**Java String class**

Q1. What will be the output of the following code snippet?

String s = " Hello ";

s += " World ";

s.trim( );

System.out.println(s);

A1. The output will be

|  |  |
| --- | --- |
| 1 | " Hello  World " |

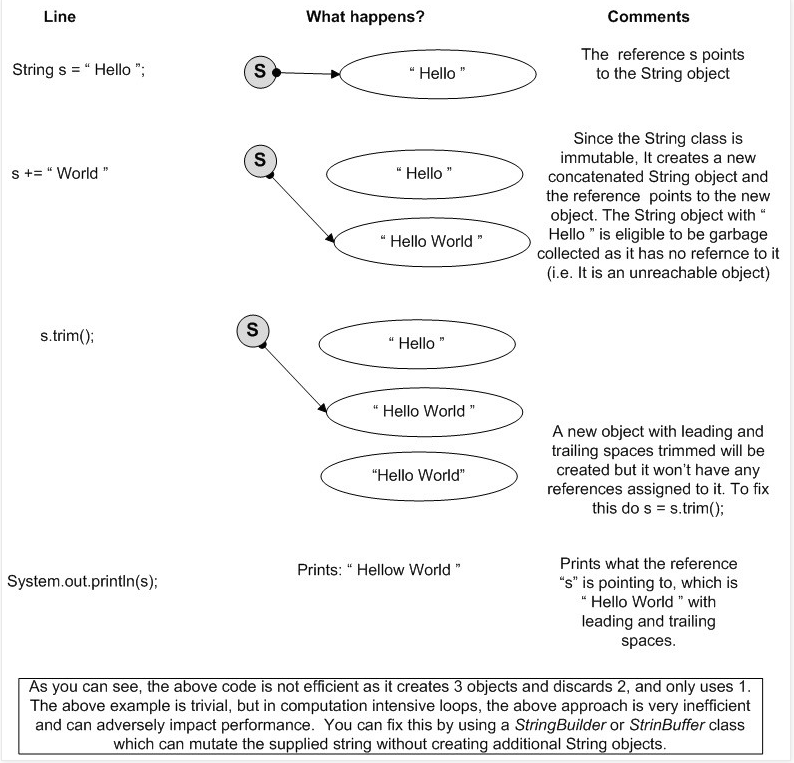
with the **leading and trailing spaces**. Some would expect a trimmed “Hello World”. So, **what concepts does this question try to test?**

1. String objects are immutable and there is a trick in s.trim( ) line.
2. Concept of object references and unreachable objects that are eligible for garbage collection. 3 String objects are created, and 2 of them become unreachable as there are no references to them, and gets garbage collected.

**What follow on questions can you expect?**

1. You might get a follow on question on how many string objects are created in the above example and when will it become an unreachable object to be garbage collected.
2. You might also be asked a follow on question as to if the above code snippet is efficient.

The best way to explain this is via a self-explanatory diagram as shown below. Click on it to enlarge.

[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-11-at-9.40.00-PM.png)

No of String objects created

If you want the above code to output “Hello World” with leading and trailing spaces trimmed then assign the s.trim( ) to the variable “s”. This will make the reference “s” to now point to the newly created trimmed String object.

The above code can be rewritten as shown below

|  |  |
| --- | --- |
| 1  2  3  4 | StringBuilder sb = new StringBuilder(" Hello ");  sb.append(" World ");  System.out.println(sb.toString().trim( )); |

Q2. What is the main difference between *String*, *StringBuffer*, and *StringBuilder*?  
A2.

* **String** is **immutable** in Java, and this immutability gives the benefits like security and performance discussed above.
* **StringBuffer** is **mutable**, hence you can add strings to it, and when required, convert to an immutable String with the toString( ) method.
* **StringBuilder** is very similar to a *StringBuffer*, but *StringBuffer* has one disadvantage in terms of performance as all of its public methods are synchronized for thread-safety. *StringBuilder* in Java is a copy of *StringBuffer* but without synchronization to be used in local variables which are inherently thread-safe. So, if thread-safety is required, use StringBuffer, otherwise use StringBuilder.

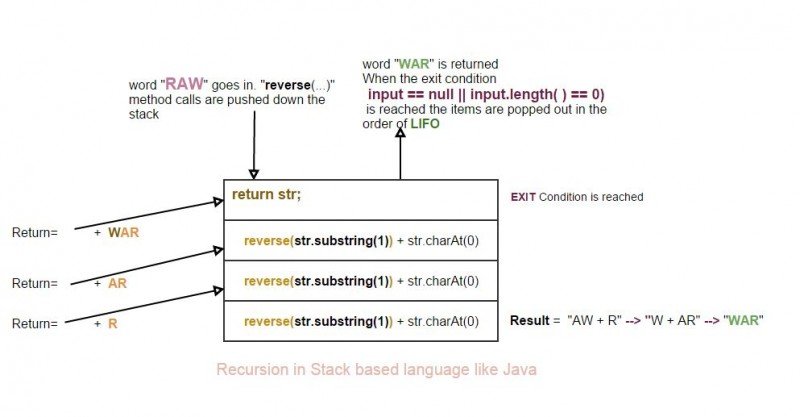
Q3. Can you write a method that reverses a given String?   
A3. A popular Java interview coding question.

