1) This is **true**. There is void methods which do not return anything and there is return type methods which will return a single value. Return methods must always have a return value otherwise there will be a compile error.

2) This is **false**. If a method has parameters, you must give those arguments and match the datatypes whenever calling the method. If the arguments are not passed during a method call, there will be a compile error.

3) Method overloading will help **improve readability and reusability,** allow **flexibility to call similar methods with different inputs,** and allows the **methods [to] share the same name**

4) This is **true**. To overload a method means to have a method with the same name. In order for a method to be overloaded it should have a different number of parameters or the datatype of the parameters should be different.

5) This is **false**. We talked about what overloading a method was in the previous question. Yes, the method name should be the same, but changing the return type of a method does not overload a method. Changing only the return type and nothing else will cause a compile error. If you overload a method successfully following the rules that we talked about previously then you are able to change the return type of the overloaded method if needed, but again the return type has nothing to do with if a method is overloaded properly or not.

6) The parts of the methods are:

    1. **Return type**
    2. **Method name**
    3. **Parameters**

7) Here we have a void method that is called in the main method. The method1 has a number that gets modified as the loop iterates through. This is the flow:

n value	i	Calculation from loop	n after calculation
5	0	5 + 0	5
5	1	5 + 1	6
6	2	6 + 2	8
8	3	8 + 3	11
11	4	11 + 4	**15**

The value of n when it is printed is **15**

8) Here we have a void method which accept a String parameter ‘s’. Executing the method2 will give us a String ‘a’ which takes the substring of our given argument starting at the 10^{th} index. The resulting string would be: *“t quickly and created a shadow”*. The if statement checks if the length of ‘a’ is more than the length of ‘s’. The length of the ‘a’

string will always be smaller than the length of the ‘s’ string because ‘a’ is a substring of ‘s’ so our if condition is false which will result in the else statement being executed which prints out ‘s’, which is just the original argument passed: *“**The sun set quickly and created a shadow**”*

9) For method3 we accept an int value which will be referred to by ‘num’ and return a String value. The argument given in the main method is 50 so when the if condition in the method is executed the else if (num < 100) statement would be when the condition is true because the one before it (num < 50) would be false because 50 != 50. The String “Less than 100” is returned from method3 in this case. That String returned is then printing given out: **Less than 100.**

10) In method4 there is two parameters: a char ‘c’ and a String ‘s’. In the main method when the method is invoked the arguments given are: “z” and “zebra”. This line will have a **compile error** because the datatype of the method do not match. The first argument is a char datatype, but the given argument is a String because of this the code will not compile and the rest of the code snippet does not need to be looked at.

11) Just to emphasize again, the order of methods inside of a class do not matter. We have method5 which accepts a String ‘str’ and returns a boolean value. The argument given in the main method is “apple”. When the method begins being executed there is a boolean ‘check’ initialized as true. The first if condition checks if our given string is empty. The string ‘str’ has a value of “apple”, so it is not empty, so the if statement block will not be executed. The 2nd if condition checks if the string does contain “a”. “apple” does contain “a” so the first if condition is true. Because the condition was true the ‘check’ boolean is changed to **false**. The ‘check’ boolean is then returned and printed. The output would show the last value of the boolean before the return which would be **false**.

12) Here we have a method7 which accepts a String ‘str’ and an int ‘s’ and the method returns a String. The arguments given when calling method7 in the main method are: “on the weekend” and 20. The return value of the method is stored to a String also named ‘str’ which is printed in the following line, but there is no conflict here because the ‘str’ parameter is in a different location than the ‘str’ in the main method. In the method we have an if condition checking if ‘s’ is less than 10. This statement is false since our given ‘s’ was 20 and 20 is not less than 10. With the first condition being false the else block is executed. Inside of the else block we have a nested if statement which checks if the length ‘str’ is more than ‘s’. The length of the ‘str’ is only 8 and 8 is not more than 20 so the next condition is checked. The else if condition checks if ‘s’ is more than 10 which is true and because 20 is more than 10. The code inside of that block is taking the substring of ‘str’ starting from the index 7 and reassigning the value back the ‘str’ variable. The resulting substring will be: *“**weekend**”*. There is nothing else after the if statements except for the return statement which will return the value of ‘str’ at that point, which is “***weekend***” and that value will be printed in the console by the code in the main method.

