**Heap Memory and Stack Memory in Java**

Learn about stack and heap memory.

G

Ganesh Kumar Marimuthu

24 Dec 2021-10 mins read

**Challenge Inside! :**Find out where you stand! Try quiz, solve problems & win rewards!

[Go to Challenge](https://www.scaler.com/topics/java/heap-memory-and-stack-memory-in-java/" \l "topic-challenge-container-150796)

**Overview**

Stack memory is the space allocated for a process where all the function calls, primitive data types (int, double, etc.) and local and reference variables of the functions are stored. On the other hand heap memory is used to store the objects that are created during the execution of a Java program. Heap follows dynamic memory allocation (memory is allocated during execution or runtime) and provides random access, unlike stack which follows Last-In-First-Out (LIFO) order.

* In this article, we will learn about stack and heap memory in java.
* We will also learn about the *OutOfMemoryError* and *StackOverflowError* in java.
* We will see the various advantages and disadvantages of stack and heap memory. This will give us a clear idea about where we should use each one of them.

**Introduction**

Java applications are compiled and executed in the RAM of the computer. Each application is allocated a certain amount of memory. This allocated memory in RAM is called application memory. Data and code are stored here. The amount of memory allocated to a Java process depends on multiple factors like Java Runtime Environment (JRE), operating system, processor, etc. The JVM divides the memory allocated for a process into five parts: **Stack, Heap, Class/Method Area, Program Counter Register**, and **Native Method Stack**. In this article, we will be diving deep into stack and heap memory.

**What is Stack Memory in Java?**

Stack memory is the space allocated for a process where all the function calls, primitive data types like int, double, etc., and local and reference variables of the functions are stored. Stack memory is always accessed in a Last-In-First-Out (LIFO) manner. In the stack memory, a new memory block is created for every method that is executed. All the primitive variables and references to objects inside the method are stored in this memory block. When the method completes its execution, the memory block is cleared from the stack memory and the stack memory is available for use. The values in the stack exist for as long as the function that created them is running. The size of the stack memory is fixed and cannot grow or shrink once created.

**Example of Stack Memory in Java**

Below is a simple Java program with three methods main, addOne, and addTwo. We will see the step-by-step explanation of the stack usage when this program executes.

public class Main {

public static int addOne(int input) {

return input + 1;

}

public static int addTwo(int input) {

return input + 2;

}

public static void main(String[] args) {

int x = 0;

x = addOne(x);

x = addTwo(x);

}

}

1. When the program is executed, the main method is executed first by the JVM. When the main method is called, a block is allocated for it in the stack. The variables in the method are created, assigned values, and then stored in the block.
2. When the addOne method is called from the main method, a new block is allocated in the stack. The variables for the method are created and stored in the block. Upon the completion of the execution of the method, the value is returned to the calling method(here it is the main method), and the block is cleared.
3. Similarly, when the addTwo method is called, a new block is allocated for it, and the variables are created and stored. When the method finishes execution, the value is returned to the calling method, and the block is cleared.
4. Finally, the main method completes its execution, and the block is cleared from the stack.

**Why StackOverflowError is thrown in Java?**

As mentioned earlier, stack memory is fixed and cannot be enlarged or shrunk once created. Therefore, if we use all the stack memory, there will be no space left for upcoming method calls, and we will get the StackOverflowError.

public class Main {

public static int factorial(int n) {

*/\* Base case is commented to make it run indefinitely*

*if (n == 0) {*

*return 1;*

*}*

*\*/*

return n \* factorial(n - 1);

}

public static void main(String[] args) {

int x = 3;

int fact = factorial(x);

}

}

**Error**

Exception in thread "main" java.lang.StackOverflowError

at StackMemory.factorial(StackMemory.java:15)

at StackMemory.factorial(StackMemory.java:15)

...

When the above program is executed, the factorial method will be called **indefinitely** because the base case is commented. A new block on the stack is allocated for each call to the factorial method. The stack size is fixed, and the factorial method is called indefinitely and doesn't return any value, so the stack memory runs out, resulting in StackOverflowError.

**What is Heap Memory in Java?**

Heap memory is used to store the objects that are created during the execution of a Java program. The reference to the objects that are created is stored in stack memory. Heap follows dynamic memory allocation (memory is allocated during execution or runtime) and provides random access, unlike stack, which follows Last-In-First-Out (LIFO) order. The size of heap memory is large when compared to stack. The unused objects in the heap memory are cleared automatically by the Garbage Collector. The heap memory can be divided into three parts

1. New or Young Generation
2. Old or Tenured Generation
3. Permanent Generation

**New or Young Generation**

Young generation is the place where all the newly created objects are allocated. New generation inturn has 3 parts, **Eden**, **Survivor1** and **Survivor2**. All the newly created objects are allocated in **Eden** space. When the Eden is full, a **minor garbage collection** happens, and the live objects are moved to **Survivor1** and then to **Survivor2**. The Survivor1 and Survivor2 contains objects that survived the minor garbage collection. Objects that survive Eden, Survivor1 and Survivor2 are moved to tenured generation. In tenured generation the garbage is collected less frequently, so Survivor1 and Survivor2 spaces are used to make sure that only long survived objects are moved to tenured generation.

