Bünyamin AKTAŞ
13/06/2020

ASSIGNMENT – 9

Q1 **The answer is C.**

A – *StringBuilder* does not support multiple threads.

B – Equality operator == can only compare references. *StringBuilder* has no privilege about this.

C – Since *StringBuilder* is mutable and returns always itself after appending or removing values to/from itself unlike *String*, it guarantees that less memory is used.

D – Both *String* and *StringBuilder* support different languages and encodings so that this statement cannot be the reason for using *StringBuilder* instead of *String*.

Q2 **The answer is D.**

A – *String* variables can be created without calling constructor. When String variables are created this way, they are assigned values from String pool which helps to use less memory.

B – String pool helps to use less memory for that it allows reusing the values for multiple variables.

C – *String* class is defined final.

D – *String* is immutable. A *String* variable cannot be changed. When a *String* variable is attempted to be changed, a new *String* variable is created which consumes more memory so that *String* pool and *StringBuilder* was invented.

Q3 **The answer is D.**

Both *String* and *StringBuilder* allow method chaining, which is invoking methods on an object in one line, when creating an object. So the expressions in all options are valid.

A – “clown” is appended to a recently created *StringBuilder* object without any value. So the last value is “clown”.

B – A new *StringBuilder* object is created with a value “clown”.

C – A new *StringBuilder* object is created with a value “cl” and then “own” is inserted starting from the index 2, which makes the last value of the object “clown”.

So, all of the expressions create a *StringBuilder* object with the same value.

Q4 **The answer is B.**

StringBuilder is mutable, which means that the data stored in it can be changed withing methods. *append()* method takes one argument and appends it to the current string value.

A new *StringBuilder* object is created with the value “333”. Then *append()* methods append “ 806” and “ 1601”, respectively. **This makes the last value of *teams* “333 806 1601”.**

Q5 **The answer is B.**

Objects can be referenced by their own reference type, or by their superclass type, or by the interface type that they implement.

List is an interface. Interfaces cannot be instantiated directly so that *List* cannot be put in the blank.

ArrayList implements *List* so that *ArrayList* objects can be assigned to *List* references which means *ArrayList* can be put in the blank.

Object must be implementing *List* in order to be able to be assigned to *List.* Since it does not implement *List,* it cannot be put in the blank.

As a result, only one type can be used in the blank.

Q6 **The answer is C.**

ArrayList is an improved structure of an array. It is dynamic and can expand in need. This code snippet creates an *ArrayList* object without specifying any capacity. In this case the object is created with a default size. However, if it does not have any elements, it would throw an extension when a non-existing element is attempted to be reached.

This code adds 3 elements one-by-one making the element number of *tools,* 3.

println() method attempts to access the second element of the *ArrayList* object which is “nail”.

Q7 **The answer is C.**

Java performs assignments right to left. This means, in order to initialize the variable which is at the left, the expression at the right must be evaluated to the end.

This code snippet attempts to use a variable in its own assignment statement which is not allowed since it may not be initialized at all. **So this code does not compile.**

If the *insert()* method was invoked after the object *sb* is initialized, the code would compile and insert “robots” right after the end of the value “radical” since the length of the string is

given as the inserting point index. “r” of “robots” would come at the index *sb.length()*, which is 7. In that case the value of *sb* would be “radicalrobots”.

Q8 **The answer is A.**

This code initializes an *ArrayList* object with a size 1. But as explained before, *ArrayList* objects can expand in need. So having a specified capacity does not prevent adding new values to these objects.

Then, three new values are added and the last one is removed since index 2 refers to the third element of the object which has three elements at all.

println() method prints the elements of the *ArrayList* objects in a neat way, which are [Natural History, Science].

Q9 **The answer is C.**

As mentioned above, *StringBuilder* methods return the object itself so that these objects are mutable. The *append()* method at the line 13 makes the value of *b* 123. Assigning this object to *b* again does not affect anything.

