L44

Friday, February 4, 2022 10:05 AM

Constructors and finalize method, Garbage Collection, Understanding the internals of String, StringBuilder Class

Garbage Collection

Q1. In memory management terminology, garbage means that portion of memory which was once occupied by objects and is currently no longer used by the program.

Garbage collection (GC) means reclaiming such memory so that, that space can be used for allocation to other objects.

In Java programming language, the JVM (Java Virtual Machine) which is responsible for running the Java application performs automatic garbage collections.

The automatic garbage collection in Java is performed by a special thread called Garbage Collector, which is a part of JVM.

Garbage Collector does not reclaim the memory of objects that are still in use. It only reclaims the memory of objects which are no longer in use.

An object is created and assigned memory when we use the new keyword followed by a constructor call.

The reference holds the address of the object in the memory.3+

That portion of memory in which the Java objects reside is called **heap**.

When the program no longer uses an object, such an object is called unreferenced object.

For example:

```
public class A {
	private String text1 = "Bali Islands";		// statement 1public String haveSomeFun() {
		String text2 = "Fly to " + text1;	// statement 2System.out.println(text2);		// statement 3}
}
public class ATest {
	public static void main(String[] args) {
		A a = new A();				// statement 4a.haveSomeFun();			// statement 5System.out.println("Happy holidays!");	// statement 6}
}
```

In the above code, after **statement 4** is executed, we have reference a pointing to an object created in the **heap**.

Similarly, inside the instance pointed by a, we have another reference text1 (which is declared in statement 1) also pointing to a String object in memory.

However, note that the object referenced by text2 declared in the **statement 2** is not yet created in memory.

The reference text2 will become alive only during the execution of the method invocation on reference a in **statement 5**.

After executing the statement 5, when the JVM is executing **statement 6**, the reference text2, whose scope was local to the method haveSomeFun() is no longer alive even though the object is in the memory. Such objects are called unreferenced objects.

Garbage collector identifies such objects (which are there in memory but are no longer referenced) and reclaims their memory.

Select all the correct statement given below.

- [ ] System.gc() method can be called to force garbage collection.

- [x]
```
Object a1 = new Object(); //statement 1
Object a2 = new Object(); //statement 2
a2 = a1;
```
In the above code, the initial object referred by `a2` in **statement 2** will be available for **GC** after the statement `a2 = a1;` is executed.

Constructors and finalize method

Q1. In Java, the constructors are used to prepare a newly created object for use by initilaizing values passed to it as parameters.

Similarly, when the GC (Garbage Collector) decides to remove an object from memory, it calls the finalize() method on the object.

The finalize() method is declared in the Object class. Hence it is available in every class.

The default finalize() method available in the Object class does not do anything.

A Java class can override and provide its special implementation in the finalize() method.

Normally we do not override finalize to provide any special implementation. However, it is good to know that we do not call the finalize() method, it is the GC which calls it.

- [ ] Constructors are automatically called by the **GC** (Garbage Collector Thread) to create instances of new classes.
- [x] **GC** (Garbage Collector Thread) automatically calls the finalize method of the object whenever it is trying to reclaim the memory occupied by the object.

String and StringBuilder Class

Q1. In Java, we have learnt that String class represents an immutable sequence of characters.

For example:
String text = "I am back!";
text = "I am back, again!";
In the above code, you will notice that the reference text can be assigned to any other String literal but the previous String literal itself cannot be modified.

It means that once a String object is created, the contents of that object (meaning the sequence of characters in that object) cannot be modified.

Such classes whose objects cannot be modified are called **immutable classes.**

A class becomes **immutable** when it does not provide any methods to manipulate the state information stored in its fields.

The String class does not have even a single method which can manipulate or change the sequence of characters it represents. The methods like substring, replace, toUpperCase, toLowerCase etc., do not change the contents of the existing string, instead these methods create new string objects with the modified contents and return their references.

The StringBuilder class represents a mutable sequence of characters. Unlike the String class, the StringBuilder class provides methods to change the sequence of characters it represents.

Below are some of the most commonly used methods in StringBuilder class which help to manipulate the sequence of characters it represents:
1. append(String str)
2. insert(int index, String str)
3. delete(int start, int end)
4. replace(int start, int end, String str)

See and retype the below code.

Below are the three important points you should note in the code:

You will notice that we are using a StringBuilder constructor which accepts an String reference. StringBuilder has many constructors.

You will also notice that we are **chaining** the append() method call in the statement sb.append("River").append("Thames");. This is possible because append method call on a StringBuilder object returns a reference to the same StringBuilder object.

