Bünyamin AKTAŞ
15/05/2020

ASSIGNMENT - 5

Q1 **The answer is D.**

A – A *do-while* loop executes its code statement or code block first, before checking its *boolean* expression. Since this loop evaluates the boolean expression at the end it is not best for its boolean condition.

B - Traditional *for* loop has a boolean expression in its construct but it is known for its index variable not for its boolean condition.

C - Enhanced *for* loop doesn't even have a boolean condition.

D - *while* loop has only one expression in its construct. It is boolean expression and it is not evaluated at the end as in *do-while* loop. **So this loop is best known for its boolean condition.**

Q2 **The answer is B.**

A - A *do-while* loop executes its code statement or code block first, before checking its *boolean* expression. Then it would check its *boolean* expression and may execute its block statement according to the return value of the *boolean* expression. And as long as it returns *true*, the loop executes. It does not need to use an index or operator.

So that *do-while* loop is known for its boolean expression.

B - *for*(traditional) loop takes three statements in its definition: initialization statement, which define variable(s); a boolean expression which will determine whether the block statement of the loop will execute or not; and an update statement which will be run after each evaluation of the loop statements.

Here the important thing for the question is the initialization statement that the variable defined in that statement could be used as a counter/index in arrays or collections. **So this traditional for loop is best known for using an index.**

C - *for-each* loop is an enhanced way of traditional *for* loop. In this way, *for* definition takes a variable of type matching with the data types of members of a collection. For example if we have an array or collection composed of String values, we declare a String value in *for-each* loop and write the reference name of the collection after a colon. That's it. At runtime, when Java encounters this loop, it will transform it to a traditional *for* loop and evaluate it until the end of the array/collection.

But as already been understood, this enhanced *for* loop does not have any index to iterate over arrays and collections.

D - *while* loops only checks a boolean expression by requirement that they may not

possess an index variable and even if they would have one, it would be more complex to implement comparing to traditional and enhanced *for* loops. So *while* loop also cannot be the one best known for using index and counters.

while loop is also known for its boolean expression same as *do-while* loop.

Q3 **The answer is A.**

Since *do-while* loops checks the boolean expression at the end of a first evaluation of the code block, *do-while* loop is guaranteed to execute the body at least once. Because the code firstly enters the code block and runs it and later checks the boolean expression.

Q4 **The answer is C.**

As already explained in **Q2** *for-each* loops only initialize a variable whose data type matches the type of the members of the array to be looped through. So Java will loop through the array's elements automatically although you don't express the index explicitly.

Q5 **The answer is B.**

There is no keyword called *end* or *skip* in Java.

break keyword is used to terminate a loop - if not expressed which loop, the nearest one - and exits the loop's body terminating the loop. So this statement does not proceed to the next iteration.

continue keyword is used to end the current iteration of the loop - again if not stated which loop, the nearest one - and jump to the next iteration of the loop.

Q6 **The answer is A.**

As stated in the previous question there is no keyword called *end* or *skip* in Java.

continue keyword is used to end the current iteration of the loop - if not stated which loop, the nearest one - and jump to the next iteration of the loop.

break keyword is used to terminate a loop - if not stated which loop, the nearest one - and exits the loop's body to continue executing after the loop.

Q7 **The answer is B.**

while and *do-while* loops have one segment in parentheses which is a boolean expression which determines if the loop body will be executed.

for-each loop has two segments. First one is an initialization segment that declares a variable of type which matches the type of the members of the array which will be expressed in the second statement.

Traditional *for* loop has three segments within parentheses; an initialization segment, a boolean expression and an update statement, respectively. The loop initialize a variable and uses it in the boolean expression and/or in the loop body. If the loop is ever executed then the update statement is run after each iteration of the loop.

Q8 **The answer is C.**

As explained in **Q2**, a traditional *for* loop takes a variable to use as index and updates it after each iteration of the loop. This variable can be started either from 0, which indicates the first index of an array, or from the last index of the array, which can be determined by *[array.length - 1]* since arrays are zero based and their last index equals to their length mines 1. Then this index can either be incremented or decremented in the update statement.

As a result a traditional *for* loop can iterate through an array starting either from index 0 or from the end.

