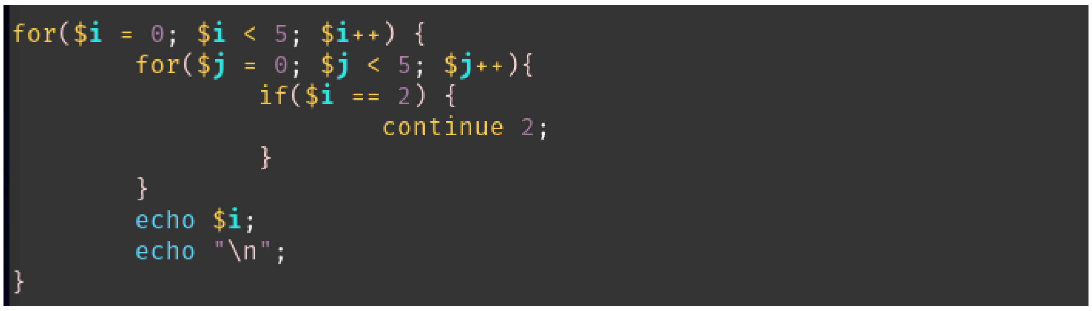


```
for($i = 0; $i < 10; $i++) {
	echo $i;
	echo "\n";
	for($j = 0; $j < $i; $j++) {
		if($j > 4) {
			break 2;//indicating the scope of the loop to
be terminated.
		}
	}
}
```

*Figure 12: Break with Scope*

In Figure 12, we see how “break” statement can be used with scope levels. In this instance, the outer loop will be terminated when the $j > 4 is satisfied, as it is indicated by its scope number “2”. The outer loop is the second loop that break statement has as parent.

The same logic is valid with “continue” too.



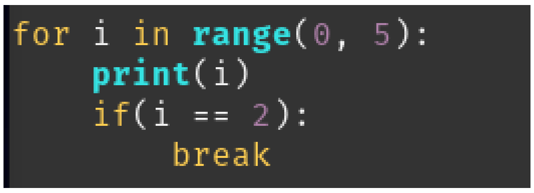
```
for($i = 0; $i < 5; $i++) {
	for($j = 0; $j < 5; $j++){
		if($i == 2) {
			continue 2;
		}
	}
	echo $i;
	echo "\n";
}
```

*Figure 13: Continue with Scope*

In Figure 13, a single iteration of the outer loop is terminated.

## Python:

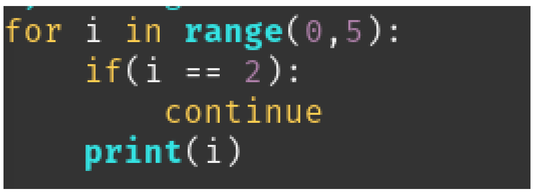
Both break and continue statements exist in Python.



```
for i in range(0, 5):
    print(i)
    if(i == 2):
        break
```

*Figure 14: Break in Python*

In Figure 14, the loop will be terminated when i == 2. Thus, values greater than 2 won’t be printed.



```
for i in range(0,5):
    if(i == 2):
        continue
    print(i)
```

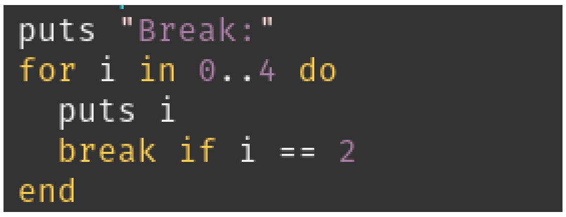
*Figure 15: Continue in Python*

In Figure 15, the value 2 will not be printed as the iteration will be terminated before it reaches the print statement when i == 2.

In Python, labels do not exist. Thus, labeled exits do not exist. It is possible to find some workaround solutions to this, like adding flag variables or putting loops in a function and using return statements.

## Ruby:

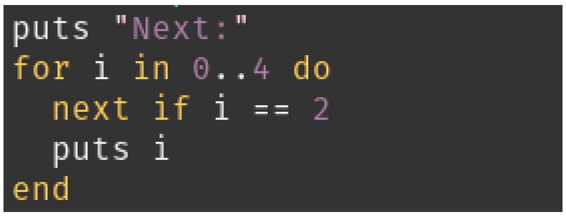
In Ruby, both the functionality of “break” and “continue” exist. The “next” keyword is used instead of “continue”.



```
puts "Break:"
for i in 0..4 do
  puts i
  break if i == 2
end
```

*Figure 16: Break in Ruby*

In Figure 16, values greater than 2 will not be printed, as the loop is terminated when i == 2.



```
puts "Next:"
for i in 0..4 do
  next if i == 2
  puts i
end
```

*Figure 17: Next in Ruby*

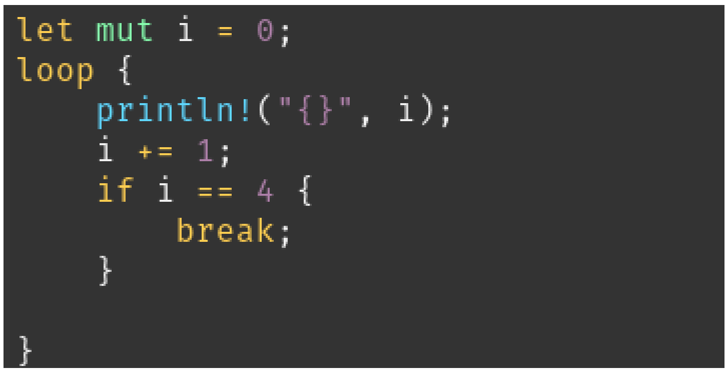
In Figure 16, value 2 will not be printed as the iteration will be terminated before it reaches puts statement when i == 2.