A StringBuilder object can be converted to a String by calling the toString() method. The toString() method creates and returns a new String object with the sequence of characters present inside the StringBuilder object. You will notice this happening whenever the statement System.out.println("sb = " + sb); is executed.

```
1  package q11307;
2  public class StringBuilderDemo {
3      public static void main(String[] args) {
4          StringBuilder sb = new StringBuilder("Ganga");
5          sb.append("River");
6          System.out.println("sb = " + sb);
7          sb.append("Nile");
8          System.out.println("sb = " + sb);
9          sb.append("River").append("Thames");
10         System.out.println("sb = " + sb);
11         sb.delete(0, 5);
12         System.out.println("sb = " + sb);
13     }
14 }
15
```

Q2. StringBuilder objects are like String objects, except that they can be modified, means internally these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

String builders offer certain advantages as they offer better performance and simpler code when compared to strings. For example, if you need to concatenate a large number of strings, appending to a StringBuilder object is more efficient.

Length and Capacity: The StringBuilder class, like the String class, has a method length() that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a capacity, the number of character spaces that have been allocated. The capacity, which is returned by the capacity() method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

StringBuilder Constructors:

- **StringBuilder()**: Creates an empty string builder with a capacity of 16 (16 empty elements).
- **StringBuilder(CharSequence cs)**: Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.
- **StringBuilder(int initCapacity)**: Creates an empty string builder with the specified initial capacity.
- **StringBuilder(String s)**: Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string

- [x] Initial capacity of string builder is 16
- [x] String builders are mutable.
- [ ] Strings provide better performance than string builders.

Q3. The principal operations on a StringBuilder that are not available in String are the **append()** and **insert()** methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder.

The append method always adds the characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

There are a number of methods of the StringBuilder class. These are few examples

- **StringBuilder append(Object obj)** Appends the argument to this string builder. The data is converted to a string before the append operation takes place.
- **StringBuilder delete(int start, int end)** This method deletes the subsequence from start to end-1 (inclusive) in the StringBuilder's char sequence
- **StringBuilder insert(int offset, int i)** Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place.

StringBuilder reverse(): Reverses the sequence of characters in this string builder. **String toString()**: Returns a string that contains the character sequence in the builder.

- [ ] append() method always adds the character at the beginning.
- [x] append() and insert() operations are not available in strings
- [x] insert() method adds the character at the specified index

Q4. Given

```
public class Main {
        public static void main(String args[]) {
            StringBuilder sb = new StringBuilder("Hi! Good Morning.");
            System.out.println(sb.length());
        }
}
```

Choose the correct output for the above program form the below options.

- [x] 17
- [ ] 16
- [ ] 15
- [ ] 18

Q5. Given

```
public class Main {
        public static void main(String args[]) {
            StringBuilder sb = new StringBuilder("Hi! Good Morning.");

        }
}
```

Write an expression that refers to the letter M in the string referred to by sb. Choose the correct option from the below.

- [ ] sb.charAt(8)
- [x] sb.charAt(9)
- [ ] sb.charAt(5)
- [ ] sb.charAt(11)

Q6. The program has a class Example with the main method. The program takes input from the command line argument. Print the output by appending all the capital letters in the input.

Sample Input and Output:
Cmd Args : HYderaBad
The result is: HYB

```java
// Type Content here...
package q24212;

public class Example {

    public static void main(String[] args) {

        String s = args[0];

        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < s.length(); i++) {

            char c = s.charAt(i);

            if (Character.isUpperCase(c)) {

                sb.append(c);

            }

        }

        System.out.println("The result is: " + sb.toString());

    }

}
```

Q7. The below code is used to understand the difference between String and StringBuilder objects. When we try to concatenate two strings using string operations a new object is created without changing the old one. In StringBuilder existing object is modified. In the below program this can be illustrated by comparing Hash Code for String object after every concat operation. Fill the missing code in the below program and observe the output.

Sample Input and Output:
In Strings before concatenation Hash Code is: 2081
In Strings after concatenation Hash Code is: 64578
In StringBuilder before concatenation Hash Code is: 321001045
In StringBuilder after concatenation Hash Code is: 321001045

```java
package q24216;

public class StringBuilderDemo {
    public static void main(String args[]) {
        String s = new String("AB");
        System.out.print("In Strings before concatenation Hash Code is: ");
        System.out.println(s.hashCode());
        s += "C";
        // print hash code after concatenating
        StringBuilder sb = new StringBuilder("AB");
        System.out.println("In Strings after concatenation Hash Code is: ");
        // print hash code before concatenating

        System.out.print("In StringBuilder before concatenation Hash Code is: ");

        System.out.println(sb.hashCode());
        sb.append("C");

        // print hash code after concatenating

        System.out.print("In StringBuilder after concatenation Hash Code is: ");

        System.out.println(sb.hashCode());
        // print hash code before concatenating
        // add string C to AB
        // print hash code after concatenating
        // and observe the output

    }
}
```

Q8. Given

```
public class StringBuilderDemo {
        public static void main(String args[]) {
                String s1 = new String("ABC");
                String s2 = new String("ABC");
                System.out.println(s1.equals(s2));
                StringBuilder sb1 = new StringBuilder("ABC");
                StringBuilder sb2 = new StringBuilder("ABC");
                System.out.println(sb1.equals(sb2));
        }
}
```

What will be the output for the above program. Choose the correct option form the below.

