
LECTURE 6.2: JUMPS EXPRESSIONS

BY LINA HAMMAD & AHMAD BARGHASH

In this part, you will learn to use break to terminate a loop. Also, you will also learn about break labels. In addition to, you will learn to use continue to skip the current iteration of a loop. Also, you will also learn about continue labels in this article.

JUMP EXPRESSIONS IN KOTLIN

- There are 3 structural jump expressions in Kotlin:
 1. Break: terminates the nearest enclosing loop.
 2. Continue: proceeds to the next step of the nearest enclosing loop.
 3. Return: by default returns from the nearest enclosing function or anonymous function.
- All of these expressions can be used as part of larger expressions.

KOTLIN BREAK EXPRESSION

- The **break statement** is used to terminate the loop immediately without evaluating the loop condition. As soon as the break statement is encountered inside a loop, the loop terminates immediately without executing the rest of the statements following break statement.
- In Kotlin there are two forms for break:
 1. Unlabeled break
 2. Labeled break

KOTLIN BREAK

- The **unlabeled break** statement terminates the loop immediately, and the control of the program moves to the next statement following the loop.
- It is almost always used with decision making statements (if...else Statement).
- The unlabeled form of break terminates the nearest enclosing loop.
- The syntax of a break statement is:
`break`

HOW BREAK WORKS?

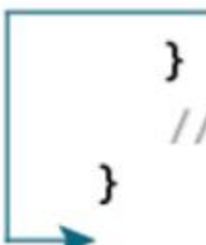
- It is almost always used with **if..else construct**. For example,

```
for (...) {  
    if (testExpression) {  
        break  
    }  
}
```

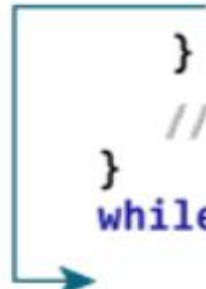
- If **testExpression** is evaluated to true, break is executed which terminates the for loop.

UNLABELED

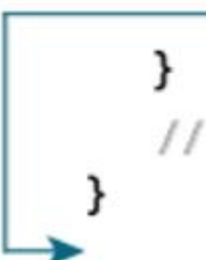
```
while (testExpression) {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
}
```



```
do {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
} while (testExpression)
```



```
for (iteration through iterator) {  
    // codes  
    if (condition to break) {  
        break  
    }  
    // codes  
}
```



EXAMPLE: KOTLIN BREAK

```
fun main() {  
    for (i in 1..10) {  
        if (i == 5) {  
            break  
        }  
        println(i)  
    }  
}
```

The output is:

1
2
3
4

- When the value of **i** is equal to 5, expression **i == 5** inside if is evaluated to true, and break is executed. This terminates the for loop.

EXAMPLE: CALCULATE SUM UNTIL USER ENTERS 0

- The program below calculates the sum of numbers entered by the user until user enters 0.

```
fun main() {  
    var sum = 0  
    var number: Int  
    while (true) {  
        print("Enter a number: ")  
        number = readLine()!!.toInt()  
        if (number == 0)  
            break  
        sum += number  
    }  
    print("sum = $sum")  
}
```

The output is:

```
Enter a number: 4  
Enter a number: 12  
Enter a number: 6  
Enter a number: -9  
Enter a number: 0  
sum = 13
```

- In the above program, the test expression of the while loop is always true.
- Here, the while loop runs until user enters 0. When user inputs 0, break is executed which terminates the while loop.

EXAMPLE: KOTLIN BREAK

```
fun main(args : Array<String>){  
    for(n in 1..10){  
        println("before break, Loop: $n")  
        if (n==5) {  
            println("I am terminating loop")  
            break  
        }  
    }  
}
```

The output is:

```
before break, Loop: 1  
before break, Loop: 2  
before break, Loop: 3  
before break, Loop: 4  
before break, Loop: 5  
I am terminating loop
```

- As you can observe in the output that as soon as the break is encountered the loop terminated.

KOTLIN BREAK EXAMPLE IN NESTED LOOP

- When break is used in the nested loop, it terminates the inner loop when it is encountered.

```
fun main(args : Array<String>){  
    for(ch in 'A'..'C'){  
        for (n in 1..4){  
            println("$ch and $n")  
            if(n==2)  
                break  
        }  
    }  
}
```

The output is:

```
A and 1  
A and 2  
B and 1  
B and 2  
C and 1  
C and 2
```

- As you can observe in the output that the outer loop never got terminated, however the inner loop got terminated 3 times.


KOTLIN LABELED BREAK

- The break label gives us more control over which loop is to be terminated when the break is encountered.
- The unlabeled form of break terminates the nearest enclosing loop. While labeled break terminate the desired loop (can be outer loop).
- The syntax of label is simple we just have to use any name followed by @ in front of the loop which we want to terminate, and the same name needs to be appended with the break keyword prefixed with @ as shown in the above example.