***reverse()* method does what it says in the name, it reverses the characters stored in *b*. Since the value of *b* is 123, the new value after calling *reverse()* method, will be 321.**

Q10 **The answer is D.**

Lambda expressions are not about converting primitive to wrapper classes, which is autoboxing, or about changing bytecode at runtime and inheriting from multiple class which are illegal in Java. **So options A, B, and C are incorrect.**

Lambdas use deferred execution which means that the code will run later. For example let’s say we have a method call as:

print(animals, a → a.canHop()).

Normally, a code line in Java is evaluated immediately. But in this line, *a.canHop()* expression will run after *print()* method calls *test()* method which is in *java.util.function.Predicate* interface. **So the option D is correct.**

Q11 **The answer is D.**

The line 5 creates a *StringBuilder* object with a value “-”. The line 6 creates another *StringBuilder* object by assigning it to the value of *line* after calling *append()* method on *line* variable. Since *append()* adds the value passed into it to the object and returns the object itself, *line* and *anotherLine* points to the same object which will cause the line 7 prints *true.*

Since *append()* method changes the value stored in *line*, length of this object became 2.

So the output will be *true 2*.

Q12 **The answer is D.**

A – *ArrayList* cannot be used since *ArrayList* does not have *length()* method. *ArrayList* returns the size via *size()* method. Even if this wasn't a problem, the third statement would cause problem since *ArrayList* creates a list consisting of *Object* elements and *Object* values cannot be assigned to *String* references without explicit casting.

B – *ArrayList<String>* also does not compile because of *length()* method being attempted to be invoked on *ArrayList* object.

C – *StringBuilder* does not have *add()* and *get()* methods. So this one also does not compile.

D – None of the above would work.

Q13 **The answer is D.**

Lambda expression does not have to have braces or brackets. If no braces and no brackets are to be used, the expression should be as:

b -> true

But if two arguments are used at the left side of a Lambda expression, or a type is specified explicitly for the argument braces must be used. And also if the right side contains more than one statements, brackets must be used, too. If brackets are used than the right side must be written as a normal statement in Java, such as ending with a semicolon, having *return* keyword. Except all these cases, braces and brackets can be used optionally.

So the above statement can be written as:

(StringBuilder b) -> {return true; }
(b) -> {return true;}
b -> {return true;}
(b) -> true
(StringBuilder b) -> true

A – *StringBuilder b* cannot be removed since *Predicate* cannot be assigned without any argument.

B – -> cannot be removed since it separates the parameter and the body.

C – If we remove *{* and *;}* then the expression becomes, *(StringBuilder b) -> return true*, which is not legal since *return* is used only when brackets are used.

D – If we remove *{ return* and *;}* then the expression becomes, *(StringBuilder b) -> true,* which is legal as showed above, the code continues to compile.

Q14 **The answer is A.**

An *ArrayList* object which consists of *Character* objects, is created at line 20. Next two lines add 'a' and 'b' *char* values which is okay since Java autuboxes these *char* values to *Character* objects.

Line 23 sets the second element, which is 'b', to 'c'. Then line 24 removes the first element. This makes that *chars* contains 1 element which is 'c'. So the print() method prints *1 false.*

Q15 **The answer is D.**

Line 12 creates a String object reference to "12" value. Line 13 creates a new String object concatenating the value of *b* and "3" and assigns this new object to the String reference *b.*

Line 14 attempts to invoke a *reverse()* method on *b,* which is illegal since *String* does not have any method named *reverse().* **This line causes a compilation error.**

If the line 14 is not written, the output would be "123".

Q16 **The answer is A.**

As explained in **Q13,** braces can be used optionally but must be used when the reference type of the argument is stated explicitly. So, *s -> false, (s) -> false, (String s) -> false* are correct syntaxes.

However *String s -> false* is incorrect since braces must used at the left side when reference type is stated explicitly. So only one line fails to compile.

Q17 **The answer is A.**

This code has no syntax error so it compile. Options C and D are incorrect.

Target interface defines one method so that this interface can be used with lambdas since Lambdas work with interfaces that have only one method. *prepare()* method takes a *double* argument and sends it to *Target* interface's *needToAim()* method to get a boolean value.

