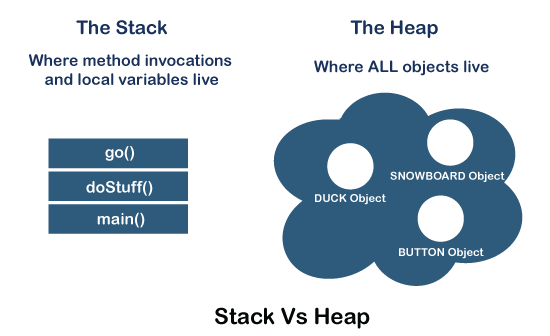
Stack Vs Heap Java

In Java, [memory management](https://www.javatpoint.com/memory-management-in-java) is a vital process. It is managed by Java automatically. The JVM divides the memory into two parts: [stack](https://www.javatpoint.com/java-stack) memory and [heap](https://www.javatpoint.com/java-heap) memory. From the perspective of Java, both are important memory areas but both are used for different purposes. The **major difference between Stack memory and heap memory** is that the stack is used to store the order of method execution and local variables while the heap memory stores the objects and it uses dynamic memory allocation and deallocation. In this section, we will discuss the differences between stack and heap in detail.



Stack Memory

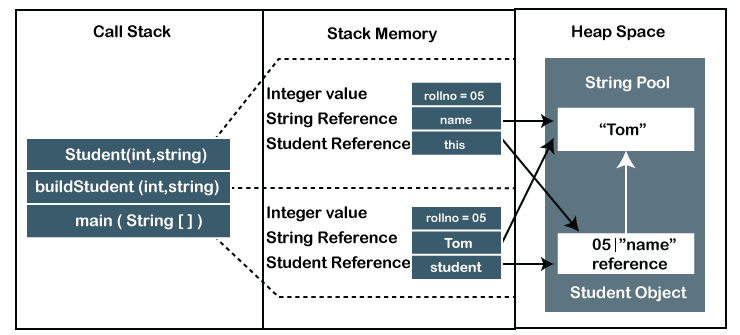
The stack memory is a physical space (in RAM) allocated to each thread at run time. It is created when a thread creates. Memory management in the stack follows LIFO (Last-In-First-Out) order because it is accessible globally. It stores the variables, references to objects, and partial results. Memory allocated to stack lives until the function returns. If there is no space for creating the new objects, it throws the java.lang.StackOverFlowError. The scope of the elements is limited to their threads. The [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates a separate stack for each thread.

Heap Memory

It is created when the JVM starts up and used by the application as long as the application runs. It stores objects and JRE classes. Whenever we create objects it occupies space in the heap memory while the reference of that object creates in the stack. It does not follow any order like the stack. It dynamically handles the memory blocks. It means, we need not to handle the memory manually. For managing the memory automatically, [Java](https://www.javatpoint.com/java-tutorial) provides the [garbage collector](https://www.javatpoint.com/Garbage-Collection) that deletes the objects which are no longer being used. Memory allocated to heap lives until any one event, either program terminated or memory free does not occur. The elements are globally accessible in the application. It is a common memory space shared with all the threads. If the heap space is full, it throws the java.lang.OutOfMemoryError. The heap memory is further divided into the following memory areas:

* Young generation
* Survivor space
* Old generation
* Permanent generation
* Code Cache

The following image shows the allocation of stack memory and heap space.



Difference Between Stack and Heap Memory

The following table summarizes all the major differences between stack memory and heap space.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Stack Memory** | **Heap Space** |
| **Application** | It stores items that have a very short life such as **methods, variables,** and **reference variables** of the objects. | It stores **objects** and Java Runtime Environment (**JRE**) classes. |
| **Ordering** | It follows the **LIFO** order. | It does not follow any order because it is a dynamic memory allocation and does not have any fixed pattern for allocation and deallocation of memory blocks. |
| **Flexibility** | It is **not flexible** because we cannot alter the allocated memory. | It is **flexible** because we can alter the allocated memory. |
| **Efficiency** | It has **faster** access, allocation, and deallocation. | It has **slower** access, allocation, and deallocation. |
| **Memory Size** | It is **smaller** in size. | It is **larger** in size. |
| **Java Options Used** | We can increase the stack size by using the JVM option -Xss. | We can increase or decrease the heap memory size by using the -[Xmx](https://www.javatpoint.com/java-xmx) and -Xms JVM options. |
| **Visibility or Scope** | The variables are visible only to the owner thread. | It is visible to all threads. |
| **Generation of Space** | When a thread is created, the operating system automatically allocates the stack. | To create the heap space for the application, the language first calls the operating system at run time. |
| **Distribution** | Separate stack is created for each object. | It is shared among all the threads. |
| **Exception Throws** | JVM throws the **java.lang.StackOverFlowError** if the stack size is greater than the limit. To avoid this error, increase the stack size. | JVM throws the **java.lang.OutOfMemoryError** if the JVM is unable to create a new native method. |
| **Allocation/ Deallocation** | It is done automatically by the **compiler**. | It is done manually by the **programmer**. |
| **Cost** | Its cost is **less**. | Its cost is **more** in comparison to stack. |
| **Implementation** | Its implementation is **hard**. | Its implementation is **easy**. |
| **Order of allocation** | Memory allocation is **continuous**. | Memory allocated in **random** order. |
| **Thread-Safety** | It is thread-safe because each thread has its own stack. | It is not thread-safe, so properly synchronization of code is required. |