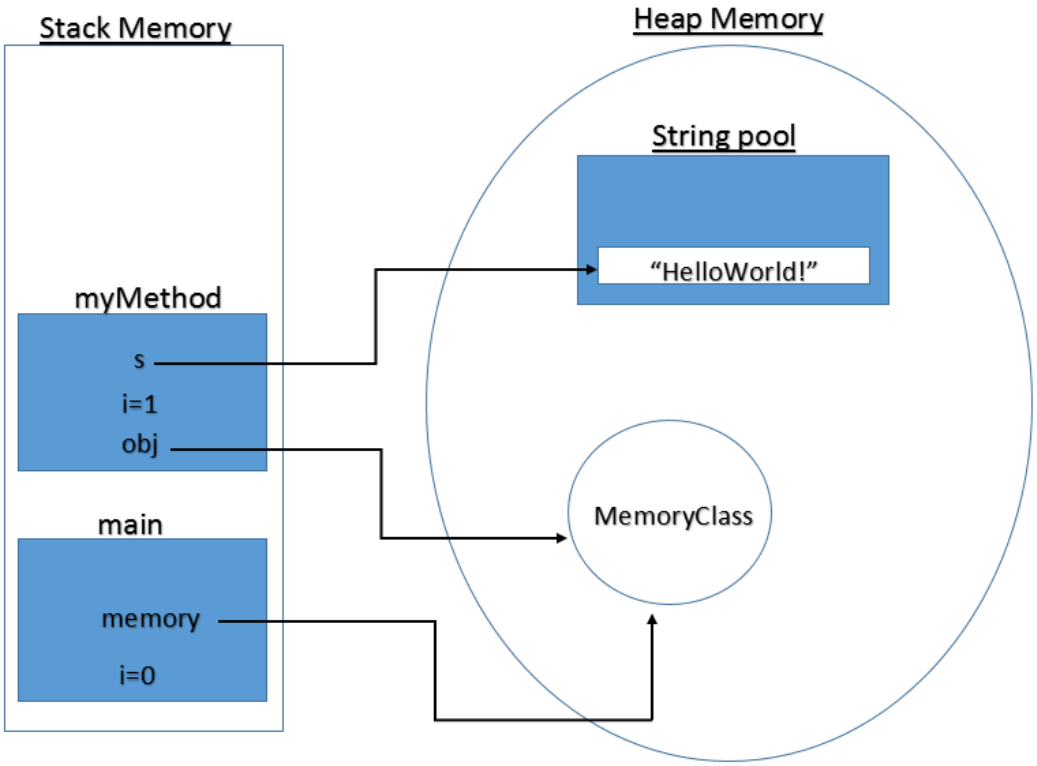
Java stack and heap memory management

Ask Question

I want to know how the memory is being allocated in the following program:

```
public class MemoryClass {

    public static void main(final String[] args) {
        int i = 0;
        MemoryClass memoryClass = new MemoryClass();
        memoryClass.myMethod(memoryClass);
    }

    private void myMethod(final Object obj) {
        int i = 1;
        String s = "HelloWorld!";

    }

}
```

Now, as far as my understanding goes, the following diagram describes how the memory allocation takes place:



In the above diagram, **memory**,**obj** and **s**, which are in the stack memory, are actually the references to their "**actual objects**" which are placed inside the heap memory.
Here is the set of questions that come to my mind:

1. Where are the methods of **s** stored?
2. Had I created another object of `MemoryClass` inside `myMethod` , would JVM allocate memory for the same methods again inside the stack memory?
3. Would JVM free the memory allocated to `myMethod` as soon as it's execution is completed, if so, how would it manage the situation mentioned in question 2(*only applicable if JVM allocates memory multiple times to the same method*).
4. What would have been the case, if I had only declared **s** and did not initialize it, would JVM still allocate memory to all the methods of `java.lang.String` class,if so,why?

java memory memory-management heap-memory stack-memory

asked Dec 13 '16 at 11:34

Adit A. Pillai
237 3 19

The nice thing about Java is how it *hides* all of the memory management from you and lets the garbage collector take control. Why do you need to go looking into it? – byxor Dec 13 '16 at 11:35

1 just out of curiosity – Adit A. Pillai Dec 13 '16 at 11:36

3 Answers

> Where are the methods of s stored?

They are stored in the String class object; it is an object loaded by a ClassLoader object when String is first referenced in the program. All the implementations of the JVM that existed when I read about this last never deallocated the memory for a class object once it was loaded. It's on the heap.

> Had I created another object of MemoryClass inside myMethod, would JVM allocate memory for the same methods again inside the stack memory?

No, methods and data for objects is kept separately, specifically because the JVM never needs more than one copy of the methods.

