**String Programming Questions**

1. What is the output of below program?
2. package com.journaldev.strings;
3. public class StringTest {
4. public static void main(String[] args) {
5. String s1 = new String("pankaj");
6. String s2 = new String("PANKAJ");
7. System.out.println(s1 = s2);
8. }

}

It's a simple yet tricky program, it will print "PANKAJ" because we are assigning s2 String to s1. Don't get confused with == comparison operator.

1. What is the output of below program?
2. package com.journaldev.strings;
3. public class Test {
4. public void foo(String s) {
5. System.out.println("String");
6. }
7. public void foo(StringBuffer sb){
8. System.out.println("StringBuffer");
9. }
10. public static void main(String[] args) {
11. new Test().foo(null);
12. }

}

The above program will not compile with error as "The method foo(String) is ambiguous for the type Test". For complete clarification read [Understanding the method X is ambiguous for the type Y error](http://www.journaldev.com/9107/the-method-is-ambiguous-for-the-type-java-ambiguous-method-call-null-error).

1. What is the output of below code snippet?
2. String s1 = new String("abc");
3. String s2 = new String("abc");

System.out.println(s1 == s2);

It will print **false** because we are using *new* operator to create String, so it will be created in the heap memory and both s1, s2 will have different reference. If we create them using double quotes, then they will be part of string pool and it will print true.

1. What will be output of below code snippet?
2. String s1 = "abc";
3. StringBuffer s2 = new StringBuffer(s1);

System.out.println(s1.equals(s2));

It will print false because s2 is not of type String. If you will look at the equals method implementation in the String class, you will find a check using **instanceof** operator to check if the type of passed object is String? If not, then return false.

1. What will be output of below program?
2. String s1 = "abc";
3. String s2 = new String("abc");
4. s2.intern();

System.out.println(s1 ==s2);

It's a tricky question and output will be **false**. We know that intern() method will return the String object reference from the string pool, but since we didn't assigned it back to s2, there is no change in s2 and hence both s1 and s2 are having different reference. If we change the code in line 3 to s2 = s2.intern(); then output will be true.

1. How many String objects got created in below code snippet?
2. String s1 = new String("Hello");

String s2 = new String("Hello");

Answer is 3.  
First - line 1, "Hello" object in the string pool.  
Second - line 1, new String with value "Hello" in the heap memory.  
Third - line 2, new String with value "Hello" in the heap memory. Here "Hello" string from string pool is reused.

Q=What is Permanent Generation (PermGen) ?

Q=cause of memory leak?

## Ans=**Mutable Static Fields and Collections**

Static fields are de facto GC roots (see the How Garbage Collection Works section earlier in this chapter), which means they are never garbage-collected! For convenience alone, static fields and collections are often used to hold caches or share state across threads. Mutable static fields need to be cleaned up explicitly. If the developer does not consider every possibility (a near certainty), the cleanup will not take place, resulting in a memory leak. This sort of careless programming means that static fields and collections have become the most common cause of memory leaks!

In short, never use mutable static fields—use only constants. If you think you need mutable static fields, think about it again, and then again! There’s always a more appropriate technique.

## **Thread-Local Variables**

A thread-local variable is basically a member field in the Thread class. In a multithreaded application, every thread instance has its own instance of such a variable. Therefore it can be very useful to bind a state to a thread. But this can be dangerous because thread-local variables will not be removed by the garbage collector as long as the thread itself is alive. As threads are often pooled and thus kept alive virtually forever, the object might never be removed by the garbage collector!

Since an active thread must be considered a GC root, a thread-local variable is very similar to a static variable. The developer, more specifically his code, needs to explicitly clean it up, which goes against the idea of a garbage collector. Just like a mutable static variable, it should be avoided unless there are very few good reasons to use it.

These kinds of memory leaks can be discovered with a heap dump. Just take a look at the ThreadLocalMap in the heap dump and follow the references (see Figure 2.26). Also look at the name of the thread to figure out which part of your application is responsible for the leak.

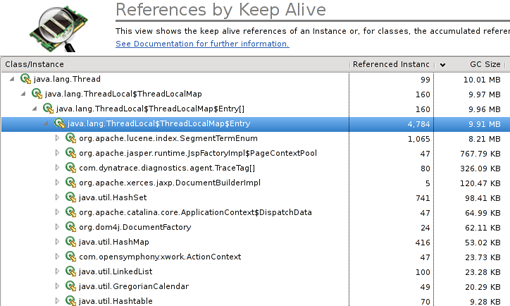


Figure 2.26: The heap dump shows more than 4K objects, which amount to about 10 MB, held by thread-locals.

## **Circular and Complex Bi-Directional References**

Memory leaks due to overly complex and circular object structures are my personal favorites, because it’s as if they’ve got their own personalities. The developer seems to be trying to trick the garbage collector so that it can’t do its job correctly.

Let me explain this particular problem by example:

org.w3c.dom.Document doc = readXmlDocument();

org.w3c.dom.Node child = doc.getDocumentElement().getFirstChild();

doc.removeNode(child);

doc = null;

The first code line reads an XML document from somewhere. The subsequent code, however, does not need the whole document, but rather just a specific portion (e.g. just the first child element). I know that the document holds a reference to the child document, so I remove it (removeNode) to allow the garbage collector to do its work. At the end of the code snippet one might think the DOM document will be garbage-collected; this is not the case!

A DOM Node object always belongs to a DOM Document. Even when removed from the document the node object retains a reference to its owning document. As long as we keep that child object, neither the document nor any of the nodes it refers to will be removed.

I’ve seen this and similar cases quite often, and they are rather hard to track down. If you do not know the code in question by heart (which allows you to see the problem just by reading the code), you need to do a heap analysis and figure out why the document object has not been garbage-collected.

## **JNI Memory Leaks**

Java Native Interface (JNI) memory leaks are particularly nasty and hard to find. JNI is used to call native code from Java. This native code can handle, call, and also create Java objects. Every Java object created in a native method begins its life as a local reference, which means that the object is referenced until the native method returns. We could say the native method references the Java object, so you won’t have a problem unless the native method runs forever. In some cases you want to keep the created object even after the native call has ended. To achieve this you can either ensure that it is referenced by some other Java object or you can change the local reference into a global reference. A global reference is a GC root and will never be removed until it is explicitly freed by the native code (see Figure 2.27).

The only way to discover JNI memory leaks is to use a heap-dump tool that explicitly marks native references. If possible, you should not use any global references. It’s better to assign the desired object to the field of a normal Java class.