In main() method, *prepare()* method is invoked taking 45 and a lambda expression, *d -> d > 5 || d < -5,* which checks if *d* is bigger than 5 or less than -5. **prepare() method will use 45 as d and since it is bigger than 5, it will return true and the output will be true.**

Q18 **The answer is A.**

As explained before, *String* is an immutable class. This class's methods return a new/copy *String* object. *concat()* method also returns a new *String* object. Since the method calls in this code snippet are not assigned to any reference, these three calls create three objects by not assigning them to any reference.

So that the value of *teams* never changes and remains "694", which will be the output of the code.

Q19 **The answer is A.**

ArrayList is in the *java.util* package.

LocalDate is in the *java.time* package.

String is in the *java.lang* package.

So only the I is in the *java.util* package.

Q20 **The answer is C.**

A – *append()* method appends the value passed into it as an argument. So this statement makes *sb* "radical robots".

B – *delete()* method deletes characters between given indexes. *delete(1, 100)* deletes all characters starting from the second character. *append("robots")* method appends "robots" to *sb* making it "rrobots". *insert()* method inserts the given String value into the *StringBuilder* object starting from the given index. *insert(1, "adical r")* means inserting "adical r" starting from index 1 which makes the value of *sb*, "radical robots".

C – As explained above, *insert()* method inserts the given String value into the *StringBuilder* object starting from the given index. "radical " has 8 elements so that inserting "robots" starting from the index 7 makes the value of *sb* "radicalrobots". This is different than others.

D – Inserting *String* values starting from the length index of a *StringBuilder* object makes same thing as *append()* method so this statement makes *sb* "radical robots".

Q21 **The answer is A.**

First statement creates a String array. This array is converted to a List via *Arrays.asList()* method. This list's first element is set to another value which is "Art". Since *List* is mutable, it now contains "Art" value which will cause println() method to print *true*.

Q22 **The answer is D.**

contains() method checks if *String/StringBuilder* object contains a value among characters.

equals() method checks if *String/StringBuilder* object has exactly the same value.

startsWith() method checks if *String/StringBuilder* object's value starts with the given value.

A – A *String* object containing "abc" value, can have another characters, too, which means *equals("abc")* may be false.

B – A *String* object containing "abc" value, may start with a different value.

C – A *String* object starting with "abc" value, can have another characters, too, which means *equals("abc")* may be false.

D – A *String* object starting with "abc" value, definitely contains "abc" which makes this option true.

Q23 **The answer is D.**

This code snippet is same as in the **Q14** until the last statement. So the last state of *chars* is [c].

The last statement, which is print() method, attempts to invoke a *length()* method on *chars* which is not legal since *List* has no method named *length()*. **So this code does not compile.**

If we change *length()* to *size()*, the the output would be 1.

Q24 **The answer is B.**

All objects in Java has *toString()* method which means this method can be invoked on all types mentioned in the options.

A – *ArrayList* has no methods called *replace()* or *startsWith()*. So *ArrayList* causes compilation error.

B – *String* has all methods so it fits properly here.

C – *StringBuilder* has *replace(int,int,String)* method, which is not the one used in this method. *StringBuilder* does not have *startsWith()* method. So *StringBuilder* cannot be used, too.

Q25 **The answer is B.**

<> is known as *diamond operator.* This operator is used to specify the type of the elements to be stored in the *List.*

The right side of an expression may have this operator with an object type or empty. But the left side either specifies an object type in the operator or does not have the operator. If it does not have the operator then the object type would be generic, *Object.*

As a result if we put this operator at P, we must specify an object type but putting the operator at Q does not need any change. **So the answer is B.**

Q26 **The answer is D.**

This code defines a *Predicate<String>* reference. Since the generic type in Predicate is *String,* the argument to be given in the right side must be *String,* too, **which makes the options A, and B incorrect.**

Option C is a String type argument but it uses the same name as the parameter of main() method so that the code still does not compile. **So none of the options A, B and C are correct.**

Q27 **The answer is A.**

String class is immutable which means the data stored in *String* objects cannot be changed. *concat()* method takes a String value as argument and returns a new *String* object after storing the String value acquired by adding the argument value with the String value of the current object. Finally it returns the new *String* object. This means *line.concat("-")* doesn't change the value in *line. anotherLine* is assigned a new *String* object.