> Would JVM free the memory allocated to myMethod as soon as it's execution is completed, if so, how would it manage the situation mentioned in question 2(only applicable if JVM allocates memory multiple times to the same method).

No. Java doesn't generally "immediately free memory" of things stored on the heap. It would make things run too slowly. It only frees memory when the garbage collector runs, and it does that only when its algorithm for running the garbage collector decides it is time.

> What would have been the case, if I had only declared s and did not initialize it, would JVM still allocate memory to all the methods of java.lang.String class,if so,why?

This depends on the JVM implementation, I think, and maybe the compiler. If you declare a variable and never use it, it is quite possible (and common) for the compiler to notice that there is no use for it and to not put it into the class file. If it isn't in the class file, it is never referenced, and therefore it and its methods are not loaded, etc. If the compiler puts it in anyway but it is never referenced, then the ClassLoader wouldn't have any reason to load it, but I'm a little vague on whether it would get loaded or not. Might depend on the JVM implementation; does it load things because there are variables of the class or only when they are referenced? How many ClassLoader algorithms can dance on the head of a 4-digit PIN?

I encourage you to read about the JVM and ClassLoaders and such; you will gain so much more by reading an explanation of how it works rather than poking at it with examples you can think up.

edited Dec 13 '16 at 12:27

byxor
2,085 2 15 26

answered Dec 13 '16 at 11:54

arcy
8,922 4 35 69

why are the methods of the second object of the **MemoryClass** not stored on the heap just like **s**? – Adit A. Pillai Dec 14 '16 at 3:45

1 They are stored in the heap, but only one copy of them. Most talk of allocating memory for objects does not make the distinction, but when we talk of allocating memory for an instance of an object, it is understood that the system is only allocating additional memory for the data. There is no need to allocate memory for an identical copy of the methods. I don't remember for sure, but probably the methods are stored with the class object for that class, since only one copy of them is ever needed. (A different classloader will load a different class object, but that's beyond me to explain here). – arcy Dec 14 '16 at 11:50

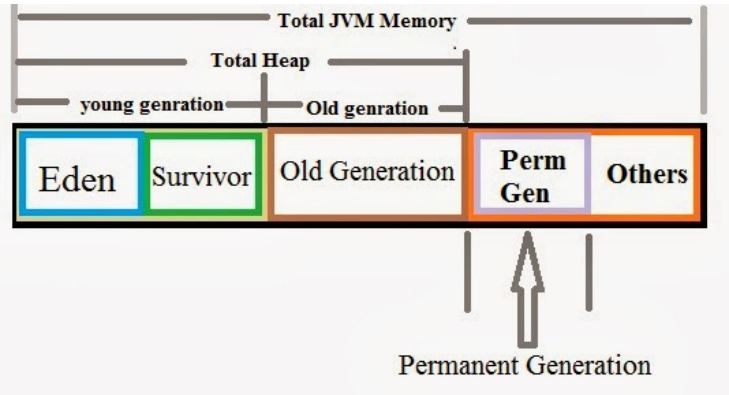
First things first: I am assuming your questions are coming out after reading this article (*because over there I see a very similar diagram as yours*) so I will not quote or highlight any of the points which are mentioned over there and will try to answer to your questions with points which were not so obvious in that post.

Reading to all your questions, my impression is that you are clear about how memory is allocated in stack and heap but have doubts about metadata of the classes i.e. where in the memory, methods of the classes would be stored and how they would be recycled. So, first let me try to explain JVM memory areas:

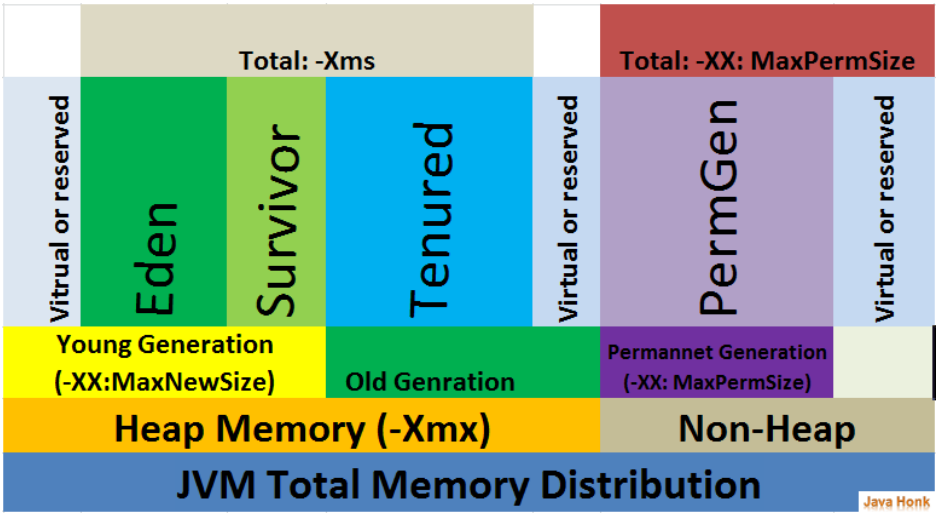
JVM Memory Areas

Let me start by putting these 2 diagrams depicting JVM memory areas:

Source of diagram



Source of diagram



Now, as clear from the above diagrams below is the tree structure of JVM memory and I will try to throw light on the same (**@Adit: please note that area which concerns you is PermGen Space or permanent generation space of non-heap memory**).

