ASSIGNMENT 9
Classes from the Java API

Q1-) Option A

StringBuffer (Thread-safe)

StringBuffer is thread-safe meaning that they have synchronized methods to control access so that only one thread can access StringBuffer object's synchronized code at a time. StringBuffer objects are generally safe to use in a multi-threaded environment where multiple threads may be trying to access the same StringBuffer object at the same time.

StringBuilder (Non-thread-safe)

StringBuilder is not synchronized so that it is not thread-safe. By not being synchronized, the performance of StringBuilder can be better than StringBuffer. If we are working in a single-threaded environment, using StringBuilder instead of StringBuffer may result in increased performance. This is also true of other situations such as a StringBuilder local variable where only one thread will be accessing a StringBuilder object.

StringBuilder is not thread-safe. It does not add support for multiple threads so, Option A is not true. Option B is also false because "==" is used to compare object references, not values. In order to compare StringBuilder values, we can use "toString()" and "equals()" methods as follows:

```
StringBuilder sbld1 = new StringBuilder("Hello");
StringBuilder sbld2 = new StringBuilder("Hello");

if(sbld1.toString().equals( sbld2.toString() )){
    System.out.println("StringBuilder objects are equal");
}else{
    System.out.println("StringBuilder objects are not equal");
}
```

StringBuilder supports different languages and encodings but this is not a reason for using StringBuilder. Because String also supports different languages and encoding. This makes Option D incorrect.

String vs StringBuffer

Since String is immutable in Java, whenever we do String manipulation like concatenation, substring, etc. it generates a new String and discards the older String for garbage collection. These are heavy operations and generate a lot of garbage in heap. So Java has provided StringBuffer and StringBuilder classes that should be used for String manipulation. Namely, ***"`StringBuilder`" saves memory by reducing the number of objects created and this is the best reason for using it instead of "`String`".***

https://www.journaldev.com/538/string-vs-stringbuffer-vs-stringbuilder#:~:text=String%20is%20immutable%20whereas%20StringBuffer,StringBuilder%20is%20faster%20than%20StringBuffer.

Q2-) Option D

```
String sys = "Advent";
String sys = new String("Advent");
```

Both lines create a `"String"` called sys. So, creating `"String"` without coding a call to a constructor is possible. This makes **Option A** is true.

The string pool is a location in the JVM that collects all strings. The string pool contains literal values that appear in the program. Strings not in the string pool are garbage collected just like any other object.

"sys" object that created without a call to constructor uses the string pool normally. The other sys object do not use the string pool. A new object is created and this is less efficient. String pool provides us to reuse String literals. So, **Option B** is also true.

A string is final and immutable, it cannot be changed once it's created. So, **Option C** is true, but **Option D** is false and it is the answer.

Q3-) Option D

```
new StringBuilder().append("clown")
new StringBuilder("clown")
new StringBuilder("cl").insert(2, "own")
```

All these lines create an object with same value ("clown"). **Option D** is the answer.

Q4-) Option B

`"append method()"` adds the parameter to the `"StringBuilder"` and returns a reference to the current `"StringBuilder".` `"StringBuilder teams"` created with value "333". Then, " 806" is appended to it. Finally " 1601" is appended and "teams" becomes `"333 806 1601"` which is the output. **Option B** is the answer.

Q05-) Option B

`"List"` in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The "List" interface is found in the `"java.util"` package and inherits the "Collection" interface. It is a factory of `"ListIterator"` interface. Through the `"ListIterator",` we can iterate the list in forward and backward directions. The implementation classes of `"List"` interface are **`"ArrayList"`**, **`"LinkedList"`**, **`"Stack"`** and **`"Vector"`**. The `"Vector"` class is deprecated since Java 5.

https://www.javatpoint.com/java-list

“`List`” is an interface, so it cannot be instantiated. In order to compile this code, we must fill in the blank with a class that implements “`List`”. Since “`ArrayList`” class one of these class, it can fill in the blank to produce code that compiles. **Option B** is the answer.

Q06-) Option C

"`ArrayList tools`" is created and values "`hammer`", "`nail`", and "`hex key`" are inserted to list. Then, the second element of tools which is "nail" is printed. **Option C** is the answer.

Q7-) Option C

The code does not compile because "`length()`" method cannot be called before "`StringBuilder sb`" was created. **Option C** is the answer.