HOW LABELED BREAK WORKS?

- Label in Kotlin starts with an identifier which is followed by @.
- Here, test@ is a label marked at the outer while loop. Now, by using break with a label (break@test in this case), you can break the specific loop.

```
test@ while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition to break) {  
            break@test  
        }  
        // codes  
    }  
    // codes  
}
```



LABELED BREAK EXAMPLE

- Here, when `i == 2` expression is evaluated to true, `break@first` is executed which terminates the loop marked with label `first@`.

```
fun main() {  
    first@ for (i in 1..4) {  
        second@ for (j in 1..2) {  
            println("i = $i; j = $j")  
            if (i == 2)  
                break@first  
        }  
    }  
}
```

The output is:

`i = 1; j = 1`

`i = 1; j = 2`

`i = 2; j = 1`

LABELLED BREAK EXAMPLE

- Note: Since, break is used to terminate the innermost loop in this program, it is not necessary to use labeled break in this case.

```
fun main() {  
    first@ for (i in 1..4) {  
        second@ for (j in 1..2) {  
            println("i = $i; j = $j")  
            if (i == 2)  
                break@second  
        }  
    }  
}
```

The output is:

```
i = 1; j = 1  
i = 1; j = 2  
i = 2; j = 1  
i = 3; j = 1  
i = 3; j = 2  
i = 4; j = 1  
i = 4; j = 2
```

LABELED BREAK EXAMPLE

```
fun main(args : Array<String>){  
    myloop@ for(ch in 'A'..'C'){  
        for (n in 1..4){  
            println("$ch and $n")  
            if(n==2)  
                break@myloop  
        }  
    }  
}
```

The output is:

A and 1

A and 2

KOTLIN CONTINUE EXPRESSION

- The **continue** construct skips the current iteration of the loop and jumps the control to end of the loop for the next iteration. The continue is usually used with if else expression to skip the current iteration of the loop for a specified condition.
- In Kotlin there are two forms for continue:
 1. Continue construct.
 2. Continue Labels.

CONTINUE CONSTRUCT

- Continue skips the iteration but it is not capable of skipping the statements that are encountered before continue.
- As soon as continue is encountered, the control jumps to the end of the loop skipping the rest of the statements.

HOW CONTINUE WORKS?

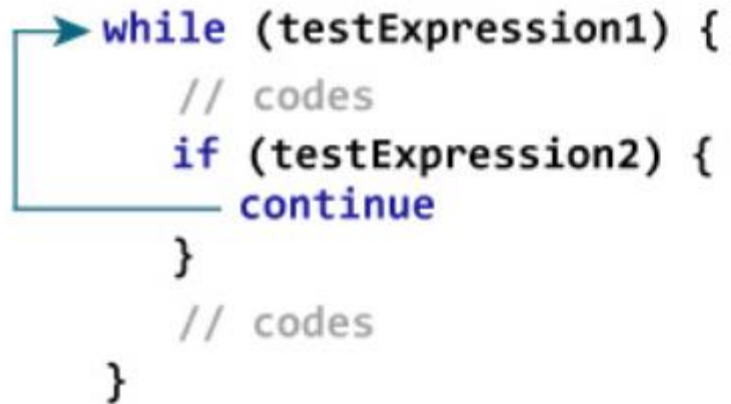
- It is almost always used with if...else construct. For example,

```
while (testExpression1) {  
    // codes1  
    if (testExpression2) {  
        continue  
    }  
    // codes2  
}
```

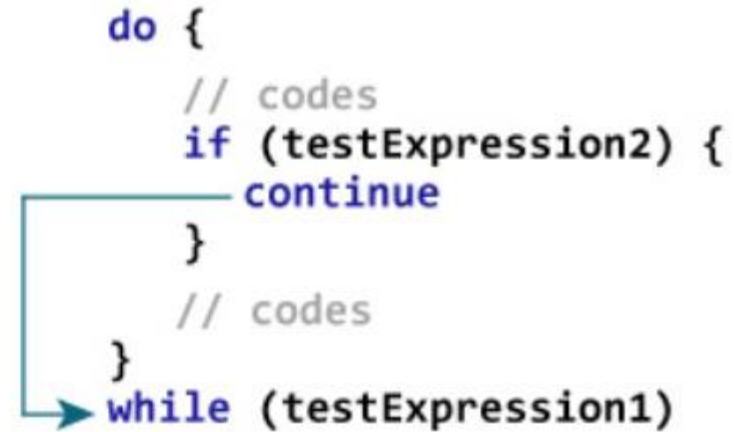
- If the **testExpression2** is evaluated to **true**, continue is executed which skips all the codes inside while loop after it for that iteration.

HOW CONTINUE WORKS?