So the equality operator returns *false* and the length of the *line* is 1. **So the output is *false 1.***

Q28 **The answer is C.**

The first line does not state any generic type for *Predicate* so it default to *Object.* Since Object class does not implement any method named *startsWith()* this line fails to compile making **the option C correct.**

If we state *String* as a generic type for *Predicate,* then the line compiles and print() method prints the result of *test()* method. This method is a functional method defined in *Predicate* interface returning boolean value. Since *dash* checks if String objects starts with "-" and the given argument is "-", it returns true.

Q29 **The answer is B.**

LocalDate includes only the date while *LocalDateTime* and *LocalTime* includes hours, minutes and seconds. **So only two of them includes hours, minutes and seconds.**

LocalDateTime also includes date beside time.

Q30 **The answer is D.**

substring() method returns the part of String between the given indexes, including the character at the former index. If the second index is not given, then this method returns a String object with value starting from the given index until to the end. This means that *builder.substring(4)* returns the part of the String after the 4. index. It is only "1". So *builder* now points a String object with a value of "1".

println() method attempts to access third element of *builder* which does not exist. **So this code throws *StringIndexOutOfBoundsException,* which makes the option D correct.**

Q31 **The answer is D.**

A – When brackets are used, *return* keyword at the beginning and a semicolon at the end must be used. This expression misses both.

B – This expression misses *return* keyword.

C – This expression has *return* but misses semicolon.

D – This expression has both *return* and semicolon.

Q32 **The answer is B.**

LocalDate class is immutable so that methods called on this type of objects returns a copy of the objects.

The first line creates a *LocalDate* object and the second line reduces the day by 1. This line returns a new *LocalDate* object but since it is not assigned to any reference, ***xmas* is not affected by this. Because *LocalDate* does not change, println() method prints 25.**

Q33 **The answer is A.**

Line 4 appends "red" to the empty *StringBuilder* object. Line 5 deletes the character at the index 0, which is 'r'. Line 6 deletes the characters between the indexes 1 and 2, including 1, which is 'd'. So *StringBuilder* object's value becomes "e", which will be printed at the end.

Q34 **The answer is B.**

The first line defines a *Predicate* reference with generic type *Object.* This *Predicate* reference *clear* checks if String objects are equal to “clear”.

print() method passes “pink” into the *test()* method of *Predicate* to be compared with “clear” which will return false. **So the output will be *false.***

Q35 **The answer is C.**

Months and days in Java starts counting from 1. Arrays and Strings starts from 0.

LocalTime does not have month and day fields so that the **option D is incorrect.**

***LocalDateTime* includes month and day so that the option C is correct.**

Q36 **The answer is C.**

Predicate is a functional interface defined to use with lambda expressions. It has one method named *test*, which takes one argument and returns a boolean value.

The method *test* does not take two parameters which makes the option C incorrect.

Q37 **The answer is B.**

ofDays() and *ofWeeks()* methods are static so that they cannot be chained. *period1* is set to period of every one week at the first but after invoking *ofDays()* it becomes a period of every 3 days.

period2 is set to period of every 10 days so **this one is the larger amount of time which makes the option B correct.**

Q38 **The answer is B.**

This code creates a *LocalDate* which is *2017,1,1*. Then a period of 1 days is created. And then a date format is created with *DateTimeFormatter* which is in *java.time.format* package.

print() method prints the date in *newYears* reference after reducing it by *period* which is 1 day. After subtracting 1 day, date changes totally and *LocalDate* handles it making it *2016, 12, 31*.

format is “MM-dd-yyyy”. MM stands for two-digit Month. dd stands for two-digit day. yyyy stands for four-digit year. And dashes are for that they are put between fields. **So that output would be *12-31-2016.***

Q39 **The answer is C.**

trim() method removes white spaces, tabs and new line characters at the beginning and at the end. It does not affect the white spaces at the middle. So *question* is “:) - (:”.

substring() method returns the part of String between the given indexes, including the character at the former index. So we need such arguments that will make *substring* return the characters between the whitespaces in *happy.* The first non-whitespace character of *happy* is at index 1. Since the *substring()* method does not include the latter index and *happy's* last index is a white space, the latter index must be the last index of *happy* which is *length() - 1* for all arrays and strings.

