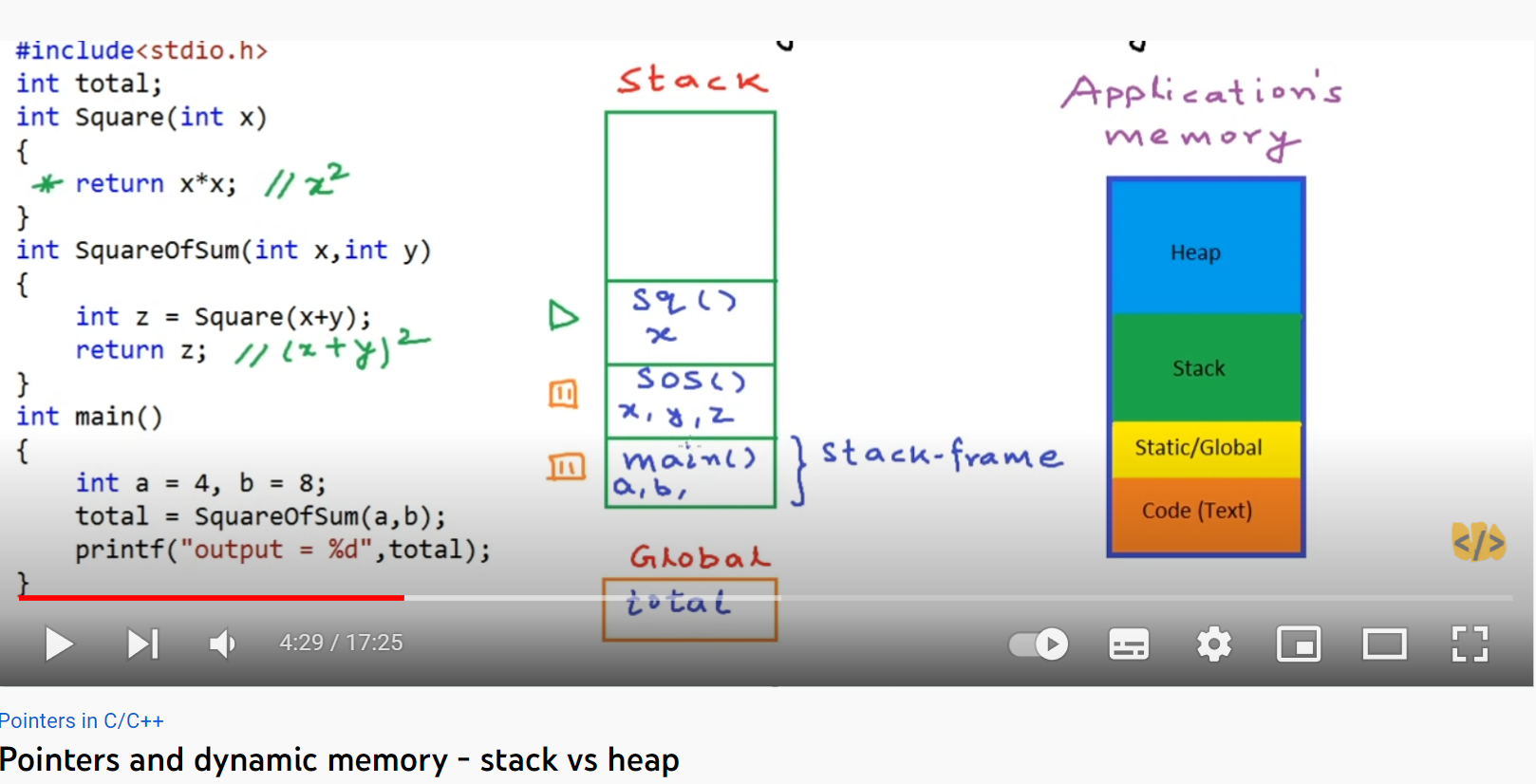
**Stack and Heap**

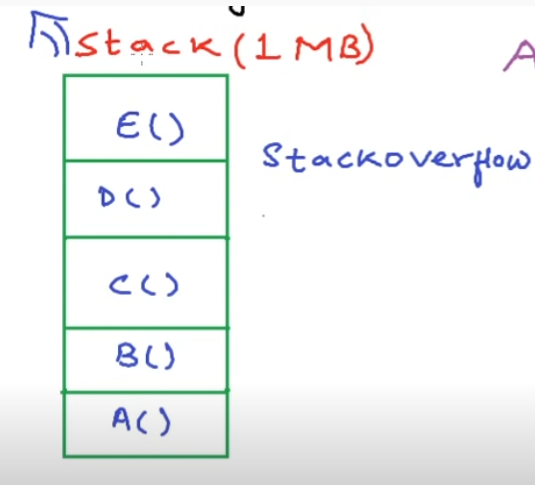


La partie « code » stocke les instructions du code à exécuter

La partie « static/global » contient les variables globales du programme, elles sont accessibles par toutes les fonctions, il est inutile de déclarer une variable locale comme étant global, ceci fera gaspiller l’espace mémoire

Le stack continent toutes les fonctions/ méthodes du processus organisées en pile (de telle façon à ce que la fonction actuelle est en haut de la pile). Lorsque l’exécution de la fonction s’achève, elle est supprimée de la pile la fonction qui l’a appelé prend le relais et continue son exécution.

Si la pile est remplie, l’erreur stackoverflow se produit, un exemple est le cas d’une récurrence infinie sur une fonction.



Le heap est un espace mémoire propre aux allocations dynamiques avec (new, malloc…), les objets ne sont pas stocker en pile, mais aléatoirement dans la mémoire, et accessible par des pointeurs stocké dans le STACK, une fois un espace mémoire dans le heap n’est plus pointé par un aucun pointeur, il est supprimé par le Garbage collector

Where are stack and heap physically?

**Stack Allocation:** The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated. This all happens using some predefined routines in the compiler. A programmer does not have to worry about memory allocation and de-allocation of stack variables. This kind of memory allocation also known as Temporary memory allocation because as soon as the method finishes its execution all the data belongs to that method flushes out from the stack automatically. Means, any value stored in the stack memory scheme is accessible as long as the method hasn’t completed its execution and currently in running state.

**Key Points:**

* It’s a temporary memory allocation scheme where the data members are accessible only if the method( ) that contained them is currently is running.