**Example 1**: It is always a best practice to reuse the API methods as shown below with the *StringBuilder(input).reverse( )* method as it is fast, efficient (uses bit wise operations) and knows how to handle Unicode surrogate pairs, which most other solutions ignore. The  code shown below handles null and empty strings, and a StringBuilder is used as opposed to a thread-safe StringBuffer, as the StringBuilder is locally defined, and local variables are implicitly thread-safe.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static String reverse(String input) {        if(input == null || input.length( ) == 0){          return input;      }    return new StringBuilder(input).reverse( ).toString( );    } |

**Example 2**: Some interviewers might probe you to write other lesser elegant code using either recursion or iterative swapping. Some developers find it very difficult to handle recursion, especially to work out the termination condition. All recursive methods need to have a condition to terminate the recursion. **Recursive** solution.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public String reverse(String str) {      // exit or termination condition      if ((null == str) || (str.length( )  <= 1)) {          return str;      }        // put the first character (i.e. charAt(0)) to the end. String indices are 0 based.      // and recurse with 2nd character (i.e. substring(1)) onwards      return reverse(str.substring(1)) + str.charAt(0);  } |

[](https://www.java-success.com/wp-content/uploads/2014/08/Java-recursion-using-string.jpg)

Java Recursion – String example

**Step 1:** reverse(“RAW”)

**Step 2:** **reverse(AW)** + “R”      [**Note:**charAt[0] = “R”, and str.substring(1) = “AW” ]

**Step 3:** **reverse(W)** + “A” + “R”      [**Note:**charAt[0] = “A”, and str.substring(1) = “W” ]

**Step 4:** return “W” + “A” + “R”      [**Exit condition is reached** when “str.length( ) <=1” ]

**outputs:** “WAR”

**Example 3**: Iterative solution.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public String reverse(String str) {      // validate      if ((null == str) || (str.length( )  <= 1)) {          return str;      }        char[ ] chars = str.toCharArray( );      int rhsIdx = chars.length - 1;        //iteratively swap until exit condition lhsIdx < rhsIdx is reached      for (int lhsIdx = 0; lhsIdx < rhsIdx; lhsIdx++) {          char temp = chars[lhsIdx];          chars[lhsIdx] = chars[rhsIdx];          chars[rhsIdx--] = temp;      }        return new String(chars);  } |

Q4. Can you remember a design pattern discussed in this post?  
A4. **Flyweight design pattern**. The flyweight design pattern is a structural pattern used to improve **memory usage**(i.e. due to fewer objects and object reuse) and **performance** (i.e. due to shorter and less frequent garbage collections).

Q5. Can you give some examples of the usage of the flyweight design pattern in Java?  
A5.

**Example 1**: As discussed above, String objects are managed as flyweight. Java puts all fixed String literals into a literal pool. For redundant literals, Java keeps only one copy in the pool.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | String author = "Little brown fox";  String authorCopy = "Little brown fox";    //only 1 String object is created. Both  author and authorCopy point to that  if(author == authorCopy) {          System.out.println("referencing the same object");  } |

**Example 2**: The Wrapper classes like *Integer*, *Float*, *Decimal*, *Boolean*, and many other classes like *BigDecimal* having the *valueOf***static factory method** to apply the flyweight design pattern to conserve memory by reusing the objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class FlyWeightWrapper {    public static void main(String[] args) {      Integer value1 = Integer.valueOf(5);      Integer value2 = Integer.valueOf(5);        //only one object is created      if (value1 == value2) {          System.out.println("referencing the same object");      }    }    } |

If you use new Integer(5), a new object will be created every time.

Both the above examples will print “**referencing the same object**“.

Q6. What is a static factory method, and when will you use it?  
A6. The factory method pattern is a way to encapsulate object creation. It has the benefits like

**1.** Factory can choose what to return from many subclasses or implementations of an interface. This allows the caller to specify the behavior desired via parameters, without having to know or understand a potentially complex class hierarchy. The lesser a caller knows about a callee’s internal details, the more **loosely coupled** a callee is from the caller.

**2.** The factory can apply the fly weight design pattern to **cache objects** and return cached objects instead of creating a new object every time. In other words, objects can be pooled and reused. This is the reason why you should favor using *Integer.valuOf(6)*as opposed to *new Integer(6)*.