13) In this code snippet we just need to execute each line one at a time to see the final result. We can take a look at method8 and see that it accepts an int array ‘arr’ and is a void

method, so it does not return anything. Now we can take a look at the main method to see the array that we are passing as the argument for our method. We use Scanner to declare the size of an int array 'a'. Afterwards the for loop is used to assign the input values into the array as its elements: [7, 3, 8, 20, 14]. This array is used as the argument of method8. Now that we know the value of the array passed as the argument, we just need to execute the code inside of method8. We have a for each loop which will read all the elements of the given array. Inside of the loop we have an if statement what checks if each element in the array is divisibly by 2, giving no remainder. When the number is divisible by 2 evenly it is not printed. The output in the console would be **73** because those are the only number in the array that are not evenly divisible by 2.

14) In this question we have three methods total in the code. One main method, a method9 which accepts three Strings: 'one', 'two' and 'three' and returned a String and also a method named middle which accepts a String 's' and returns a char datatype. The main method uses a Scanner to read 3 inputs from the user, each input is stored as a String and used directly as an argument for method9. The resulting String from method9 is stored into a String 'str' which will be printed in the following line. Now going one step deeper we see method9 accepts the three strings and uses each of those strings are the arguments for the method middle. This means for each given String argument there will be a char returned that is then part of the return value of method9. Taking a look, the method middle we see that the middle character of each string is returned as the char value. This is down by taking the length of the String and dividing it by 2 and using that value as the index to read a character using the charAt method. Just one thing to note, if a String is even there is always two middle characters and the length divided by 2 number will always be the index of the 2nd middle character. So far, we talked about what is happening in the code snippet, but now let's execute each part.

'one' has a value of "lawn" which means when we do the length /2 we will get (4/2 = 2) and read the character at index 2 which is "w"

'two' has a value of "solar" which means when we do the length /2 we will get (5/2 = 2) and read the character at index 2 which is "l"

'three has a value of "activity" which means when we do the length /2 we will get (8/2 = 4) and read the character at index 4 which is "v"

Each of those char values is returned from the method middle and are all concatenated together in the method9 return statement ("" + 'w' + 'l' + 'v') which gives us a final return String value of: "wlv" which is the output of the code snippet.

15) Here we have some overloaded methods with different return types and values. Just from a glance you wouldn't be able to tell if it is valid or not, you should go through each line of execution to see what the result would be. Looking at the main method we have a double variable 'number' which is storing our total number from all the operations. This is the result executing each line:

- We start with the 'add' method that accepts a double parameter (3.4). The second overloaded 'add' method will be the one executed because the parameter is a double.

That method will return 2.5 which will be added to 'number' making the value 2.5 so far.

- The next 'add' method that accepts a String parameter ("dual") which means the third overloaded version of the 'add' method will be executed which will return 10 making our total of 'number' 12.5 so far.
- The next 'add' method that accepts an int parameter (parse method will return an int value) which means the first overloaded version of the 'add' method will be executed which will return 5 making our total of 'number' 17.5 so far.
- The last 'add' method that accepts a double parameter (parse method will return a double value) which means the second overloaded version of the 'add' method will be executed, which will return 2.5 making our total of 'number' **20.0** at the end.

16) In this snippet we have four overloaded methods named 'action'. These are the different parameter types: no parameters, int, String and boolean. Starting to look at the main methods let's see the output: We declare a int variable 'total' to start at 0.

    - The first method call will accept a boolean datatype of true. Because it is a boolean datatype the fourth 'action' overloaded method will be executed. The if statement will be false (!b → !true → false), so the else statement will run and return 10 making our 'total' value 10 so far.
    - The next method call will accept an int datatype having the value of (6). The first overload method will run and return (i *2 → 6*2 → 12) and adding to our 'total' making the value 22 so far.
    - The second to last method call accept a String so it uses the second overloaded version of the method is executed because the argument given was String. The method returns the length of the String, which was "false" in this line, so the method returns 5 and adds it to the value of 'total' making the value 27 at this point.
    - The last method call will also have a String argument so again the second overloaded version of the method is executed returning the length of the String. This time the String is "four" so 4 is returned and added to 'total' giving us a final value of **31.**

17) This statement is **true**, the wrapper classes are a way to represent the primitive datatype, but as objects instead of primitives. This is important when working with collections, like ArrayList, that cannot have primitive elements.