Q9 **The answer is A.**

A *for-each* loop has two segments; one statement that declares a variable whose type matches the type of the members of the collection to be iterated through, and the other one is the collection's reference name. Then, this loop starts to iterate through the collection starting from index 0. There is no syntax to declare an index number so that we don't have ability to iterate this collection starting from the end. **So only the statement I is true.**

Q10 **The answer is A.**

As explained in **Q2 - A**, a *do-while* loop executes the body at first once and then checks the *boolean* expression. The structure of a *do-while* statement:

```
do {
        // body
} while(booleanExpression);
```

So the answer is A.

Q11 **The answer is B.**

A *while* loop takes one segment which is a boolean expression and runs according to the return value of this boolean expression. In this code an integer value of type *int*, is attempted to be used as a boolean value in *while* loop within parentheses. Using *int* values as boolean is not valid in Java. **So this code does not compile.**

We can try to fix the code by replacing the statement within the parentheses with *singer <= 0* in order to have the loop run at least once and not to be an infinite loop. In that situation value of *singer* would be incremented by 1 and the original value of the *singer* would be printed.

Q12 **The answer is B.**

In this question, *Arrays.asList()* method is used. This method takes an array as an argument and converts it to a List. But the parameter used in *Arrays.asList()* method is varargs. So that we can pass multiple individual arguments, variables to the method. The method will take them and consider them as an array.

So in the code an array that has 2 elements, is created. The elements are *can* and *cup*, respectively.

Then a traditional *for* loop is implemented, which iterates through our list starting from the end. So the output will be reverse order of the array created at the first place which is *"cup, can"*. Since a comma is printed in each iteration **the output will be "cup,can,"**.

Q13 **The answer is A.**

Similarly to the previous question a List *bottles* is created by converting an array by the method *Arrays.asList()*. Then a traditional *for* loop is implemented that is missing the third segment which is update statement. So under normal circumstances this loop can be considered as an infinite loop which means a loop that never ends. But a *break* statement is put in the loop which will terminate the loop after the first iteration.

So in the first and the last iteration of the loop, the first element of the list will be printed followed by a comma. After the loop another word "end" will be printed **so the output will be "glass,end".**

Q14 **The answer is A.**

A String variable *letters* is created with no characters. Then a *while* loop is implemented checking the length of the value of the variable. Until the variable has a value of length 2, the loop will be iterated and will add the character *a* to the value. So in the second iteration, the second letter *a* will be concatenated to the variable and the length of the variable will be 2. So the third execution of the boolean statement which checks the length of the variable will return *false*, which will terminate the loop.

So the last value of the variable, **which is "aa", will be printed.**

Q15 **The answer is D.**

In the *for* loop a variable is set to the length of the array of main() method. Then it is checked if it equals to or greater than 0, which is going to return *true* each time because the length of

the main() method array will be 0 at least, even if no argument is passed. Then the update statement is set as an incrementer of the variable *i*, which will cause to the boolean expression of the loop will return *true* each time concluding in an infinite loop. So the code will never be terminated and is always going to print “args”.

Q16 **The answer is B.**

Two class variables are created. First is an int variable. Second is a String array. The elements of the array are initialized explicitly but the first variable *count* is not initialized. Since these are class variables, *count* will default to value 0.

Then a *while* loop is implemented in main() method which will iterate through the String array. The update of the index variable *count* is provided in *if-then* statement. The *if-then* statement checks if the element in the current index has a length less than 8. If it has, the loop is terminated by a *break* statement.

Finally the value of the *count* will be printed which will be the index of the first element that has a length less than 8, plus 1. Since the first less than 8 length element of the array is in the index 1, the output will be 2.

Q17 **The answer is C.**

A local variable, *count*, is declared and initialized in a nested *do-while* loop. The inner loop increments *count* until it reaches to 2 and then exits. Then outer loop is terminated by *break* statement. The fault is here that the local variable, *count*, is attempted to be used outside of its scope. **So that the code does not compile since it cannot reach count.**

If we make *count* reachable by taking it out from the loop, then the code would compile and print 2.

Q18 **The answer is D.**

All segments of traditional *for* loop are optional that anyone of them may be missed. For example if we do not want to use any of them then we can simply write:

for (; ;) {}

So the answer is D.

Q19 **The answer is C.**

while and *do-while* loops take only segment which is a boolean expression. If we ensure that this boolean expression returns always *true* then the loop will never terminate.