* It allocates or de-allocates the memory automatically as soon as the corresponding method completes its execution.
* We receive the corresponding error Java. lang. StackOverFlowError by JVM, If the stack memory is filled completely.
* Stack memory allocation is considered safer as compared to heap memory allocation because the data stored can only be access by owner thread.
* Memory allocation and de-allocation is faster as compared to Heap-memory allocation.
* Stack-memory has less storage space as compared to Heap-memory.
* CPP

|  |
| --- |
| int main()  {     // All these variables get memory     // allocated on stack     int a;     int b[10];     int n = 20;     int c[n];  } |

**Heap Allocation:**The memory is allocated during the execution of instructions written by programmers. Note that the name heap has nothing to do with the heap data structure. It is called heap because it is a pile of memory space available to programmers to allocate and de-allocate. Every time when we made an object it always creates in Heap-space and the referencing information to these objects are always stored in Stack-memory. Heap memory allocation isn’t as safe as Stack memory allocation was because the data stored in this space is accessible or visible to all threads. If a programmer does not handle this memory well, a [memory leak](https://www.geeksforgeeks.org/what-is-memory-leak-how-can-we-avoid/) can happen in the program.

**The Heap-memory allocation is further divided into three categories:-** These three categories help us to prioritize the data(Objects) to be stored in the Heap-memory or in the Garbage collection.