- [ ] true
  true
- [ ] false
  true
- [x] true
  false
- [ ] false
  false

Q9. Given

```
public class StringBuilderDemo {
        public static void main(String args[]) {
    String s = "Hello";
    s.concat("World");
    System.out.println(s);
        }
}
```

What will be the output for the above program. Choose the correct option form the below.

- [ ] Hello World
- [x] Hello
- [ ] World
- [ ] hello

Q10. Given

```
public class StringBuilderDemo {
        public static void main(String args[]) {
        StringBuilder sb = new StringBuilder("Hello ");
    sb.append("World");
    System.out.println(sb);
        }
}
```

What will be the output for the above program. Choose the correct option form the below.

- [ ] HelloWorld
- [x] Hello World
- [ ] World
- [ ] Hello

String concatenation and memory

Q1. The Java compiler apart from compiling the source code into **.class** files, at times also performs certain optimizations.
For example, the below code

```
String text1 = "I ";
String text2 = "AM ";
String text3 = "THAT ";
String text4 = "I AM";
String fullText = text1 + text2 + text3 + text4;
```

is optimized by the Java compiler as

```
String text1 = "I ";
String text2 = "AM ";
String text3 = "THAT ";
String text4 = "I AM";
String fullText = new StringBuilder().append(text1).append(text2).append(text3).append(text4).toString();
```

By doing this the compiler ensured that during the runtime, a single StringBuilder object is created in memory and used to concatenate all the four String objects.

However when we perform concatenation in loops, the optimization may not be as efficient. For example:

```
String finalText = "";
for (String text : reallyBigStringArr) {
        finalText = finalText + text;
}
```

will be optimized by the compiler as below code

```
String finalText = "";
for (String text : reallyBigStringArr) {
        finalText = new StringBuilder().append(finalText).append(text).toString();
}
```

The above code is not really optimized, since a new StringBuilder object is created in each iteration of the loop (which is eventually discarded).

In such cases we could have used StringBuilder as below:

```
StringBuilder sb = new StringBuilder();
for (String text : reallyBigStringArr) {
        sb.append(text);
}
String finalText = sb.toString();
```

See and retype the below code.

```java
package q11308;
public class ConcatenationDemo {
    public static void main(String[] args) {
        String[] wordsArr = {"I", "AM", "THAT", "I AM"};
        StringBuilder sb = new StringBuilder();
        for (String word : wordsArr) {
            sb.append(word).append(" ");
        }
        String theRealMe = sb.toString();
        System.out.println("theRealMe = " + theRealMe);
    }
}
```

String Buffer Constructors

Q1. The StringBuffer class is a thread-safe, mutable sequence of characters.These are different types of string buffer constructors. They are

- **public StringBuffer()**: Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
- **public StringBuffer(int capacity)**: Constructs a string buffer with no characters in it and the specified initial capacity. It throws NegativeArraySizeException if the capacity argument is less than 0
- **public StringBuffer(String str)**: Constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is 16 plus the length of the string argument. It throws NullPointerException if **str** is null
- **public StringBuffer(CharSequence seq)**: Constructs a string buffer that contains the same characters as the specified CharSequence. The initial capacity of the string buffer is 16 plus the length of the CharSequence argument. If the length of the specified CharSequence is less than or equal to zero, then an empty buffer of capacity 16 is returned. It throws NullPointerException if **seq** is null

- [x] Initial capacity of string buffer constructor is 16 characters.
- [ ] public StringBuffer(int capacity), if capacity is less than 0 it throws NullPointerException.
- [x] public StringBuffer(CharSequence seq), if seq is null it throws NullPointerExceotion.

Q2. The below program explains different StringBuffer constructors. Follow the comments given below and write the missing code.

The below program has a class StringbufferExample with main method. The program takes input from the command line arguments. Print the output as follows.

Sample Input and Output:
Cmd Args : Hello World
Initial capacity is: 16
Capacity after passing parameter is: 27
Creating a StringBuffer object with the given capacity: 50

```java
package q24215;

public class StringbufferExample {
    public static void main(String args[]) {
        // create instance of StringBuffer
        // find the initial capacity
        // find the capactiy after passing a parameter args[0] using command line
        // argument
        // find the capatity by intializing capatity to 50

        StringBuffer sb = new StringBuffer();

        System.out.println("Initial capacity is: " + sb.capacity());

        sb = new StringBuffer(args[0]);

        System.out.println("Capacity after passing parameter is: " + sb.capacity());

        sb = new StringBuffer(50);

        System.out.println("Creating a StringBuffer object with the given capacity: " + sb.capacity());
    }
}
```