Homework 10

Q1)(E)

The compiles but throws an exception at runtime. `sb` variable is initialized with an empty `StringBuilder` object. Then the program arguments are iterated. For every given argument, for loop tries to insert the `String` element being iterated to the `sb` `StringBuilder` at the index of character `'c'` in `sb`. But since the the `sb` `StringBuilder` is empty, `sb.indexof("c")` return `-1`, and hence code throws `StringIndexOutOfBoundsException` at runtime.

Q2)(C, E)

Addition operator has higher precedence over assignment operator, so the option A is wrong. For option B, addition operator is listed last, but it has lower precedence over the previous three operators, so it is wrong too. Option D is wrong because the subtraction operator has lower precedence over the multiplication operator. Option C and option E has the correct order of operators according to increasing or the same level of precedence.

Q3)(C, B, F)

Option A is a getter method but it has a parameter in its method signature, so it is not a valid JavaBean. Option D should should return a boolean, looking at the name, at it is prefixed with ‘is’, so this option is not a valid bean too. Lastly Option E is wrong because there is no such naming convention. C, B, F are valid beans.

Q4)(A, E)

Array has `length` variable instead of a `size()` method. `size()` method is used for ArrayLists. Upon the correction of this one line, code would compile and run without issue. The nested for loop iterates correctly, and without trying to access out of bounds of the crossword array. It only iterates through the half of the every array in the second dimension though. Because the second dimension has `10` arrays in it with every array having length `20`, we are iterating every `10` array but stopping after the ninth element which corresponds to half of the whole elements. And unassigned half has their values set to default value `0`.

Q5)(B, D)

If a file system resource becomes temporarily unavailable, a checked exception must be used. Error is not a subclass of the class `Exception`, but `Throwable` class. If a user enters invalid input, i think an unchecked exception must be used, because we can consider it a programming error. Although it is possible, it is very unlikely that we would’ve want to catch it.

Q6)(D, A, C)

`import jungle.tree.*` statement imports all of the class inside tree package, allows us to access them without specifying the full package name. This is also true for the import statement `import savana.*`. `import forest.Bird` imports a class, `Bird` class, so again we don't need to use its full package name. `java.lang.Object` package does not need to be imported, since its inherently imported, we can use `Object` class without a full package name. `savana.sand.Wave` class cannot be accessed as asked in the question since the import statement only import classes in the `savana` package. Classes option E and F, should be used with full package name too since only Bird class imported from the forest package. Key point here is that we should not overlook the fact that by using '*', we are not recursively importing all packages.

Q7)(C)

Two of the given four variables represent immutable objects. `Strings` and `LocalDateTime` object are immutable objects, while `Arraylists` and `StringBuilder` classes represent mutable objects.

Q8)(C)

The output of the given code is `"wing"`. Given code iteratively removes characters from the `StringBuilder` unless the length of the `Character` sequence is less than `5`. First iteration leaves the object with the value `"s growing"`, since the length is greater than `5`, iteration runs again, leaving the value with `"wing"`. `while` condition check returns `false` since the length is now less than `5`. And the last value of the variable which is `"wing"` is printed after the `do-while` loop.

Q9)(D)

All of the created `String` objects are new objects, meaning that the all three variables points to different object. `new` keyword always instantiates a new object in the heap. Only the `ceiling` string is interned, but the next two variable are instantiated and initialized with `new` keyword. The `==` operator checks if the references point to the same object, so since none of the three point to the same object as other, there is no any possibility that any equality check on these three references returns `true`. equals method returns `true` with every combination of these three variables, one of them as caller other the argument. Because the `equals` method is implemented to check the elements in the `String` rather than comparing references.

Q10)(C)

The code prints true three times. Since `String` objects are mutable, all the methods which return a `String` after an operation which seems to mutate the string, actually returns a new `String` object. Because again, `Strings` are immutable. Last three statements prints **`true`**. Since they are equality checks on the contents of the strings rather than the references, and the `ignoreCase` is used where the contents are different.

Q11)(A, B, C)

Lines 15 and 17 can be removed since we are not referencing to them. Lines 15, 16, 21 describes the outer loop with its label. We can remove those lines safely. Because the operation is inside the inner do-while, and the condition of the outer loop does not affect the output since when the inner loop condition returns **`false`**, outer loop condition would also, always return **`false`**. So it doesn't really affect the outcome, but it would if the condition would be checking against some value bigger than `5`.

Option D describes removing the inner do-while, which would cause the output to change since the condition in the **`while`** statement is different. There would be 4 `'x'` character added and printed instead of `5`. Option E and F is certainly wrong since they suggest a do-while without a while.