Similarly, traditional *for* loop takes three segment, one of which is a boolean expression.

We can ensure that this boolean statement will always return *true* or we can simply remove it since it is optional in traditional *for* loop.

Enhanced *for* loop has one job, iterating over an array or a collection. When it comes to the end of the array/collection, it terminates. We may think to expand the collection during the loop ensuring the loop never terminating but it is not allowed in Java. When it is attempted the code compiles but a ConcurrentModificationException is thrown out at runtime.

So three of the loop types allow us to write a code that creates an infinite loop.

Q20 **The answer is A.**

Arrays.asList() method has a varargs parameter which takes an array or multiple individual arguments, converting them into array, as argument. Then it converts this array to a List.

Traditional *for* loop in this code iterates through the list created from an array composed of [can, cup], starting from the index 0. **So the output will be "can,cup,"**.

Q21 **The answer is B.**

Since *do-while* loop executes the body first, once and then checks the boolean expression the body will be executed first. It will print "helium". Then the boolean expression will be checked and since it is the value *false* the loop will be terminated immediately.

So the answer is the code outputs "helium" once.

Q22 **The answer is B.**

This traditional *for* loop iterates through an array called *fun* and prints its members. The questions asks us the equivalent of this loop.

A - This loop implementation has such a statement: *String f = fun*. And then it tries to use this variable *f* to print. Firstly, enhanced *for* loop does not have any assignment statement so that this loop cannot be enhanced *for* loop. We can think this as a traditional loop, but then it is missing the optional segment markers, two semi-colons. Even if this syntax was true, then the code would still not compile because a String array is attempted to assign to a String variable which is not valid in Java. So this code never compiles.

B - This loop's syntax is same as enhanced *for* loop's syntax. So Java will take the first member of the array *fun* and will assign it to the variable *f* declared before the colon. Then it will iterate to the next member of the array and assign it to the *f*, again. So this enhanced *for* loop will print every member of the array starting from the index 0.

C - This implementation has a weird statement that does attempt to assign our String array *fun* to the data type *String*, which is totally not allowed. Then it tries to print something called "*it*" which does not exist neither in the code nor in Java. So this code does not compiles.

D - Since one of the options is true, this option is incorrect.

Q23 **The answer is C.**

The diagram shows that the inner loop will print the value of *ch* and then terminates immediately. So the inner loop has no significance actually due to that it will execute only once in every iteration of the outer loop. So the statement we need must ensure that the inner loop terminates and control pass to the outer loop.

Here we have something new, labels which are optional in Java. Optional labels can be used before loops in order to have a pointer to be able to reach to these loops easily. The labels are written in upper-case letters traditionally but it is not required. So lower-case labels are valid, too. Here the label *letters* points to the outer loop while *numbers* label points to the inner loop.

break; statement terminates the nearest loop when it is used standalone. But if a label is written after a *break* keyword, then it will terminate the loop the label points to. If we use *break* statement standalone after println it will terminate the nearest loop which is the inner loop. So *break;* does what we want.

break letters; statement will terminate the outer loop which is not desired and also will result in printing only the first letter, “a”. The diagram shows a code printing all lower-case letters of the alphabet.

break numbers; statement terminates the inner loop since *numbers* points to that loop.

So two of the statements follow the code flow in the diagram.

Q24 **The answer is B.**

Since the question uses the same diagram as the previous question, explanations about the diagram has been skipped.

Here the label *letters* points to the outer loop while *numbers* label points to the inner loop.

continue; statement only ends the current iteration of the nearest loop while *break* statement terminates the loop. Since diagram requires the inner loop to be terminated and *continue*; statement does only end the current iteration of the nearest loop which is the inner loop, this statement is wrong.

continue letters; statement will end the current iteration of the outer loop which will cause the inner loop to be terminated automatically. This does what is required by the diagram.

continue numbers; statement will end the loop the label *numbers* points, which is the inner loop. So this statement actually does the same thing as *continue;* statement does. So this statement does also not fit in requirements.

Finally, only one of the statements does fit the flow of the diagram.