Q8-) Option A

The code compiles and prints "[Natural History, Science]". Even if "`ArrayList museums`" is created with one slot, it can be expanded. "Natural History", "Science", and "Art" are added to list by "`add()`" method then "Science" which is the third element of the list is removed by "`remove()`" method. **Option A** is the answer.

Q9-) Option C

The code compiles and prints "321". "`StringBuilder b`" is created and "12" is assigned it. Then "3" is appended and makes "`StringBuilder b`" "123". "`reverse()`" method reverses the characters and returns a reference to the current StringBuilder. So, the output is "321". **Option C** is the answer.

Q10-) -

Q11-) Option D

The code compiles and prints “`true 2`”. “`line.append(“-”)`” adds “`-`” to “`line`” and also returns reference to “`line`”. So, “`line==anotherLine`” is always true and length of them is “2”. **Option D** is the answer.

Q12-) Option D

We cannot fill in the blank with "`StringBuilder`" because "`add()`" and "`get()`" methods are not compatible with it. These methods can be used with "`ArrayList`" but "`length()`" method is not available on "`ArrayList`" class. In order to get number of elements in an "`ArrayList`" "`size()`" method should be used. So, none of these types is the proper for the blank. **Option D** is the answer.

Q13-) -

```
Predicate<StringBuilder> p = (StringBuilder b) -> {return true;};
```

Option A: Remove `StringBuilder b`

We cannot remove this because a Predicated needs a parameter.

Option B: Remove `->`

We cannot remove this because "->" must be used in the lambda expression.

Option C: Remove `{ and ;}`

Option D: Remove `{ return and ;}`

Q14-) Option A

The code compiles and prints "1 false". 'a', 'b' are added to list in order and than 'b' which is at index one is set to 'c'. After that, 'a' which is at index zero is removed. So, size of "chars" becomes 1 and "chars" does not include "b". **Option A** is the answer.

Q15-) Option D

The code does not compile because "`reverse()`" method is not available on "`String`" class. It can be used with "`StringBuilder`" class. **Option D** is the answer.

Q16-) Option A

```
Predicate<String> pred3 = String s -> false;
```

This line prevents the code from compiling. If parameter type is declared, parentheses are needed. This line can be corrected as follows:

```
Predicate<String> pred3 = (String s) -> false;
```

Q17-) Option A

The code compiles and prints “true”. Body of lambda expression is true because 45 is greater than 5. **Option A** is the answer.

Q18-) Option A

The code compiles and prints “694”. Since String class is immutable, “concat” method does not return reference to `“String teams”.` It returns a new object and “teams“ remains its value which is “694.”. **Option A** is the answer.

Q19-) Option A

`ArrayList:` java.util

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

`LocalDate:` java.time

https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html

`String:` java.time

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Only “ArrayList” class is in the “java.util” package. **Option A** is the answer.

Q20-) -

Q21-) Option A

The code compiles and prints "true". The `“asList()”` method of `“java.util.Arrays”` class is used to return a fixed-size list backed by the specified array. Since the first element of museums list which is "Natural History" is set to "Art", `museums.contains("Art")` returns true. **Option A** is the answer.

Q22-) Option C

Option A:

If `String s="abc123"`, `s.contains("abc")` is true but `s.equals("abc")` is false.

Option B:

If `String s="123abc";` `s1.contains("abc")` is true but `s2.startsWith("abc")` is false

Option C:

If `s.startsWith("abc")` is true, `s.contains("abc`") is always true. This option is the answer.

Option D:

If `String s="abc123"`, `s.contains("abc")` is true but `s.equals("abc")` is false

Q23-) Option D

The code does not compile because `"length()"` method is not available on `"ArrayList"` class. As explained in Q12, in order to get number of elements of an `"ArrayList"`, `"size()"` method should be used. **Option D.**

Q24-) Option B

Option A: `ArrayList`

The method replace(String, String) and the method startsWith(String) are undefined for the type ArrayList.

Option B: `String`

This is the answer. String is can be used to fill in the blank. All these method signatures are valid for class String.

Option C: `StringBuilder`

The method replace(int, int, String) in the type StringBuilder is not applicable for the arguments (String, String) and the method startsWith(String) is undefined for the type StringBuilder

Q25-) Option B

```
ArrayList<String> list = new ArrayList<String>();
ArrayList<String> list = new ArrayList<>();
```

Java 5 allows us to tell the compiler what the type would be by specifying it between < and >. Starting in Java 7, we can even omit that type from the right side. The < and > are still required, though. This is called the diamond operator because <> looks like a diamond. The type cannot be omitted from the left side (P), so options A and C are incorrect. We can omit <> from the right side (P). **Option B** is the answer.