Q12)(B, C)

Option B does not compile since since we are assigning a **`double`** literal to a **`long`** type reference, java cannot convert **`double`** to **`long`** since it would not be promotion. But note that a **`long`** value would be promoted to **`double`** value. Option C does not compile too because a floating point number is given with a `'L'` postfix which defines a **`long`**. That is clearly a syntax error. Option D and E does compile, both `'l'` and the uppercase `'L'` are practically the same. Option A compiles, because an **`int`** can be inherently promoted to **`long`**. Option F compiles too because underscores are fine to use between numbers for readability of big numbers.

Q13)(A)

Since `time.getHour()` returns `1`, **`while`** loop is never branched, hence no line is printed. But keep in mind that the `time.plusHours()` is dangerous here since it returns a reference to a copy, not a reference to the time object itself, since the `LocalTime` class is immutable. If the while condition was to evaluate **`true`**, we would have an infinite loop.

Q14)(D)

The code compiles but throws a `NullPointerException` at runtime. Since `game` variable is not initialized, it gets the value **`null`** which is the default value for objects. But hidden with the first thrown exception there is another issue with the code. A reference type variable of

one-dimensional Object array is assigned with `game` variable, which is a two-dimensional array itself. Which means that the every element we obtain with `obj` reference (with something like `obj[3]`) is an one dimensional **`int`** array (**`int`**`[]`). A character is assigned to where should be an object of one-dimensional array. This is a problem which would cause an `ArrayStoreException`.

Q15)(C, E)

Option B, D and F does not even compile because no type argument given on the left hand side. Option A gives a warning because of the usage of a raw type when instantiating the ArrayList. Compiler gives warnings because a raw type prevents compiler from doing some type safety checks. It is strongly discouraged to use raw types.

Option C and E does compile without warning because they properly utilize generics. For the option C (`List<String> c = new ArrayList<>();`), it is not mandatory to give a type argument on the right side, because it is inferrible by the compiler from the argument given to the reference type on the left side.

Q16)(B, D)

After the assignments, `shoe1` and `shoe3` references both points to `"flip flop"` and `shoe2` reference points to `"croc"`. `"sandal"` is not referenced by any variable so it is eligible for garbage collection just before the `main()` method ends. After it ends, all off them are eligible anyway. Option D is correct because being eligible to be garbage collected does not mean it is guaranteed that the garbage collector will run.

Q17)(C)

One line of the application does not compile. ”`f.getFish();`” statement does not compile because even though the `Clownfish` class’ `getFish()` method does not declare an exception `Fish` class’ `getFish()` method does and the type of the reference variable `'f'` is `Fish` so we need to declare or handle it in the `main()` method.

Q18)(A)

Only one line printed. static `print()` method expects a `List` and a `Predicate` as parameter. In `print()` method, every item in the list is iterated and is given as argument to Predicate’s `test()` method. If the `test()` returns **`true`** number is printed.
In the `main()` method, we see that a lambda expression “`e -> e < 0`” is given to the `print()` method. This expression tests the parameter `e` if it is less then `0` or not. The given `list` variable contains three items, one of which is ‘`-5`’. This item makes the `test()` method returns **`true`**, and hence it is printed.

Q19)(F)

None of the given options are mandatory with a **`try`** statement. `finalize()` is a `Object` method which is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. **`catch`** and **`finally`** blocks are optional as long as one of them is present. **`throws`** keyword is used to declare exceptions in method signatures.

Q20)(A)

The code outputs `5`. The outer while loop is branched because variable `result` is bigger than `7`. Inner do-while loop decrements the `result` variable until it's equal to `5` and then hands over the control because the contiditon does not meet anymore. The very next statement after the inner do-while loop breaks the loop labeled "loop" which is the outer loop itself. Using a label here is unnecessary actually. After that, we see a print statement which print the value of `result` variable which is `5`.

Q21)(C)

The result of the given application after compiling and executing is "`1 2`". The constructor of the Alligator class increments the class variable, which by the way is initialized with default value `0`, every time when an instance of the class is created. The `teeth` in the `main()` method resolves to the class variable `teeth`. With the first invoke of the `snap()` method, `teeth` is already incremented by `1` so `1` is passed to the `snap()` method. But the `snap()` method's parameter is also named `"teeth"` so it shadows the class variable `teeth` which resides in the outer scope. Referring `teeth` inside `snap()` method does not refers to the class variable but the local variable `teeth`, hence the class variable is not affected by the decrementation. So, the first invoke of the `snap()` method prints 1, and by the second invoke of the `snap()` method, for the reasons explained above, `2` is printed.

Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()

Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()
Q)()