- **Heap memory**
  - Young generation
    - Eden Space
    - Survivor Space
  - Old generation
    - Tenured Generation
- **NonHeap memory**
  - Permanent Generation
  - Code Cache (*I think included "only" by HotSpot Java VM*)

Heap memory

Heap memory is the runtime data area from which the Java VM allocates memory for all class instances and arrays. The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.

Young generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called Minor GC. Young Generation is divided into below 2 parts

Eden space: The pool from which memory is initially allocated for most objects.

Survivor space: The pool containing objects that have survived the garbage collection of the Eden space.

Old generation

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it's full. Old Generation Garbage Collection is called Major GC and usually takes longer time. Old generation contains below part:

Tenured space: The pool containing objects that have existed for some time in the survivor space.

Non-Heap memory

Non-heap memory includes a method area shared among all threads and memory required for the internal processing or optimization for the Java VM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on the implementation, a Java VM may not garbage collect or compact it. Like the heap memory, the method area may be of a fixed or variable size. The memory for the method area does not need to be contiguous.

Permanent generation

The pool containing all the reflective data of the virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas.

Code cache

The HotSpot Java VM also includes a code cache, containing memory that is used for compilation and storage of native code.

Answering OP's questions specifically

> Where are the methods of s stored?

Non-Heap memory --> Permanent Generation

> Had I created another object of MemoryClass inside myMethod, would JVM allocate memory for the same methods again inside the stack memory?

Stack memory only contains local variables so your ORV (object reference variable) of new `MemoryClass` would still be created in stack frame of `myMethod`, but JVM would not load all the methods, metadata etc. of `MemoryClass` again in "Permanent Generation".

JVM loads class only once and when it loads the class then space is allocated on "Permanent Generation" for that class and that happens only once while the class is loaded by JVM.

> Would JVM free the memory allocated to myMethod as soon as it's execution is completed, if so, how would it manage the situation mentioned in question 2(only applicable if JVM allocates memory multiple times to the same method).

Stack frame created for `myMethod` will be removed from stack memory, so all the memory created for local variables will be cleaned but this doesn't mean that JVM will clean up the memory allocated in "Permanent Generation" for the class those object you have created in `myMethod`

> What would have been the case, if I had only declared s and did not initialize it, would JVM still allocate memory to all the methods of java.lang.String class, if so, why?

Specifically talking about `String` class, JVM would have allocated space for `String` in "Permanent Generation" way too early, while JVM is launched and whether you initialize your String variable or not, it doesn't matter from "Permanent Generation" perspective.

Talking about other user-defined classes, JVM would load the class and allocate memory in "Permanent Generation" as soon as you define the class, again even if you do not create an object of the class, memory is allocated in "Permanent Generation" (**non-heap area**) and when you create an object of the class then memory is allocated in "Eden Space" (**heap area**).

Sources of above information and further reading:

- http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html
- http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java
- https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation
- Java heap terminology: young, old and permanent generations?

edited May 23 '17 at 12:18 answered Dec 29 '16 at 15:44


Community ♦
1 1


hagrawal
7,208 1 12 37

@Adit: Bounty is generally placed when the answers already provided are not "detailed" and sufficient enough, so I am wondering why did you placed a bounty on this question of your's because answer by "arcy" was already given on "Dec 13" and you placed bounty recently? – hagrawal Jan 2 '17 at 14:23

Since the arsy's accepted answer and hagrawal's answer's are clear, just want to elaborate on the fourth question :

> What would have been the case, if I had only declared s and did not initialize it, would JVM still allocate memory to all the methods of java.lang.String class,if so,why?

Basically while its true that the class data - that has the fields and methods information - is stored in the permanent-generation (meta-space from JDK-8 onwards), it is important to note that its the objects within the java.lang.String class (such as the char[] which holds all the character information for that string) for which data is allocated on the heap.

This does not happen until either a new string object is created - either using the 'new' keyword or by creating a new string literal (Eg: "helloworld").

answered Dec 29 '16 at 19:43


Ravindra HV
1,304 8 19