So the only method calling which will produce the same string as *trim()* is *happy.substring(1, happy.length()-1).*

Q40 **The answer is C.**

A – A period is immutable so that methods getting invoked on these objects return a copy of the current object. So that if return values of methods are not assigned to any reference, the result of the methods gets lost.

B – A period is used to add and subtract time from/to dates.

C – Periods can only represent dates (days, weeks, months and years), not time (minutes, hours, second, etc.).

D – As explained above, periods can represent dates.

Q41 **The answer is D.**

StringBuilder is a mutable class but *substring()* method returns *String* object. So calling *substring()* on *StringBuilder* objects does not change the data in the object.

So println() method will print the second character of "54321", which is '4'.

Q42 **The answer is B.**

This code creates an ArrayList consisting of Integer objects. Next three lines add 3, 2, 1, respectively. Then *remove(int index)* is invoked on this list. This method takes *int* argument and removes the element that is in the given index. So *remove(2)* does not remove the object, that is 2, but removes the object at the index 2, which is 1. **So println() method prints [3, 2].**

Q43 **The answer is C.**

A – ArrayList does not have any of the methods used in this code. So ArrayList cannot be the answer.

B – String class does not have *insert()* method.

C – *StringBuilder* has all of the methods so that putting this in the blank ensures that the code compiles.

Q44 **The answer is C.**

Seconds are *second*, *decisecond*, *centisecond*, *millisecond*, *microsecond*, *nanosecond*, *and picosecond*, in order from larger to smaller. In this question, the smallest unit is *picosecond* but *LocalTime* does not have this unit as a field. **So the option D is incorrect.**

The second smallest unit is *nanosecond* and *LocalTime* has this as a unit so the option C is correct.

Millisecond is larger than *nanosecond* but *LocalTime* does not also have a field for it.

Q45 **The answer is D.**

This code is same as in **Q38,** with one difference that the format defined for *format* is "mm-dd-yyyy". "mm" represents minutes not months. For months, "MM" is used. Since *LocalDate* does not have time fields, this *format* cannot be applied to *LocalDate newYears.* The code compiles, though. **This code throws an *UnsupportedTemporalTypeException* at runtime.**

Q46 **The answer is D.**

replace() method has two signatures as follows:

String replace(char oldChar, char newChar),
String replace(CharSequence oldChar, CharSequence newChar).

So data type *char* can be used in *replace()* method. *CharSequence* is an interface and both *String* and *StringBuilder* implements this interface which means both can be used in *replace()* method.

So the answer is I, II, and III.

Q47 **The answer is C.**

This code creates an *ArrayList whose generic type is Strin*g and adds -5, 0, 5, respectively. And then print() method is called giving the list and a lambda expression, *e -> e < 0,* which will return values that is less then 0, as arguments.

The code is okay until here. However, print() method gives *String* values to *Predicate* reference p, whose generic is *Integer.* Since *p* expects an *Integer* as argument **this code would not compile.**

If we attempt to fix the code by converting String values to Integer, then the would compile and print -5 since it is the only element that is less than 0.

Q48 **The answer is D.**

This code creates an ArrayList consisting of *String* objects. Then two String values are added via *add()* method but all is removed via *clear()* method. After that one String value is added again.

At the line 17, the element at the index 1 is attempted to be removed. But *magazines* has only 1 element so this line will throw *IndexOutOfBoundsException* at runtime which makes the option D correct.

If the line 17 is removed, println() method prints 1. If line 17 attempts to remove the element at the index 0, println() methods prints 0.

Q49 **The answer is C.**

First statement in the main() method attempts to assign a *char* value to a *String* reference which is illegal so that this code does not compile.

If we fix the problem by changing 'b' to "b", which means changing from *char* to *String*, then the code would compile. *concat(tail)* method returns a new *String* object and this new object is assigned to *witch*. So println() would print "black".

Q50 **The answer is C.**

LocalDate is an immutable class. Since the data in immutable classes is not meant to be changed, immutable classes do not have setter methods. *setYear()* method is not defined in *LocalDate* class, too. **So this code does not compile.**

If the second line is removed then the code would compile and print 2016.