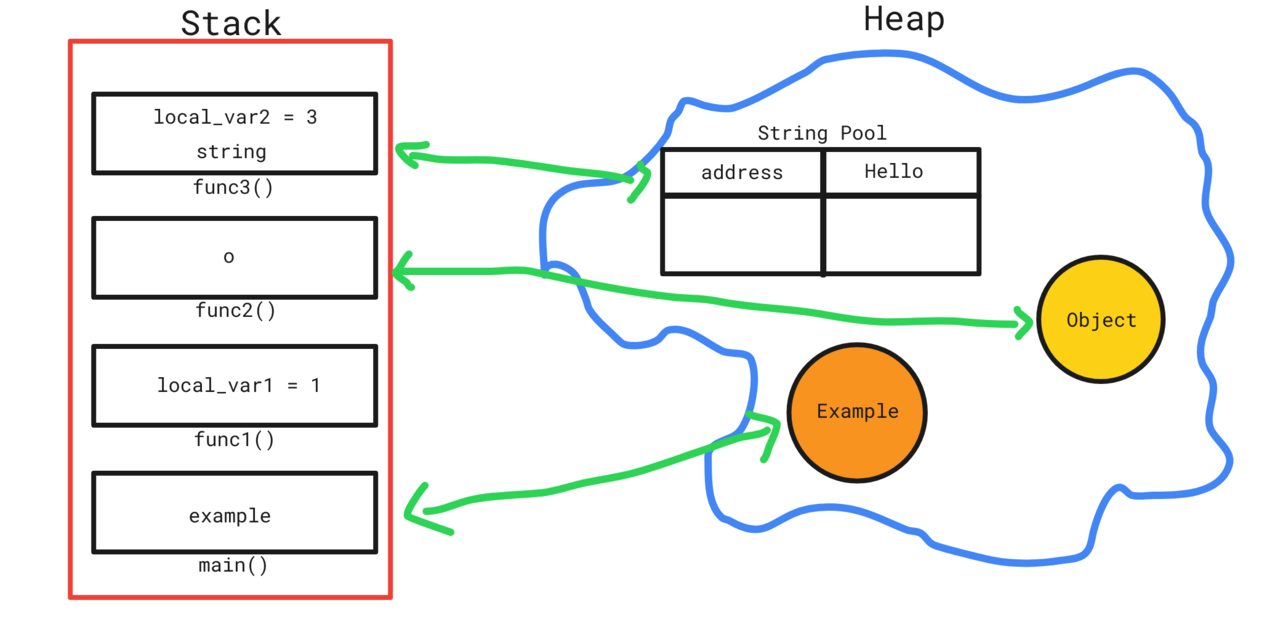
* **Young Generation –** It’s the portion of the memory where all the new data(objects) are made to allocate the space and whenever this memory is completely filled then the rest of the data is stored in Garbage collection.
* **Old or Tenured Generation –** This is the part of Heap-memory that contains the older data objects that are not in frequent use or not in use at all are placed.
* **Permanent Generation –**This is the portion of Heap-memory that contains the JVM’s metadata for the runtime classes and application methods.

**Key Points:**

* We receive the corresponding error message if Heap-space is entirely full,  java. lang.OutOfMemoryError by JVM.
* This memory allocation scheme is different from the Stack-space allocation, here no automatic de-allocation feature is provided. We need to use a Garbage collector to remove the old unused objects in order to use the memory efficiently.
* The processing time(Accessing time) of this memory is quite slow as compared to Stack-memory.
* Heap-memory is also not threaded-safe as Stack-memory because data stored in Heap-memory are visible to all threads.
* Size of Heap-memory is quite larger as compared to the Stack-memory.
* Heap-memory is accessible or exists as long as the whole application(or java program) runs.

**What and Where are Stack and Heap?**

* Publié le 22 juillet 2018



[[](https://www.linkedin.com/in/maxim-malisciuc/)](https://www.linkedin.com/in/maxim-malisciuc/)

**[Maxim M.](https://www.linkedin.com/in/maxim-malisciuc/)**

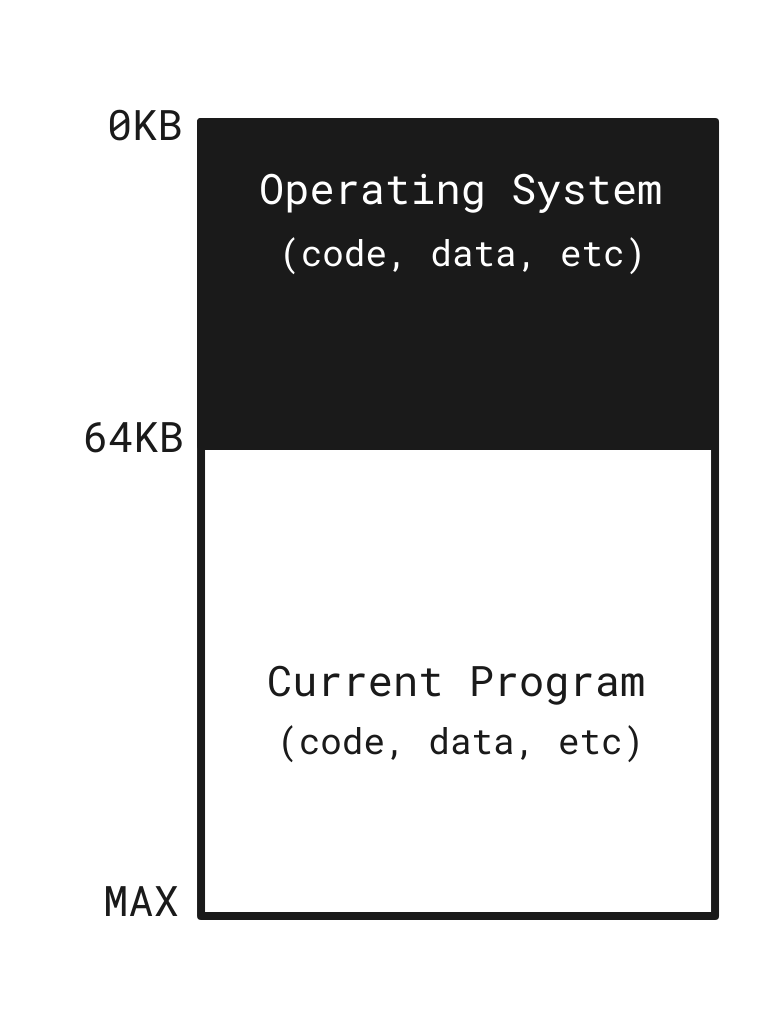
Senior Android Developer at AUTHADA GmbH