18) Autoboxing is taking a primitive datatype and automatically converting it to the Wrapper class object. Unboxing is automatically converting your Wrapper class object to a primitive datatype. So, the giving statement is **true**.

19) As mentioned before, collections, like ArrayList cannot store primitive datatypes. They can only have object type elements, so the given statement is **true**.

20) One key difference between the Arrays and ArrayList is the size. Arrays have a fixed size that cannot be changed after the array object is created. On the other hand, the ArrayList

objects' size is not fixed. Whenever a new element is added, and more space is needed the object will automatically expand to store more elements. The given statement is **true**.

21) Here we need to just keep track of each execution and determine how the ArrayList is changed. The key ideas here are how the methods of ArrayList are working in combination with indexes of the elements. The left side will show the code executed and the right side will show the ArrayList after that operation.

nums.add(4)	[4]
nums.add(2)	[4, 2]
nums.add(10)	[4, 2, 10]
nums.add(0,6)	[6, 4, 2, 10]
nums.add(-7)	[6, 4, 2, 10, -7]
nums.add(2,2)	**[6, 4, 2, 2, 10, -7]**

22) This code snippet is similar to question number 21, except this time the ArrayList has String elements instead of Integer elements. We will follow the same process we used above to see how the ArrayList changes after each line is executed.

strs.add("j")	[j]
strs.add("1")	[j, 1]
strs.add("i)	[j, 1, i]
strs.add("5")	[j, 1, i, 5]
strs.remove(1)	[j, i, 5]
strs.add("e")	[j, i, 5, e]
strs.add("i")	[j, i, 5, e, i]
strs.remove("i")	**[j, 5, e, i]**

23) This code snippet is taking the given ArrayList of Integers and checks some specific conditions in order to build up a String made up of ones and zeros. The loop goes from index 0 to the last index so all the elements will be checked in each iteration.

'i' value	Actions taken	's' value so far
0	The first if is false. Second if condition is true (numbers. get(i) == 2 → 2 == 2 is true)	1
1	The first if is true because 97 > 5, goes to next iteration	1
2	The first if is false. Second if condition is true (numbers. get(i) == 2 → 2 == 2 is true)	11
3	The first if is true because 56 > 5, goes to next iteration	11
4	The first if is true because 46 > 5, goes to next iteration	11

5	The first if is true because 73 > 5, goes to next iteration	11
6	The first if is true because 6 > 5, goes to next iteration	11
7	The first if is false. Second if condition is true (numbers. get(i) == 2 → 2 == 2 is true)	111
8	The first if is false. Second if condition is false (numbers. get(i) ==3 → 3 % 2 is false)	1110
9	The first if is true because 7 > 5, goes to next iteration	**1110**

The final value is **1110** after going through each step

24) Here we have an ArrayList of Strings 'words' with some initial Strings. We also have a Integer ArrayList called 'lengths' which is empty in the beginning. We use a for each loop to read all the elements from 'words' and calling the length() method on each element of 'words'. Each length number is then stored into the 'lengths' ArrayList giving a final ArrayList of **[6, 5, 4, 4, 6, 6, 4]**

25) The idea in this snippet is the same as the first two code analysis questions about ArrayList. We need to just keep track of the ArrayList during each execution. We have an ArrayList 'numsOne' with some values that we will iterate through and another empty ArrayList 'numsTwo'. The for each loop is used to iterate through the 'numsOne' elements as shown below

Case 4: break, nothing to run	[ ]
Case 1: adds 1, adds 50, break	[1, 50]
Case 8: removes index 0, break	[50]
Case -42: adds 0 at index 1, break	[50, 0]
Case 2: adds 20000, break	[50, 0, 20000]
Case 10: adds 0 at index 1, break	**[50, 0, 0, 20000]**