Q25 **The answer is C.**

The variable *singer* is equated to 0. The *while* loop checks if *singer* is greater than 0. Since that check will return *false* value the loop will never execute the body. **So the code outputs nothing.**

Q26 **The answer is C.**

Enhanced *for* loop has two segments, first one of which is a variable declaration whose data type is the same as the data type of the members of the collection that is to be iterated through and that is the second segment of the loop.

In this code the variable's type is Object which means that the array or the collection called *taxis* must contain objects.

ArrayList<Integer> means a List composed of Integer object which means it suits the requirements.

int[] is an array composed of *int* primitive values. At first glance this seems not working. But here *autoboxing* comes in. Since Java can convert an *int* to its wrapper object *Integer* and *Integer* is an *Object*, int[] will also work.

StringBuilder is an object type consisting of String values. But the problem here is that enhanced *for* loop can only iterate over objects that implements *java.lang.Iterable* class. **Since StringBuilder does not implement Itrable class, this type will cause the compiler give an error.**

Q27 **The answer is B.**

A boolean variable *balloonInflated* is initialized in the code to the value *false*. Then a *do-while* loop is implemented. Without checking the boolean expression the code begins to execute. *If-then* statement checks if *balloonInflated* is *false* and it returns *true*. So that the body of the *if-then* statement begins to work which makes *balloonInflated true* and prints "inflate-". Then the code exits *if-then* statement and the boolean expression of the *do-while* loop, which is checking if ! *balloonInflated* is *false*, is evaluated. It returns *false*, since the value of *balloonInflated* is changed to *true* in *if-then* statement. The loop terminates and the code prints "done". **The output is "inflate-done".**

Q28 **The answer is D.**

The loop implemented in the code checks if the String variable *letters* has 3 characters. Since it is initialized zero length it will return *false*. Then the loop begins to execute expanding the *letters* by 2 each time. So the length of the *letters* will never be 3. **The loop will never terminate.**

Q29 **The answer is B.**

A traditional *for* loop optionally initialize a variable at first, then evaluates a boolean expression and if it returns *true* then execute the body. After executing the body it will evaluate the update statement and again evaluate the boolean expression.

So in similar to the explanation above, the order of the three expressions is: ***initiliazation expression, boolean conditional, update statement.***

Q30 **The answer is B.**

The code creates an *int* variable and a List consisting of Character objects which is a wrapper class for *char* primitive data type. Then it goes in a *do-while* loop. This loop adds one character to the List object named *chars* and subtract the final length of the list from *count*. So it will be as;

(the value of count) - (current size of the list) = (new value of count)

At first the length of the *chars* is 1 so the count will be 10 - 1 = 9. Then the length of the *chars* will be 2 so the *count* will be 9 - 2 = 7. It will continue as: 7 - 3 = 4 and 4 - 4 = 0.

When *count* is 0 the loop will terminate. **And the size of the list is 4 as explained above.**

Q31 **The answer is A.**

Here we have a *while* loop inside a *for* loop that is starting from 10 and goes until the variable *i* equals to 0. But the *while* loop inside the *for* loop subtract 3 until *i* is not greater than 3. So it will subtract it as;

10 - 3 = 7, 7 - 3 = 4, 4 - 3 = 1, and when *i* is 1 the *while* loop will terminate and *k* will be incremented by 1. Then the update statement of the *for* loop will decrement the *i* by 1 and since it is now 0, the *for* loop will terminate, too. **The output will be the final value of *k*, which is 1.**

Q32 **The answer is D.**

As explained in **Q22**, there is no syntax such as = in enhanced for loop. *String f fun* is not also a valid statement. So **the options A** and **C** does even not compile. Also the argument "*it*" passed to the *println*, does not exist as anything.

The true syntax of the enhanced *for* loop as in **the option B**, will iterate through the array starting from the index 0. But our traditional *for* loop written in the question iterates the array starting from the end.

So none of the options are true.

Q33 **The answer is C.**

A List is initialized by *Arrays.asList(varargs)* method firstly. Since the method has a varargs parameter, individual variables can be passed into the method as argument. These variables will be put into an array by Java and the method will convert this array into a List.

Then a traditional *for* loop is implemented to iterate over this List but this code has a compiler error that *break* statement is actually outside of the loop since there is no brackets encapsulating it for the *for* loop. And Java does not allow *break* statements to be used outside of a loop or a switch. **So this code does not compile this way.**

We can fix the code by taking the two statement after the loop definition in brackets. If we do this, then the loop would print the first element of the List *bottles* and then would be terminated immediately by the *break* statement.