[4 articles](https://www.linkedin.com/in/maxim-malisciuc/recent-activity/posts/)Suivre

**Physical Level**

   First of all, you should understand that *Stack* and *Heap* does not have a physical representation - it is a memory abstraction, so there is not physical difference between them, they are both in RAM. I will not explain how RAM works in this article. Those abstraction are created by OS and are helping us to develop some good software and not to think about where each variable and functions is going in RAM and keep tracking it there.

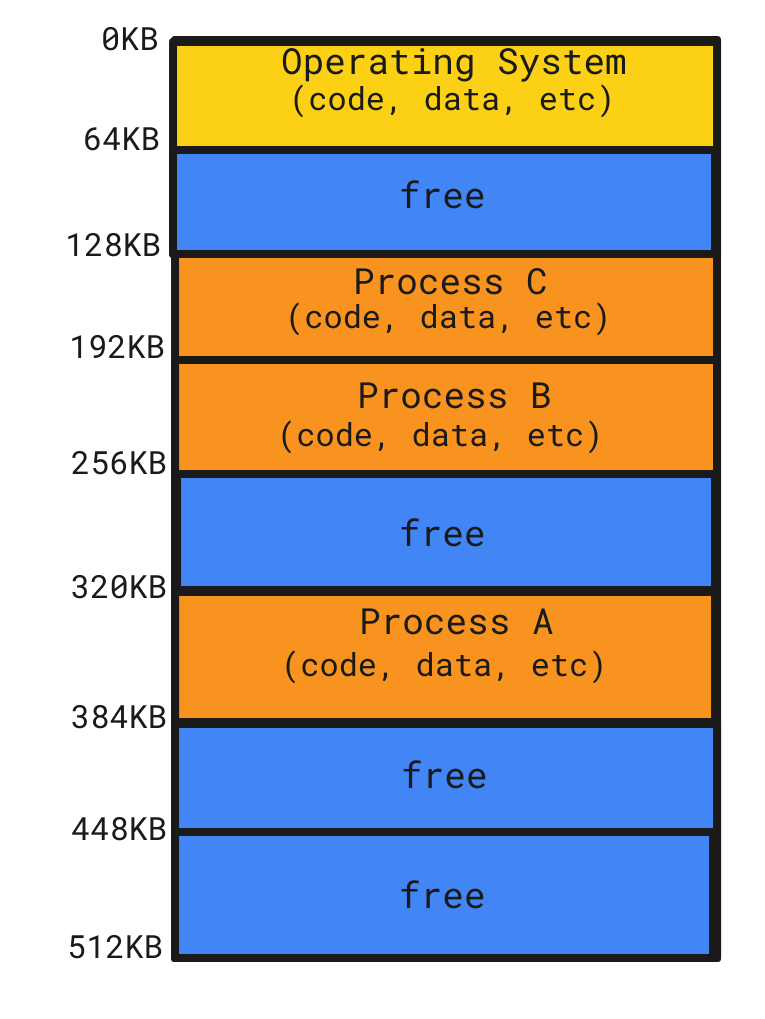
Some history here: First computers were able to run just one program(a process), that used all the memory it needed(starting from physical address 64k - max for this example).



