Stack Memory and Heap Space in Java

baeldung.com/java-stack-heap

By
baeldung

11 July
2018

1. Introduction

To run an application in an optimal way, JVM divides memory into stack and heap memory. **Whenever we declare new variables and objects, call new method, declare a *String* or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space.**

In this tutorial, we'll discuss these memory models. We'll enlist some key differences between them, how they are stored in RAM, the features they offer and where to use them.

2. Stack Memory in Java

Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects that are in a heap, referred from the method.

Access to this memory is in Last-In-First-Out (LIFO) order. Whenever a new method is called, a new block on top of the stack is created which contains values specific to that method, like primitive variables and references to objects.

When the method finishes execution, it's corresponding stack frame is flushed, the flow goes back to the calling method and space becomes available for the next method.

2.1. Key Features of Stack Memory

Apart from what we have discussed so far, following are some other features of stack memory:

- It grows and shrinks as new methods are called and returned respectively
- Variables inside stack exist only as long as the method that created them is running
- It's automatically allocated and deallocated when method finishes execution
- If this memory is full, Java throws *java.lang.StackOverFlowError*
- Access to this memory is fast when compared to heap memory
- This memory is threadsafe as each thread operates in its own stack

3. Heap Space in Java

Heap space in Java is used for dynamic memory allocation for Java objects and JRE classes at the runtime. New objects are always created in heap space and the references to this objects are stored in stack memory.

These objects have global access and can be accessed from anywhere in the application.

This memory model is further broken into smaller parts called generations, these are:

1. **Young Generation –** this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up
2. **Old or Tenured Generation –** this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set and when that threshold is reached, the object is moved to the old generation
3. **Permanent Generation –** this consists of JVM metadata for the runtime classes and application methods

These different portions are also discussed in this article – Difference Between JVM, JRE, and JDK.

We can always manipulate the size of heap memory as per our requirement. For more information, visit this linked Baeldung article.

3.1. Key Features of Java Heap Memory

Apart from what we have discussed so far, following are some other features of heap space:

- It's accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation
- If heap space is full, Java throws *java.lang.OutOfMemoryError*
- Access to this memory is relatively slower than stack memory
- This memory, in contrast to stack, isn't automatically deallocated. It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage
- Unlike stack, a heap isn't threadsafe and needs to be guarded by properly synchronizing the code

4. Example

Based on what we've learned so far, let's analyze a simple Java code and let's assess how memory is managed here:

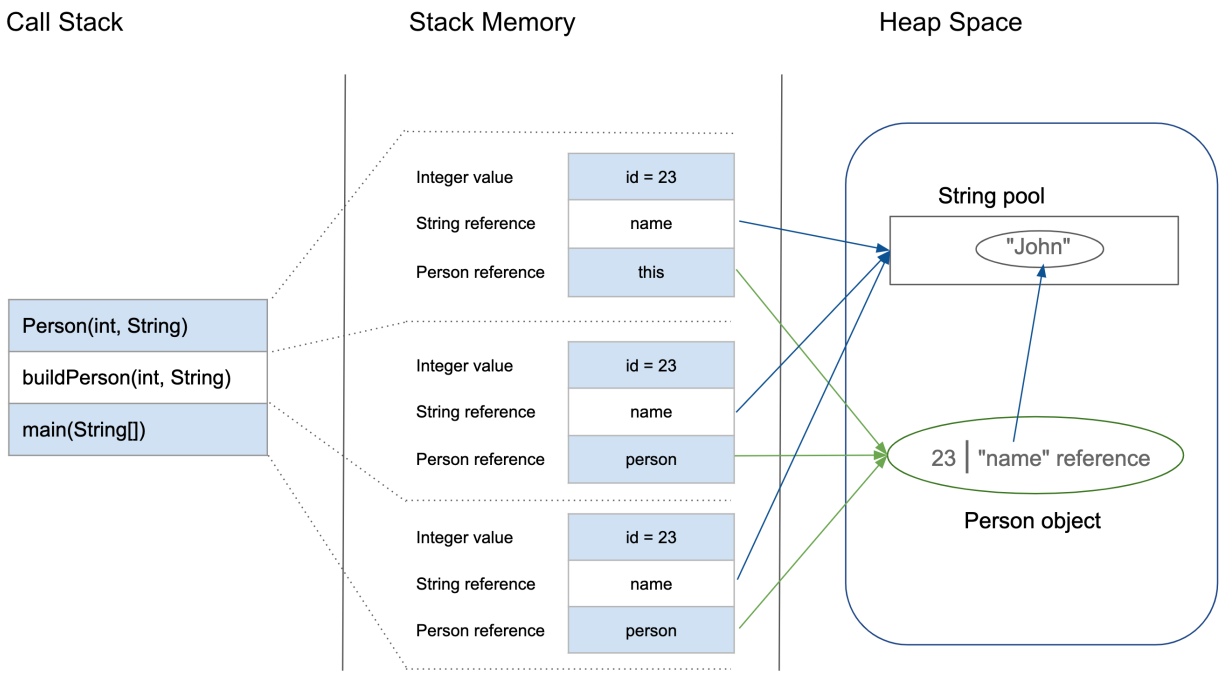
```
class Person {
int id;
String name;

public Person(int id, String name) {
this.id = id;
this.name = name;
}
}

public class PersonBuilder {
private static Person buildPerson(int id, String name)
{
return new Person(id, name);
}

public static void main(String[] args) {
int id = 23;
String name = "John";
Person person = null;
person = buildPerson(id, name);
}
}
```

Let's analyze this step-by-step:

1. Upon entering the *main()* method, a space in stack memory would be created to store primitives and references of this method
    - The primitive value of integer *id* will be stored directly in stack memory
    - The reference variable *person* of type *Person* will also be created in stack memory which will point to the actual object in the heap

2. The call to the parameterized constructor *Person(int, String)* from *main()* will allocate further memory on top of the previous stack. This will store:
    - The *this* object reference of the calling object in stack memory
    - The primitive value *id* in the stack memory
    - The reference variable of *String* argument *name* which will point to the actual string from string pool in heap memory
3. The *main* method is further calling the *buildPerson()* static method, for which further allocation will take place in stack memory on top of the previous one. This will again store variables in the manner described above.
4. However, for the newly created object *person* of type *Person*, all instance variables will be stored in heap memory.

This allocation is explained in this diagram:



5. Summary

Before we conclude this article, let's quickly summarize the differences between the Stack Memory and the Heap Space:

Parameter	Stack Memory	Heap Space
Application	Stack is used in parts, one at a time during execution of a thread	The entire application uses Heap space during runtime

Parameter	Stack Memory	Heap Space
Size	Stack has size limits depending upon OS and is usually smaller then Heap	There is no size limit on Heap
Storage	Stores only primitive variables and references to objects that are created in Heap Space	All the newly created objects are stored here
Order	It is accessed using Last-in First-out (LIFO) memory allocation system	This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation.
Life	Stack memory only exists as long as the current method is running	Heap space exists as long as the application runs
Efficiency	Comparatively much faster to allocate when compared to heap	Slower to allocate when compared to stack
Allocation/Deallocation	This Memory is automatically allocated and deallocated when a method is called and returned respectively	Heap space is allocated when new objects are created and deallocated by Gargabe Collector when they are no longer referenced

6. Conclusion

Stack and heap are two ways in which Java allocates memory. In this article, we understood how they work and when to use them for developing better Java programs.

To learn more about Memory Management in Java, have a look at this article here. We also discussed the JVM Garbage Collector which is discussed briefly over in this article.

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE

Learning to "Build your API with Spring"?