Bünyamin AKTAŞ
12/06/2020

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 by appending or by removing values to/from itself unlike *String*, it guarantees that less memory is used when *StringBuilder* 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* is final since it is immutable.

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.**

Since both *String* and *StringBuilder* allow method chaining when creating an object so that all of these options create objects with same value.

Q4 **The answer is B.**

StringBuilder is mutable and *append()* method returns the object itself which makes the object have the old value appended the value in the argument passed into *append()* method. **So *teams* object has “333 806 1601” value.**

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 type 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 has a default size.

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

Even if the println() method attempted to access the third element, it would print "hex key" since this is considered a single String value, not two words.

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.

Q8 **The answer is A.**

This code initialize 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 in *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

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

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**

Q17 **The answer is**

Q18 **The answer is A.**

As explained before, *String* is an immutable class. This class’s methods return a new *String* object. *concat()* method also returns a new *String* object. Since the method calls in this code snippet are 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 *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. The 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.