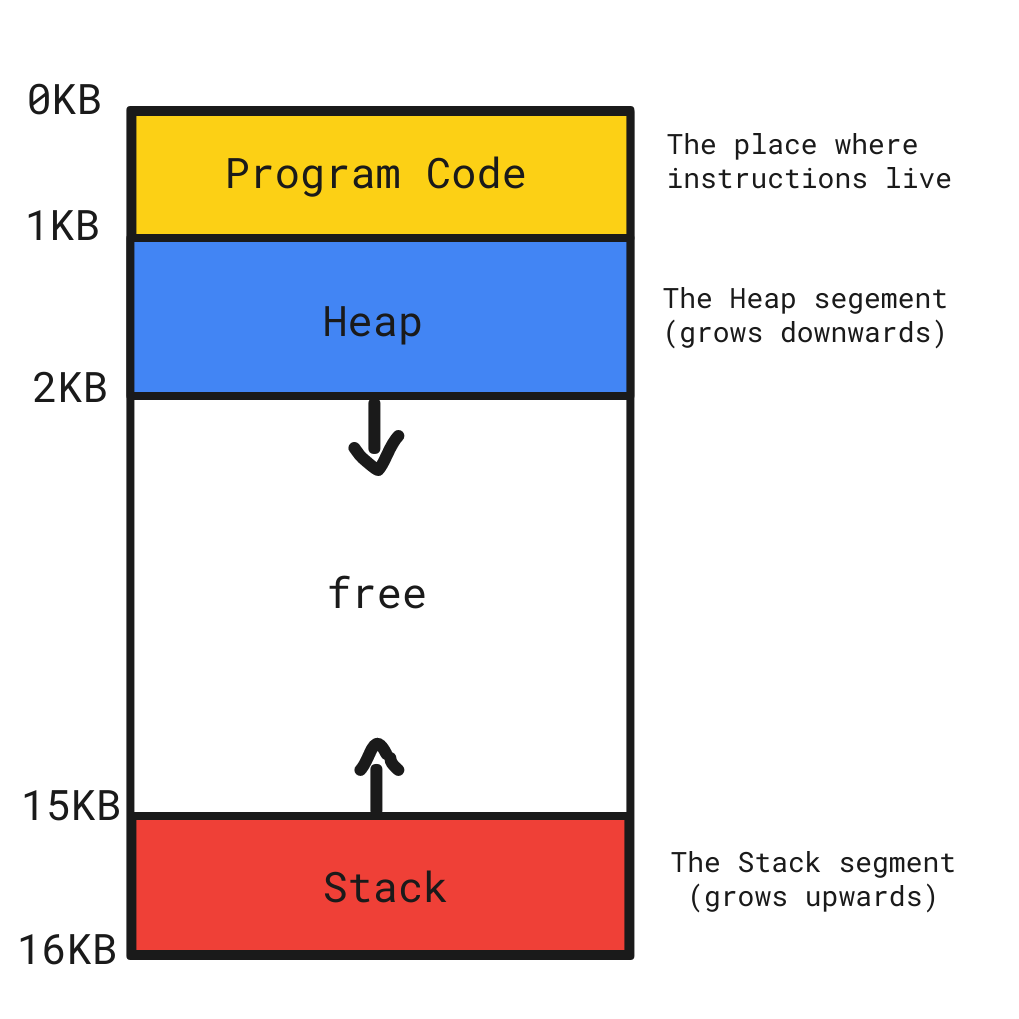
**multiprogramming** was born, ih which multiple processes were ready to run at a given time and OS would switch between them in order to *maximize CPU use*. Soon people began demanding more of machine and the era of **time sharing**(*minimizing response time*) was born in order to serve more users who used different terminals to access same machine.

Computers were expensive and it was useless to buy them for performing just one operation. Thus

  One way to implement time sharing would be to run one process for a short while, giving it full access to all memory, then stop it, save all of its state to some kind of disk (including all of physical memory), load some other process’s state, run it for a while, and thus implement some kind of crude sharing of the machine. Unfortunately, this approach has a big problem: it is way too slow, particularly as memory grows. While saving and restoring register-level state is relatively fast, saving the entire contents of memory to disk is brutally non-performant. Thus, what we’d rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently.