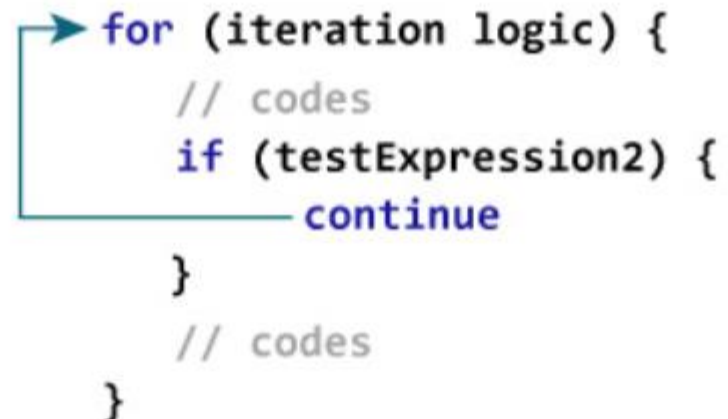
```
→ while (testExpression1) {  
    // codes  
    if (testExpression2) {  
        continue  
    }  
    // codes  
}
```



```
do {  
    // codes  
    if (testExpression2) {  
        continue  
    }  
    // codes  
} → while (testExpression1)
```



```
→ for (iteration logic) {  
    // codes  
    if (testExpression2) {  
        continue  
    }  
    // codes  
}
```



EXAMPLE: KOTLIN CONTINUE

```
fun main(args: Array<String>) {  
    for (i in 1..5) {  
        println("$i Always printed.")  
        if (i > 1 && i < 5) {  
            continue  
        }  
        println("$i Not always printed.")  
    }  
}
```

The output is:

```
1 Always printed.  
1 Not always printed.  
2 Always printed.  
3 Always printed.  
4 Always printed.  
5 Always printed.  
5 Not always printed.
```

- When the value of **i** is greater than **1** and less than **5**, `continue` is executed, which skips the execution of the following statement: `println("$i Always printed.")`
- However, the statement `println("$i Not always printed.")` is executed in each iteration of the loop because this statement exists before the `continue` construct.

EXAMPLE: KOTLIN CONTINUE

```
fun main(args : Array<String>){  
    for (n in 1..5){  
        println("before continue, Loop: $n")  
        if(n==2 || n==4)  
            continue  
  
        println("after continue, Loop: $n")  
    }  
}
```

The output is:

```
before continue, Loop: 1  
after continue, Loop: 1  
before continue, Loop: 2  
before continue, Loop: 3  
after continue, Loop: 3  
before continue, Loop: 4  
before continue, Loop: 5  
after continue, Loop: 5
```

- As you can see in the output that `println("before continue, Loop: $n")` statement didn't execute for the loop iterations `n==2` and `n==4` because on these iterations we have used the `continue` before this statement which skipped the iteration for these values of `n`.
- **However** you can observe that `println("after continue, Loop: $n")` statement executed on every iteration because it is executed before `Continue` is encountered.

EXAMPLE: CALCULATE SUM OF POSITIVE NUMBERS ONLY

- The program below calculates the sum of maximum of 6 positive numbers entered by the user. If the user enters negative number or zero, it is skipped from calculation.

```
fun main(args: Array<String>) {  
    var number: Int  
    var sum = 0  
    for (i in 1..6) {  
        print("Enter an integer: ")  
        number = readLine()!!.toInt()  
        if (number <= 0)  
            continue  
  
        sum += number  
    }  
    println("sum = $sum")  
}
```

The output is:

```
Enter an integer: 4  
Enter an integer: 5  
Enter an integer: -50  
Enter an integer: 10  
Enter an integer: 0  
Enter an integer: 12  
sum = 31
```

EXAMPLE: KOTLIN CONTINUE

```
fun main(args : Array<String>){  
    for (n in 1..10){  
        if(n%2!=0)  
            continue  
  
        println("$n")  
    }  
}
```

The output is:

