› Answers and Explanations

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is C.

    - A class is a blueprint for constructing objects of the type described by the class definition.

2. The answer is A.

    - When two strings are concatenated (added together), the second one is placed right after the first one (do not add a space).

```
newString = anotherString + "Hello"
          = "Home" + "Hello" = "HomeHello"
```

3. The answer is D.

- This question requires that you understand the two uses of the + symbol.
- First, execute what is in the parentheses. Now we have:

```
System.out.println(7 + 8 + 15 + "Hello" + 7 + 8 + 15);
```

- Now do the addition left to right. That's easy until we hit the string:

```
System.out.println(30 + "Hello" + 7 + 8 + 15);
```

- When you add a number and a string in any order, Java turns the number into a string and then concatenates the string:

```
System.out.println("30Hello" + 7 + 8 + 15);
```

- Now every operation we do will have a string and a number, so they all become string concatenations, not number additions.

```
System.out.println("30Hello7815");
```

4. The answer is B.

- The `indexOf` method returns the index location of the parameter.
- The letter "a" has index 9 in "presto chango" (remember to start counting at 0).
- The substring will start at index 9 and stop *before* index 10, meaning it will return only the letter at index 9 in "abracadabra".
- Once again, start counting at 0. Index 9 of "abracadabra" is "r".
- Concatenate that with i which equals 9. Print "r9".

5. The answer is C.

- `compareTo` takes the calling string and compares it to the parameter string. If the calling string comes *before* the parameter string, it returns a negative number. If it comes *after* the parameter string, it returns a positive number. If they are equal, it returns 0.
- The first `compareTo` is "avocado".compareTo("banana") so it returns -1.
- The second `compareTo` switches the order to "banana".compareTo("avocado") so it returns 1.
- Note that although we often only care about whether the answer is positive or negative, the actual number returned has meaning. Because "a" and "b" are one letter apart, 1 (or -1) is returned. If the example had been "avocado" and "cucumber", 2 (or -2) would have been returned.

6. The answer is D.

- Let's take the word "cat" as an example. It has a length of 3, but the indices of its characters are 0, 1, 2, so the last character is at length − 1.
- We need to start our substring at `str.length() - 1`. We could say:

```
str.substring(str.length() - 1, str.length());
```

but since we want the string all the way to the end we can use the single parameter version of substring.

7. The answer is C.

- The basic form for generating a random int between high and low is:

```
int result = (int)(Math.random() * (high - low + 1)) + low;
```

- Our "high" is 27, and our "low" is 13, so the answer is C.
- Be sure to place the parentheses correctly. They can go after the multiplication or after the addition, but not after the `Math.random()`.

8. The answer is D.

- This problem doesn't have the necessary parentheses discussed in the explanation for problem 7.
- As a result, the (int) cast will cast only the value of `Math.random()`. Since `Math.random()` returns a value between 0 and 1 (including 0, not including 1), truncating will *always* give a result of 0.
- Multiplying by 1 still gives us 0, and adding 10 gives us 10.

9. The answer is A.

- This problem is similar to problem 7.
- Fill in (high − low + 1) = (8 − (-23)) + 1 = 32 and low = -23 and the answer becomes `(int)(Math.random() * 32 - 23).`
- Notice that this time the parentheses were placed around the whole expression, not just the multiplication.

10. The answer is B.

-
```
(int)(Math.PI * Math.pow(r, 2))
    = (int)(3.1415926... * 10000)
    = (int)(31415.926...)
    = 31415
```