**Old or Tenured Generation**

Age is set for the objects allocated in young generation. When that age is met, those live objects are moved to the old generation. Generally, long-surviving objects are stored in the old generation. A **major garbage collection** runs on the old generation to collect dead objects.

**Permanent Generation**

Permanent generation is used by JVM to store metadata about the classes and methods. JVM also stores Java standard libraries in permanent generation. This space is cleaned as a part of **full garbage collection**.

**Example of Heap Memory in Java**

import java.util.ArrayList;

import java.util.List;

public class HeapMemory {

public static void main(String[] args) {

int x = 10;

List < Integer > list = new ArrayList < > ();

list.add(1);

list.add(2);

list.add(3);

}

}

In the above example, the variable *x* is allocated in the stack, whereas the object *list* is allocated in the heap. Only the reference to the *list* object is stored in a stack.

**Why OutOfMemoryError is thrown in Java?**

OutOfMemoryError is thrown when there is no more space left in the heap to create and store a new object. This happens when the Garbage Collector could not freeup any space to store new objects.

public class HeapMemory {

public static void main(String[] args) {

for (int i = 1; i < 100; i \*= 2) {

int n = (int) Math.pow(2, i);

int[] array = new int[n];

for (int j = 0; j < n; j++) {

array[j] = 1000;

}

}

}

}

**Error**

Exception in thread "main" java.lang.OutOfMemoryError: Requested array size exceeds VM limit

at HeapMemory.main(HeapMemory.java:7)

Consider the above program where we are repeatedly generating arrays of bigger sizes and storing values in them. Once the space ran out in the heap, it threw OutOfMemoryError.

**Differences Between Stack vs Heap**

| **Property** | **Stack Memory** | **Heap Memory** |
| --- | --- | --- |
| Size | The size of stack memory is **smaller** | The size of heap memory is **larger** |
| Order | Stack memory is accessed in **Last-In-First-Out (LIFO)** manner | Heap memory is **dynamically allocated** and does not follow any order |
| Speed | Access to stack memory is **faster** because of Last-In-First-Out (LIFO) ordering | Access to heap memory is **slower** because it does not follow any order and is allocated dynamically |
| Resizing | Resizing of variables is **not allowed** in stack | Resizing of variables is **allowed** in a heap |
| Allocation | Memory is allocated and deallocated automatically when a method starts and completes its execution respectively | Memory is allocated when objects are created and deallocated by the garbage collector when they are no longer in use |
| Storage | Local variables and object references inside the function are stored in stack | The newly created objects and the JRE classes are stored in a heap |
| Exception | **StackOverflowError** is thrown when there is no more space left in the stack for new method calls | **OutOfMemoryError** is thrown when there is no space left in a heap to allocate new objects |
| Thread Safety | Each thread is allocated with a new stack, and it is **thread-safe** | Heap memory is shared across all threads, and it is **not thread-safe** |

**When to Use Stack and Heap Memory in Java**

We can decide when to use stack and heap memory in java based on the above-listed properties. Stack memory can be used

* When the variables are not used outside the method scope.
* When the size required for the variables is small.
* When the variables created should be thread-safe.
* When accessing variables should be faster.

Heap memory can be used.

* When the variables are used outside the method scope.
* When the size required for the variables is large.
* When variables need to be shared across threads.

**Advantages of using Stack Memory in Java**

* Stack memory is thread-safe because each thread has its stack area.
* Memory allocation and deallocation are faster.
* Memory is automatically allocated and deallocated for a method.
* Access to stack memory is faster.

**Disadvantages of using Stack Memory in Java**

* Stack memory is fixed and cannot grow or shrink once created.
* Stack memory works in Last-In-First-Out (LIFO) manner. So, random access is not possible.
* Stack memory is neither scalable nor flexible.

**Advantages of using Heap Memory in Java**

* Heap memory is not fixed, and it can grow and shrink.
* Heap memory doesn't flow any ordering and allows random access.
* Heap memory is scalable and flexible.

**Disadvantages of using Heap Memory in Java**

* Heap memory is shared across threads and is not thread-safe.
* Access to heap memory is slower.
* Memory allocation and deallocation is complex when compared to stack.

**Conclusion**

* Stack and Heap memory are allocated to a program by the Java Virtual Machine (JVM).
* All the primitive data types and references to objects created inside a method are stored in the stack.
* The values in the stack exist for as long as the function that created them is running
* Stack memory is accessed in a Last-In-First-Out (LIFO) manner.
* The size of the stack is small and fixed.
* Stack can be used when the scope of the variable is not used outside the method scope, the access to the memory should be faster, and the variables should be thread-safe.
* All the objects created are stored in a heap.
* The size of the heap is large when compared to the stack.
* Heap memory follows dynamic allocation and has three parts, New Generation, Young Generation, and Permanent Generation.
* Heap memory can be used when the scope is global and accessed across threads.