In Ruby, labeled break and continue statements do not exist. It is possible to use Continuations, Exceptions, and trow/catch instead.

## Rust:

Firstly, I experimented with Rust’s infinite loops in this assignment. Thus, the code segments in this part may seem a bit complicated. In the parts where I explain the functionality of “continue,” I also used “break” statements to exit infinite loops.

In Rust, both labeled and unconditional exits exist.

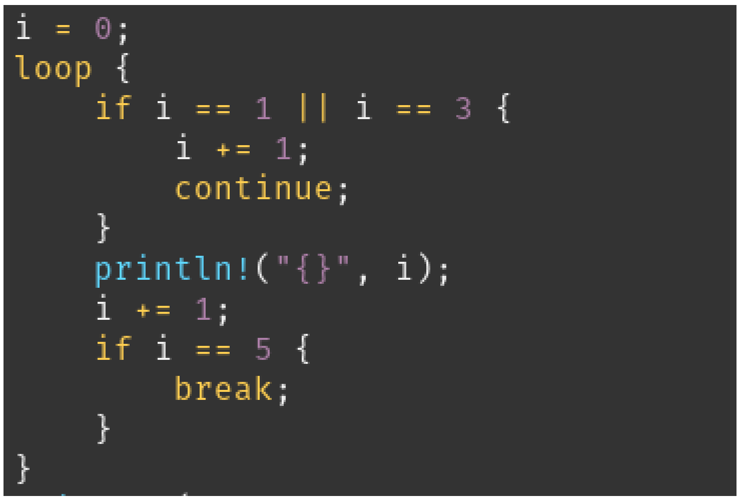


```rust
let mut i = 0;
loop {
    println!("{}", i);
    i += 1;
    if i == 4 {
        break;
    }

}
```

*Figure 18: Break in Rust*

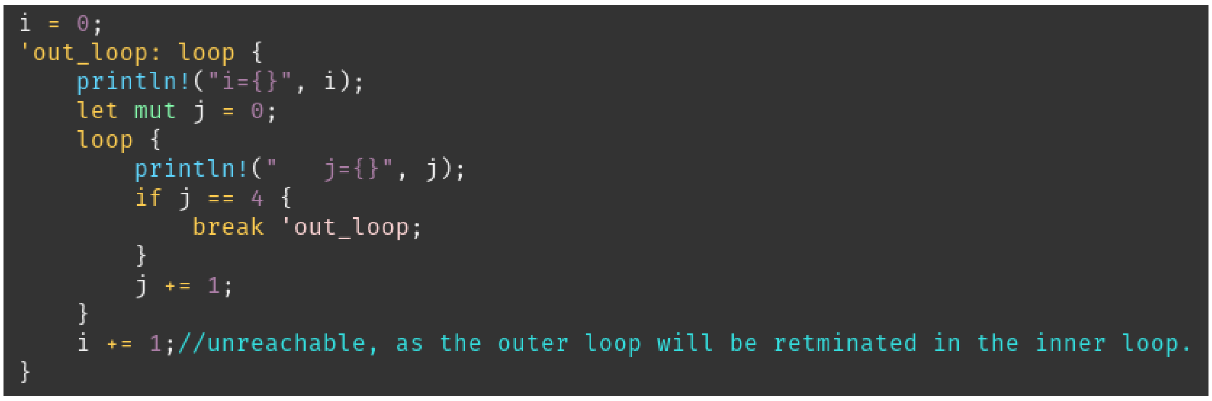
In Figure 18, the values greater than 4 will not be printed, as the infinite loop is terminated when i == 4.



```rust
i = 0;
loop {
    if i == 1 || i == 3 {
        i += 1;
        continue;
    }
    println!("{}", i);
    i += 1;
    if i == 5 {
        break;
    }
}
```