Q26-) Option D

Lambda expression's parameter is expected to be of type `"String"`, so options **A and B** are incorrect. Although the type is "String", **Option C** is also incorrect because variable name `"s"` already used as parameter of main method. It prevents the code from compiling. None of the options can fill in the blank to make the code compile. **Option D** is the answer.

Q27-) Option A

The code compiles and prints "false 1". Since String is immutable, `"line.concat("-")"` returns a different object. So, `"line == anotherLine"` is false and `"line.length()"` is still "1". **Option A** is the answer.

Q28-) Option C

The code does not compile. "Since" a the Predicate does not have a generic type, `"c"` is an `"Object"` and `"startsWith()"` method is undefined for the type `"Object"`. **Option C** is the answer.

Q29-) Option B

`LocalDate:`

Contains just a date, no time and no time zone. A good example of LocalDate is your birthday this year. It is your birthday for a full day regardless of what time it is.

`LocalTime:`

Contains just a time—no date and no time zone. A good example of LocalTime is midnight. It is midnight at the same time every day.

`LocalDateTime:`

Contains both a date and time but no time zone. A good example of LocalDateTime is “the stroke of midnight on New Year’s.” Midnight on January 2 isn’t nearly as special, and clearly an hour after midnight isn't as special either.

```
System.out.println(LocalDate.now()); // prints 2020-06-12

System.out.println(LocalTime.now()); // prints 11:14:08.743

System.out.println(LocalDateTime.now()); // prints 2020-06-12T11:14:08.743
```

LocalTime and LocalDateTime both include hours, minutes, and seconds. There is not a class called "`LocalTimeStamp`" in Java. **Option B** is the answer.

Q30-) Option D

The code compiles and throws a "`StringIndexOutOfBoundsException`" due to line "6". "`builder.substring(4)`" says to take the characters starting with index 4 through the end, which gives us "1". "1" is assigned to "builder". "`builder.charAt(2)`" asks for the third character of the builder, but there are only one characters present. **Option D** is the answer.

Q31-) Option D

```
<Predicate<Integer> ip = i -> i != 0;
```

In lambda expressions; if we use brackets, we must use both "return" and ";".

```
i -> { return i != 0; }
```

This is the correct syntax, **Option D** is the answer.

Q32-) Option B

The code compiles and prints "25". "`LocalDate`" is immutable so, "`xmas.plusDays(-1)`" does not change the value of day which is "25". **Option B** is the answer.

Q33-) Option A

The code compiles and prints "e". First, "red" is appended to empty "`StringBuilder sb`". Then, The first character is removed and "`StringBuilder sb`" becomes "ed". Finally, "`delete()`" method deletes the second element which is "d". **Option A** is the answer.

Q34-) Option B

If parameter type is not declared, Java assume that as an `"Object"`. `"equals()"` method available on `"Object"` so the code compiles without an issue. Since "pink" does not equal to "clear", `"clear.test()"` method returns false. **Option B** is the answer.

Q35-) Option C

Option A: Array indexes

Array indexes are start with zero. The first element of an array is at index "0".

Option B: The index used by charAt in a String

The Java String charAt(int index) method returns the character at the specified index in a string. The index value that we pass in this method should be between 0 and (length of string-1).

Option C: The months in a LocalDateTime

The months in a LocalDateTime starts counting from one.

Option D: The months in a LocalTime

LocalTime contains just a time—no date and no time zone, so no months.

Q36-) Option C

Predicate is a functional interface. It has one method named test that returns a boolean. This method takes any type and only one parameter. So, **Option C** is false and it is the answer.

https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

37-) Option B

```
Period period1 = Period.ofWeeks(1).ofDays(3);
```

Since "Period.ofXXX" methods are static methods, only the last method "ofDays(3)" is used. So "period1"means every three days.

```
Period period2 = Period.ofDays(10);
```

period2 means every ten days.

Namely, period2 represents a larger amount of time. **Option B** is the answer.