Finally the last *println* method would print "end" and the program would terminate.

Q34 **The answer is C.**

Here we have two String arrays that are in different sizes.

And also we have a *for* loop which initializes two variables in initialization segment which is allowed as long as they are of same data type and separated by a comma. And this is valid for the boolean and update statements, too. The error is there that the update statements are separated by a semi-colon instead of a comma which is illegal in Java. **So this causes the code not to compile.**

But if we attempt to fix the code replacing that semi-colon with a comma then the loop would iterate as long as the length of the shortest array which here is *times*. So the first two elements of the two arrays would be printed which are "Downtown Day-Uptown Night-".

Q35 **The answer is D.**

The code compiles because there is no syntactic error.

The parameter of the main() method acts like a varargs that it takes individual arguments from the command line and puts them into an array. So main() method's argument *args* is created three-sized in this code which has the following elements; [September, 3, 1940].

The problem is that arrays are zero based so that the last index of arrays is their length - 1. But here the code attempts to reach the index numbered same as the length of the array which actually does not exist. **So this code will throw an ArrayIndexOutOfBoundsException at runtime.**

If we fix the code replacing *args.length* with *args.length - 1* then it would execute smoothly. The output will be the printing the elements of the *args* in reverse order.

Q36 **The answer is B.**

This code has a String variable that is initialized to *null* object. The *while* loop implemented just after that has a boolean expression checking if the variable equals to *null*. Since it is *true* at first, the body begins to execute. There are two statement after the *while* loop definition but since they are not within brackets only the first one will be executed as the *while* loop's body. That first one assigns a String value to the String variable which causes the loop terminate because the variable is no longer equal to the *null*.

The second statement after the *while* loop prints the value of the String variable which is "Shoelace".

Q37 **The answer is C.**

In this question an optional label is used which is a feature of Java in order to be able to point to a certain place so that we can reach that place easily afterwards. They are defined traditionally all with uppercase letters but this is not required.

break loop; statement terminates the loop that optional label *loop* points to. But if we remove the lines 25 and 28 then we happen to remove the *while* loop which will cause a compiler error since the *break* statement cannot be used outside of a loop or a switch. **So the code no longer compiles.**

Q38 **The answer is C.**

The code has two static variables, one of which is an *int* type variable named *count* and the other one is a String array. The *while* loop implemented in main() method executes the body if count is less than the length of the array. And an *if-then* statement inside of the loop checks if the member of the array which is in the index *count++* is less than 8. If it is, *if-then* statement executes its body which contains one statement which is *continue*. *continue* statement end the current iteration of the loop and jump to the next iteration. But even if this statement was not used, the loop would jump to next iteration since there is no more statements after *if-then* statement. So that *if-then* statement here is unnecessary.

However, the boolean expression used in *if-then* statement has post-unary operator with *count* which increments it by 1 in each iteration. Since the String array has 4 elements, the loop will be iterated 4 times which will make *count* 4. **The output will be 4.**

Q39 **The answer is C.**

In Java no other types than *boolean* or *Boolean* is allowed to be used as boolean expressions. In this code the *StringBuilder* object *builder* is attempted to be used as a boolean variable. **So the code does not compile.**

We can try to fix the code using equality operator checking if the *builder* equals to equals to *null*. Then the code can compile and run.

Q40 **The answer is A.**

This code has a nested *do-while* loop. Inner loop increments the *count* since it comes to be 2. Then there is a *break* statement. A *break* statement terminates the nearest loop immediately. The nearest loop to this *break* statement is the outer loop because *break* is not within the inner loop's brackets.

By the way the outer loop's boolean expression is *true* itself which means this loop is an infinite loop if *break* was not used.

Since *break* immediately terminated the outer loop the value of *count* is 2.

Q41 **The answer is C.**

The code has a nested *while* loop. All loops are infinite by nature because the boolean value *true* is used as boolean expression. Since the value is always *true* the loops will never terminate. So we must terminate the outest loop to terminate the nested loop.