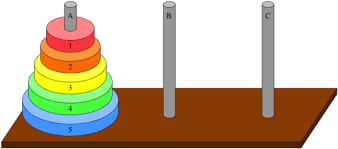
But this method has its own problems like **protection**, each process can access same memory. So process A could change some variables process B depends on. So the **address space** was created as new abstraction level. Each process has its own address space which is divided in different segments and looks basically like this:



For example program's code(instructions) have to live somewhere in memory and thus they are in address space so are **stack** and **heap**which we will discuss.

**Stack**

   Stack is a collection of items where addition and the removal of items always takes place at the "**top**"(stack also has "**base**" - opposite of "top"). You can imagine stack as tower from "[**Tower of Hanoi**](https://en.wikipedia.org/wiki/Tower_of_Hanoi)" problem.



  When you add an item it always goes on top of the tower and if you want to remove item "*2*" for example you should also remove item "*1*" in order to clear the *top*. This principle is called "**LIFO**"(*Last In First Out*), so you will always have older items near the *base*, while newer near the *top*.

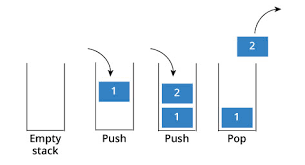
Example: when you use mobile app or browser and press back button - it will take you on the previous page, because as you navigate from page to page, those pages are placed on a stack with current page always on the top and the first page you looked on the base. When you click on the back button you will remove your current page(on the top) from the stack and the previous would become current.

 Each program gets a stack which is always allocated when it "starts" and can grow and shrink within constraints limited by OS. Stack is thread safe, because each thread has its own stack(but shares same heap)

 Every time you call a function: the stack pointer increments to the next physical memory address, creates new **block/**reserves space for its variables etc and copies it to that address. New block will have all local variables, function parameters, object references and return values.

 When you return from the function the block is removed from the top of the stack, memory released, the objects, variables etc used by this function are not accessible anymore and stack pointer decrements to the previous available item. The name of those procedures are "**Push**"(storing something on stack) and "**Pop**"(retrieving+deleting something from stack).

Stack is stores only primitive data types and addresses pointing to objects stored on Heap(object references). And all variables created on Stack are local and exists only while function executes, this concepts is called "[**variable scope**](https://www.tutorialspoint.com/cplusplus/cpp_variable_scope.htm)"(local and global variables).



You will get a **StackOverflowException** thrown when you program had used up all the available space on the stack and almost certainly the symptom of the endless recursive calls, when function starts calling itself and allocate memory on stack for its local variables each time till you are out of memory on Stack.

**Heap**

Heap memory is a Dynamic memory(its size changes as program run) used to store arrays, global variables(with global scope/accessible from any function) and any created class instances(objects) at runtime in Java which are referred by the reference variables from Stack memory. It also becomes a space for String pool which is used to store the String values in Java (every String values has a unique address inside the pool).

  Talking about the size of memory, Heap memory has bigger size than Stack memory and unlike the Stack, the Heap does not have size restrictions on variable size apart from the physical limitations of your computer) also unlike Stack(allocated at compile time), Heap is allocated at runtime. When the heap memory is out of space, you will get OutOfMemoryError: Java Heap Space exception thrown. When object reference is not longer available on Stack - Java Garbage Collector will deallocate space reserved by that object on Heap. Heap memory is slower than Stack, because it has to use [**pointer**](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) to access the allocated regions on the Heap.

**Summary**

**Heap                            Stack**



**Stack**

1. Stack is limited in size by Operating System.
2. Stack is fast.
3. Stack is used only for storing small data types(variables, object references).
4. Variables and References on Stack have Local Scope and are accessible only within their Stack memory block.

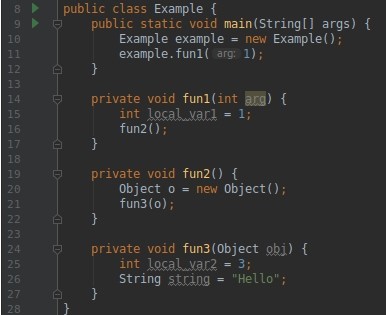
**Heap**

1. Heap has no size limits, apart from physical RAM limitations.
2. Heap is not as fast as Stack.
3. Heap stores all instances of the class(objects).
4. Variables and References on Heap have Global Scope and are accessible throughout the program.