*Figure 19: Continue in Rust*

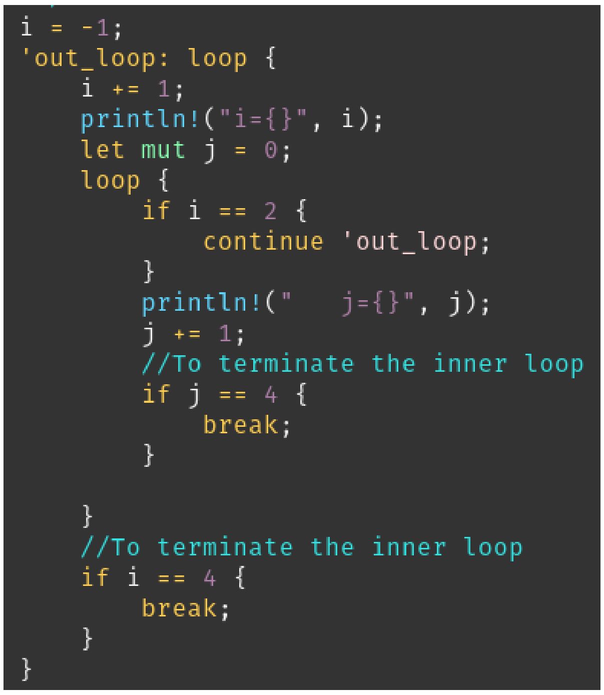
In Figure 19, values 1 and 3 will not be printed as these iterations will be terminated before reaching println! statement.



```rust
i = 0;
'out_loop: loop {
    println!("i={}", i);
    let mut j = 0;
    loop {
        println!("    j={}", j);
        if j == 4 {
            break 'out_loop;
        }
        j += 1;
    }
    i += 1;//unreachable, as the outer loop will be retminated in the inner loop.
}
```

*Figure 20: Labeled Break in Rust*

In Figure 20, the outer infinite loop will only iterate once as j will take value 4 in the inner infinite loop for sure in outer loops first iteration. Hence, we do not see any i values being printed other than 0. The inner infinite loop iterates 4 times and prints values 0, 1, 2, 3, and 4.



```
i = -1;
'out_loop: loop {
    i += 1;
    println!("i={}", i);
    let mut j = 0;
    loop {
        if i == 2 {
            continue 'out_loop;
        }
        println!("    j={}", j);
        j += 1;
        //To terminate the inner loop
        if j == 4 {
            break;
        }

    }
    //To terminate the inner loop
    if i == 4 {
        break;
    }
}
```

*Figure 21: Labeled Continue in Rust*

In Figure 21, no j value will be printed for i == 2, as the iteration of the outer loop will be terminated before the iteration of inner loop reaches to println! statement.

## 2) Readability&Writability

### Dart-Javascript:

There is not much to say about Dart's and Javascript's readability and writability. They have quite a standard syntax both for break/continue statements and labels, and it works well.
It is possible to write "break" and "continue" statements in both languages. I don't think their implementation hurts readability: When you look at the code you do not get lost.

### Lua:

Lua is not writable when it comes to unconditional exits. You need to find a workaround every time you want to use "continue", as "continue" does not exist in Lua. The developers of Lua state that **"continue was only one of a number of possible new control flow mechanisms"**[1]. While their explanation is understandable, their design choice harms writability. It also harms readability, because when you look at the code you wouldn't understand what is going on at a moment.

### PHP:

PHP has no problem in terms of writability, but readability is an issue, as numbers are used instead of labels. This is a problem, as you need to examine the code more carefully to understand which loop is going to be terminated.

### Python-Ruby:

Both Python and Ruby could be more writable in terms of unconditional and labeled exits. It is not possible to use labeled break and continue statements.

Guido van Rossum, the developer of Python programming language, refused PEP3136(PEP stands for Python Enhancement Proposal) where it was proposed to add labeled break and continue statements to the language[2]. Guido explaining the reason as follows: "**... I'm rejecting it on the basis that code so complicated to require this feature is very rare. In most cases there are existing work-arounds that produce clean code, for example using 'return'.**[3]" So, this design choice is based on inexistence of sufficient amount of real-world situations. His reasoning is sensible, but it harms writability.

I couldn't find any source stating Ruby's developers' choice, but I believe Ruby is more or less in the same situation as Python.

### Rust:

Rust is incredibly readable and writable, and I had no issue using unconditional and labeled exits in Rust. I feel like using the infinite loop feature of Rust combined with unconditional and labeled exits can be useful when there is a need for incredibly complex algorithms. I didn't try implementing such an algorithm in my code, this is just a guess.

It is necessary to use "**'**" in labels, which I think supports readability. I do not think this harms writability that much, if any.

## 3) Learning Strategy

For most of the languages, I tried to find code samples on the internet implementing asked questions in this assignment, as it was the fastest way to reach information. Of course, I didn't copy-paste any of them.

Making educated guesses about syntax was helpful. For example, I didn't need to make much research about how to use break/continue statements and labels in Dart and Javascript: I just guessed it, wrote it, and it worked.

To understand the design choices of languages, I tried to find official documents. For example, you can consider Guido van Rossum's rejection mail to PEP3136. Such documents coming directly from the developers of languages were incredibly helpful in understanding design choices. I tried to apply this strategy to Lua and Ruby programming languages as well.

I believe that reading the documentation of Rust and constantly failing&reading compiler errors and suggestions cycle is the best strategy to learn how any concept works on Rust. I applied this strategy. I believe it worked.

What I wrote above was my general learning strategy.

## References

1)https://www.tutorialspoint.com/why-does-lua-have-no-continue-statement
2)https://peps.python.org/pep-3136/
3)https://mail.python.org/pipermail/python-3000/2007-July/008663.html