Q38-) Option B

The code compiles and prints "12-31-2016". `"LocalDate newYears"` is created as "2017-01-01" `("yyyy-MM-dd")`. Then, `"Period period"` which represents one day is created. After this line, `“DateTimeFormatter format”` is defined with the pattern `"("MM-dd-yyyy")"`. Finally, one day is subtracted and the formatted version is printed. **Option B** is the answer.

Q39-) Option C

The `“trim()”` method removes whitespace from the beginning and end of " :) - (: ", then ":) - (:" is assigned to "really". The method `“substring()”` looks for characters in a string and returns parts of the string. The method signature of it as follows:

```
int substring(int beginIndex, int endIndex)
```

In order to get ":) - (:", we should take the characters starting with index "1" of "happy" until, but not including, the the last character which is (happy.length() - 1).

Option C (`happy.substring(1, happy.length() - 1)`) is the answer.

Q40-) Option C

Objects of Period class are immutable and we can use them for adding or subtracting time from dates. By using `"Period.ofDays", "Period.ofWeeks", "Period.ofYears"` we can create periods which represent days, weeks or years. But there is not a method in order to create a period which represents minutes. **Option C** is the answer.

Q41-) Option D

The code compiles and prints "4". Since `"substring()"` method returns a `“String”` rather than a `“StringBuilder”, "StringBuilder builder"` is not changed. The code prints "4" because "1" is at fourth index of `"builder"`. **Option D** is the answer.

Q42-) Option B

The code compiles and prints "[3, 2]". `"ArrayList pennies"` is created and "3", "2" and "1" is added to list in order then "2"is removed. **Option B** is the answer.

Q43-) Option C

“`StringBuilder`” can best fill in the blank to complete this method. “`charAt()`”, “length()”, and “`insert()`” methods are not available on “`ArrayList`”. “`insert()`” is also cannot be used on a “`String`”. The code does not compiles with a “`String`” or “`ArrayList`” parameter. **Option C** is the answer.

Q44-) Option C

“`LocalTime`” is an immutable date-time object that represents a time, often viewed as hour-minute-second. Time is represented to nanosecond precision. For example, the value "13:45.30.123456789" can be stored in a “`LocalTime`”. **Option C** is the answer.

https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html

Q45-) Option D

The code compiles and throws “`UnsupportedTemporalTypeException`”. Lowercase 'mm' (minute) pattern is used but probably 'MM' (month) must be used to format a “`LocalDate`” object because these objects do not include minutes. **Option D** is the answer.

Q46-) Option D

-
  ```
  public String replace(char oldChar,
                        char newChar)
  ```

Returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.

If the character `oldChar` does not occur in the character sequence represented by this `String` object, then a reference to this `String` object is returned. Otherwise, a `String` object is returned that represents a character sequence identical to the character sequence represented by this `String` object, except that every occurrence of `oldChar` is replaced by an occurrence of `newChar`.

Examples:

```
"mesquite in your cellar".replace('e', 'o')
        returns "mosquito in your collar"
"the war of baronets".replace('r', 'y')
        returns "the way of bayonets"
"sparring with a purple porpoise".replace('p', 't')
        returns "starring with a turtle tortoise"
"JonL".replace('q', 'x') returns "JonL" (no change)
```

Parameters:

    oldChar - the old character.

    newChar - the new character.

Returns:

    a string derived from this string by replacing every occurrence
    of oldChar with newChar.

- `public String replace(CharSequence target,`
  `                      CharSequence replacement)`

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Parameters:

    target - The sequence of char values to be replaced

    replacement - The replacement sequence of char values

Returns:

    The resulting string

Since:

    1.5

Two versions of "replace()" method on String class is explained above. The first version takes char parameters and the second version takes char sequences such as String and StringBuilder. **Option D** is the answer.

Q47-) Option C

The code does not compile because "num" is a "String" but print method takes Integer as argument. **Option C** is the answer.

Q48-) Option D

The code compiles bur it throws "`IndexOutOfBoundsException`" at runtime. "magazines.clear()" removes all elements of the "ArrayList magazines". Than "The Economist" is added to list as first element Ih is at index "0". Attempting to remove the element at index "1" causes this exception index "1" does not exist. There is only one element and it is at index "0". **Option D** is the answer.

Q49-) Option C

The code does not compile because 'b' which is a "char" cannot be assigned to "String witch". **Option C** is the answer.

Q50-) Option C

The code does not compile because "`LocalDate`" class does not have a setter method. This class is immutable. **Option C** is the answer.