We can use *break* statement which terminates loops when used. But *break* statement terminates the nearest loop if used alone. So we must explicitly state the outest loop to the *break*. This can be accomplished with optional labels. In this code we have two labels, *t* and *f*, pointing to the outer and the inner loop, respectively. So we need to use the outer loop's label with *break* statement in order to terminate the loops. **So the answer is *break t;*.**

Q42 **The answer is B.**

This question is the fixed way of the Q34. Traditional for loop accepts more than o statements in initialization and update segments as long as they are separated by a comma. Additionally the variables initialized in initialization segment must be of same data type. This code snippet meets all these requirements.

So the code runs and prints the elements of the two arrays until one of them's index reaches to the last. ***Since the shortest one has 2 elements, 2 elements of both arrays will be printed which is Downtown Day-Uptown Night-.***

Q43 **The answer is B.**

The answer has a List created by *Arrays.asList()* method. This method takes individual arguments, puts them into an array and then converts that array into a List, since the parameter of the method is varargs.

Then a nested *for-each* loop is implemented that each loop iterating over the same List. Since the List has 2 elements, each loop will iterate two times printing 2 elements line by line. In the first iteration the inner loop will print 2 lines and in the second iteration the inner loop will print 2 lines again. **So in total, 4 line will be printed.**

Q44 **The answer is B.**

The traditional *for* loop executes first the initialization segment. Then it evaluates the boolean segment and if it returns *true* then continue to execute the body of the loop. After the executing the body, it executes the update segment and then again evaluates the boolean segment.

In this example, the boolean segment *beta* always returns *false* which means the body will never be executed and the loop will be immediately terminated.

So the flow of execution in this loop can be described as ***alpha, beta.***

Q45 **The answer is B.**

A traditional *for* loop terminates whenever the boolean segment returns *false*, unless a *break* statement is used. So if the loop body is run once, this means the boolean segment returned *true* at first and *false* at the second evaluation. So according to the explanation above (first paragraph), **the flow of the loop can be described as *alpha, beta, delta, gamma, beta.***

Q46 **The answer is C.**

All of the loop types except *do-while* firstly evaluates the boolean segment and if it returns *true*, executes the body. But *do-while* loop executes the body first then evaluates the boolean segment so that it iterates one more time than the others.

All of the loops in options increment the *k* 5 times so that they iterate 5 times. **Since *do-while* iterated once before evaluating and incrementing *k*, it iterated 6 times.**

Q47 **The answer is D.**

This code snippet has a semicolon right after the *while* loop definition which is actually equivalent to a *while* loop with an empty body. So the next statements are not related to the *while* loop. This means if the statement within the parentheses returns *true* the loop will start to iterate and since it does not have any statement that could update the *tie* that is the variable used in boolean expression, it will iterate forever. This creates an infinite loop. **So the output is not any of the options A, B and C.**

If we would remove that semicolon then *println* method would print the latest value of *tie* which is “shoelace”.

Q48 **The answer is C.**

Java has some keywords that are reserved and blocked from being used in names of variable/label/class/method. *for* is one of them so that it cannot be used as a label name. The code does not compile.

If we change the label name then inner *while* loop would decrement result to 5 and the

break statement would terminate the outer loop. **The output would be 5.**

Q49 **The answer is D.**

A boolean variable *baloonInflated* is initialized in the code to the value *false*. Then a *do-while* loop is implemented. Without checking the boolean expression the code begins to execute. *If-then* statement checks if *baloonInflated* is *false* and returns *true*. So that the body of the *if-then* statement begins to work which makes *baloonInflated true* and prints "inflate-". Then the code exits *if-then* statement and the boolean expression of the *do-while* loop, which is checking if *baloonInflated* is *false*, is evaluated. It returns *true*, since the value of *baloonInflated* is changed to *true* in *if-then* statement. So the loop iterates again and checks if *!baloonInflated* is *true* and since it return *false* the *if-then* statement is skipped. And since there is no other statement in the loop to change the value of *baloonInflated* and it returns always *true* the loop iterates forever. **This is an infinite loop.**

Q50 **The answer is B.**

The semicolon in a traditional *for* loop definition is a separator of segments. So a semicolon cannot be used in initialization segment.

In order to initialize more than one variables in initialization segment we must separate them with a comma and make them same data type. In Java a semicolon also means a statement that one statement can only have one data type word so that **the option D** is also wrong since it uses *int* keyword twice.

The only correct syntax is in option B.