**How does it work:**

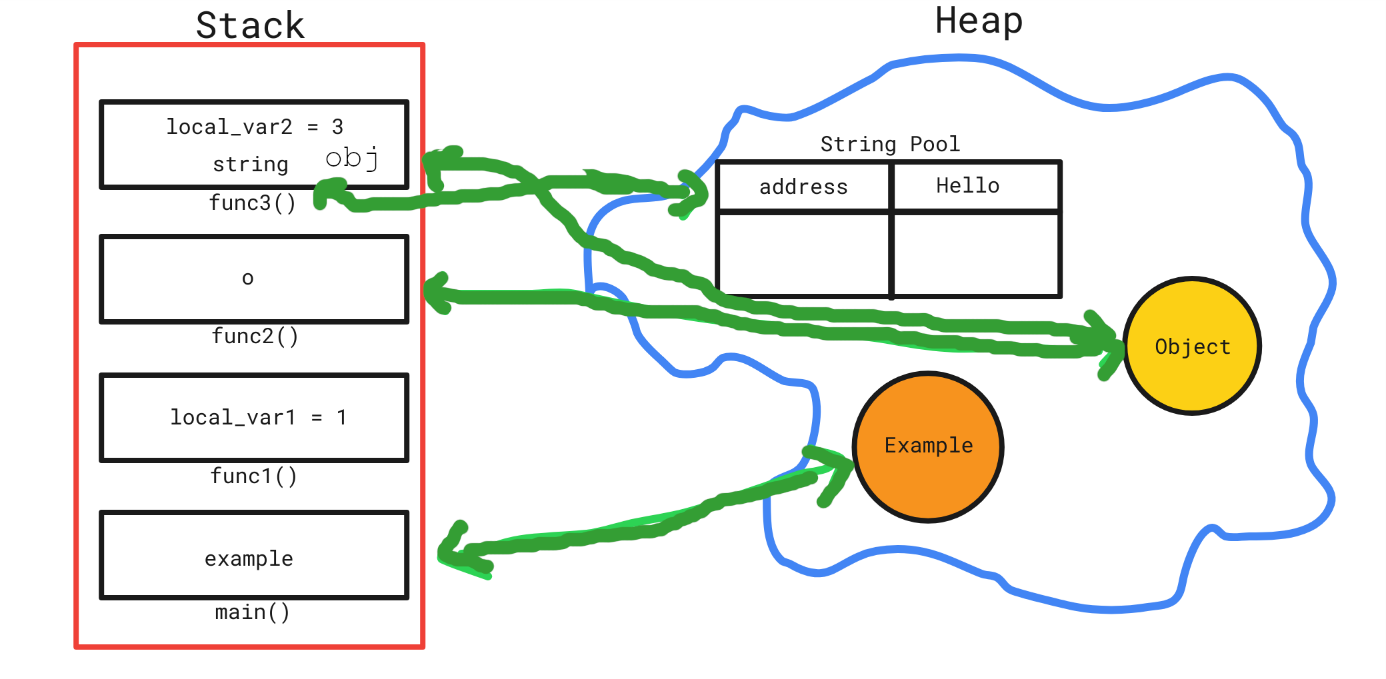
      Every time you create an instance of a class, some memory gets allocated on Heap to store that object and return a reference pointer to the start of that block of memory. This reference pointer comes in the form of a unique number represented in hexadecimal format, and as an integer it is stored on the Stack, so when we need to access that object on Heap, we find its reference on Stack which points to objects location on Heap, which is then accessed by that reference.

**Example:**



What is happening behind the wall(the scheme is showed below):

1. When JVM find **main()** method, the Stack frame will be created. After we create an instance of class Example, which means, that memory will be allocated on Heap to store the object and its address will be stored on Stack in form of a pointer.
2. When method **fun1()** is called, one more Stack frame will be created. **local\_var1**and its value will be stored on it, as it is local primitive variable.
3. Method **fun2()** is called. New Stack frame created and as with main() method memory allocation for object **o** will happen on Heap and pointer will be returned and saved on Stack.
4. Method **fun3()** is called. Parameter **obj** is saved on Stack as it is pointer to an object on Heap, **local\_var2** and its value is saved on Stack as well. Memory allocation for **string** will happen on the Heap in[**String Pool**](https://www.journaldev.com/797/what-is-java-string-pool) in Java.



   5. After all GC will be invoked and memory will be released.