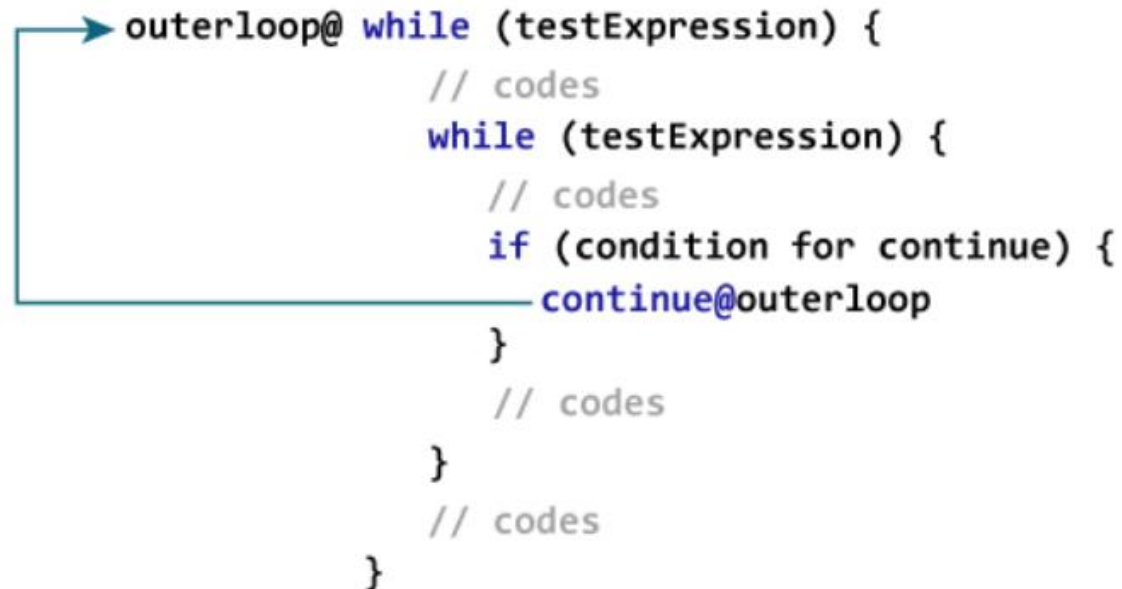
```
2  
4  
6  
8  
10
```

KOTLIN LABELED CONTINUE

- Continue construct is unlabeled form of continue, which skips current iteration of the nearest enclosing loop. While, the continue labels used to skip the iteration of the **desired loop** (can be outer loop).
- The labels in labeled continue are cool basically they give us more control when we are dealing with nested loops.

HOW LABELED CONTINUE WORKS?

- Label in Kotlin starts with an identifier which is followed by @.
- Here, `outerloop@` is a label marked at outer while loop. Now, by using `continue` with the label (`continue@outerloop` in this case), you can skip the execution of codes of the specific loop for that iteration.



```
outerloop@ while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition for continue) {  
            continue@outerloop  
        }  
        // codes  
    }  
    // codes  
}
```

EXAMPLE: LABELED CONTINUE

```
fun main(args: Array<String>) {  
    here@ for (i in 1..5) {  
        for (j in 1..4) {  
            if (i == 3 || j == 2)  
                continue@here  
            println("i = $i; j = $j")  
        }  
    }  
}
```

The output is:

```
i = 1; j = 1  
i = 2; j = 1  
i = 4; j = 1  
i = 5; j = 1
```

- The use of labeled continue is often discouraged as it makes your code hard to understand. If you are in a situation where you have to use labeled continue, refactor your code and try to solve it in a different way to make it more readable.

CONTINUE LABEL VS CONTINUE CONSTRUCT

- In this example we have a nested for loop and we are not using label. When we do not use labels we do not have any control and as soon as the continue is encountered the current iteration is skipped for the inner loop.

```
fun main(args : Array<String>){  
    for (x in 'A'..'D'){  
        for (n in 1..4){  
            if (n==2 || n==4)  
                continue  
  
            println("$x and $n")  
        }  
    }  
}
```

The output is:

```
A and 1  
A and 3  
B and 1  
B and 3  
C and 1  
C and 3  
D and 1  
D and 3
```

CONTINUE LABEL VS CONTINUE CONSTRUCT

- The above example output that for n value 2 and 4 the iteration is skipped for the inner loop. If we want to skip the iteration for the outer for loop, we can do so with the help of continue labels.

```
fun main(args : Array<String>){  
    myloop@ for (x in 'A'..'D'){  
        for (n in 1..4){  
            if (n==2||n==4)  
                continue@myloop  
            println("$x and $n")  
        }  
    }  
}
```

The output is:

```
A and 1  
B and 1  
C and 1  
D and 1
```

RETURN EXPRESSION

- We will discuss the return jump expression in detail when talk about functions.



LECTURE 7: PRACTICAL SESSION

BY LINA HAMMAD & AHMAD BARGHASH

In this part, you will practice the loops, if and when statement , break and continue.

ASSIGNMENT I

- How to find the greatest common divisor. The two numbers should be a user input.

	U	V
■ Step 1	150	35
■ Step 2	35	10
■ Step 3	10	5
■ Step 4	5	0

ASSIGNMENT 2

- Write a program that inputs 3 integers ordered from minimum to maximum then find the sum of even numbers between the first two and the sum of odd numbers between the second two.

ASSIGNMENT 3

- Write a program that finds and prints the prime numbers between 1 and a number entered by the user.

ASSIGNMENT 4

- Write a Kotlin program that generates the following output using one pair of nested for loops.

**

*

ASSIGNMENT 5

- Write a Kotlin program that generates the following output using one pair of nested for loops.

*

**