**3.** The factory methods have **more meaningful names**than the constructors. For example, *getInstance( )*, *valueOf( )*, *getConnection( )*, *deepCopy( )*, etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static List<Car> deepCopy(List<Car> listCars) {       List<Car> copiedList = new ArrayList<Car>(10);       for (Car car : listCars) {                          //JDK 1.5 for each loop           Car carCopied = new Car( );               //instantiate a new Car object           carCopied.setColor((car.getColor( )));           copiedList.add(carCopied);      }      return copiedList;  } |

Q7. How will you split the following string of text into individual vehicle types?

“Car,Jeep, Wagon Scooter Truck, Van”

A7. Regular expressions to the rescue.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class String3 {      public static void main(String[ ] args) {            String pattern = "[,\\s]+";  //regex pattern – a comma or white space repeated 1 or more times            String vehicles = "Car,Jeep, Wagon   Scooter        Truck, Van";          String[ ] result = vehicles.split(pattern);          for (String vehicle : result) {              System.out.println("Vehicle = \"" + vehicle + "\"");          }      }  } |

Q8. What are the different ways to concatenate strings? and which approach is most efficient?  
A8.

**Plus (“+”) operator**:

|  |  |
| --- | --- |
| 1  2 | String s1 = ”John” + “Davies”; |

Using a **StringBuilder** or **StringBuffer** class.

|  |  |
| --- | --- |
| 1  2  3 | StringBuilder sb = new StringBuilder(“John”);  sb.append(“Davies”); |

Using the **concat(…)** method.

|  |  |
| --- | --- |
| 1  2 | “John”.concat(“Davies”); |

The efficiency depends on what you are concatenating and how you are concatenating it.

Concatenating constants: Plus operator is more efficient than the other two as the JVM optimizes constants.

|  |  |
| --- | --- |
| 1  2 | String s1 = ”John” + “Davies”; |

Concatenating String variables: Any one of the three methods should do the job.

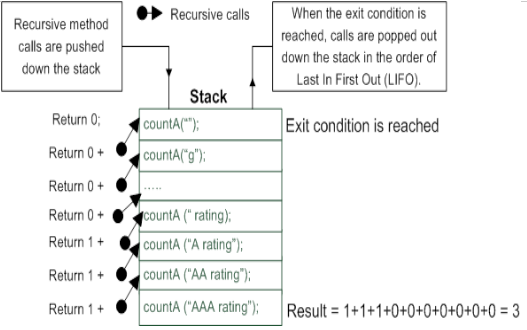
|  |  |
| --- | --- |
| 1  2  3  4 | String s1 = s2 + s3 + s4;  String s1 =  “name=”;  s1 += name; |

Concatenating in a for/while loop: ***StringBuilder*** or ***StringBuffer*** is the most efficient. Avoid using plus operator as it is the worst offender.

|  |  |
| --- | --- |
| 1  2  3  4  5 | StringBuilder sb = new StringBuilder(250);  for( int i=0; i<SIZE; i++ ) {        sb.append(“Item:” + i);  } |

Prefer StringBuilder to StringBuffer unless multiple threads can have access to it.

Q9. Java being a stack based language, allows you to make recursive method calls. Can you write a recursion based solution to count the number of A’s in string “AAA rating”?  
A9. A function is recursive if it calls itself. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive functions are useful in removing iterations from many sorts of algorithms.

[](https://www.java-success.com/wp-content/uploads/2014/08/Screen-shot-2014-08-16-at-8.24.49-PM.png)

Recursion in stack based language like Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class RecursiveCall {        public int countA(String input) {            // exit condition – recursive calls must have an exit condition          if (input == null || input.length( ) == 0) {              return 0;          }            int count = 0;            //check first character of the input          if (input.substring(0, 1).equals("A")) {              count = 1;          }            //recursive call to evaluate rest of the input          //(i.e.  2nd character onwards)          return count + countA(input.substring(1));      }        public static void main(String[ ] args) {          System.out.println(new RecursiveCall( ).countA("AAA rating"));    // 3      }  } |

Recursion might not be the efficient way to code, but recursive functions are shorter, simpler, and easier to read and understand. Recursive functions are very handy in working with tree structures and avoiding unsightly nested for loops.

Learn more at [Recursion Vs. Tail Recursion](https://www.java-success.com/recursion-vs-tail-recursion/).

**Bonus Java String Q&A**

Q10. How do you stream a string class in Java 8? ★ ♟  
A10. **chars()** method.

|  |  |
| --- | --- |
| 1  2  3 | public static void main(String[] args) {      "cactus".chars().forEach(c -> System.out.println((char)c));  } |

**Q.** Does parallel processing as shown below preserve the order?

|  |  |
| --- | --- |
| 1  2  3 | public static void main(String[] args) {      "cactus".chars().parallel().forEach(c -> System.out.println((char)c));  } |

**A.** No.