11. The answer is B.

- `ex1.substring(1, ex1.indexOf("p"))` starts at the character at index 1 (remember we start counting at 0) and ends before the index of the letter "p". ex1 now equals "xam".
- Let's take this one substring at a time. At the beginning of the statement, `ex2 = "elpmaxe"`.
  - `ex2.substring(3, ex1.length())` The length of ex1 is 3, so this substring starts at the character at index 3 and ends *before* the character at index 3. It stops before it has a chance to begin and returns the empty string. Adding that to the current value of ex2 does not change ex2 at all.
  - `ex2.substring(3, ex2.length())` starts at the character at index 3 and goes to the end of the string, so that's "maxe".
  - Add that to the current value of ex2 and we have "elpmaxemaxe".
- We are going to take two substrings and assign them to ex3.
  - The first one isn't too bad. `ex1 = "xam"`, so `ex1.substring(1) = "am"`. Remember that if substring only has 1 parameter, we start there and go to the end of the string.
  - The second substring has two parameters. Let's start by figuring out what the parameters to the substring are equal to.
  - Remember ex2 = "elpmaxemaxe" so indexOf("x") is 5.
  - The second parameter is `ex2.length() - 2`. The length of ex2 = 11, subtract 2 and the second parameter = 9.
  - Now we can take the substring from 5 to 9, or from the first "x" to just before the second "x", which gives us "xema".
  - Add the two substrings together: "amxema".

12. The answer is A.

- Let's take this piece by piece. A forward slash "/" does not require an escape sequence, but a backslash needs to be doubled since "\" is a special character. Our print sequence needs to start: `/\\`
- The next interesting thing that happens is the tab before "time". The escape sequence for tab is `"\t"`. Our print sequence is now: `/\\what\ttime` (Don't be confused by the double t. One belongs to "\t" and the other belongs to "time".)
- Now we need to go to the next line. The newline escape sequence is `"\n"`

- We also need escape sequences for the quote symbols, so now our print sequence is:
```
/\\what\ttime\n\"is it\"
```
- Add the last few slashes, remembering to double the "\" and we've got our final result.
```
/\\what\ttime\n\"is it?\"//\\\\
```

13. The answer is B.

- This problem is like problem 7 only backwards. If (high − low + 1) = 5 and low = 3, then high = 7. The first line gives us integers from 3 to 7 inclusive.
- The second line squares all those numbers, giving us: 9, 16, 25, 36, 49.

14. The answer is E.

- Taking it line by line, val1 = 10, val2 = 7 so:
```
val1 = (val1 + val2) % (val1 - val2) = 17 % 3 = 2
val2 = 3 + val1 * val2 = 3 + 2 * 7 = 17
```
- The next statement becomes:
```
val3 = (int) (Math.random() * 17 + 2)
```
- We know low = 2 and (high − low + 1) = 17, so high = 18.
- val3 will contain integers from 2 to 18 inclusive.

15. The answer is C.

- The easiest way to solve this is to plug in examples. We need to make sure we test both positive and negative numbers, so let's use 2.9 and -2.9. The answer we are looking for in both cases is 3.
- Choice A:
```
(int) Math.abs(2.9 - 0.5) = (int) Math.abs(2.4) = (int) 2.4 = 2
```
NO (but interestingly, this works for -2.9).
- Choice B:
```
(int) Math.abs(2.9 + 0.5) = (int) Math.abs(3.4) = (int) 3.4 = 3
```
YES, but does it work in the negative case?
```
(int) Math.abs(-2.9 + 0.5)= (int) Math.abs(-2.4) = (int) 2.4 = 2
```
NO, keep looking.
- Choice C:
```
(int)(Math.abs(2.9) + 0.5) = (int)(2.9 + 0.5) = (int) 3.4 = 3
```
YES, but does it work in the negative case?
```
int)(Math.abs(-2.9) + 0.5) = (int)(2.9 + 0.5) = (int) 3.4 = 3
```
YES, found it!

16. The answer is E.

- Options A and B are incorrect. They use the math notation 4ac and 2a for multiplication instead of 4 * a * c and 2 * a.
- Solution C is incorrect. This option divides by 2 and then multiplies that entire answer by a, which is not what we want. 2 * a needs to be in parentheses.
- Look carefully at D and E. What's the difference? The only difference is the placement of the closing parenthesis, either after the (b, 2) or after the 4 * a * c. Which should it be? The square root should include both the Math(b, 2) and the 4 * a * c, so the answer is E.