The most important point is that heap and stack are generic terms for ways in which memory can be allocated. They can be implemented in many different ways, and the terms apply to the basic concepts.

* In a stack of items, items sit one on top of the other in the order they were placed there, and you can only remove the top one (without toppling the whole thing over).



* In a heap, there is no particular order to the way items are placed. You can reach in and remove items in any order because there is no clear 'top' item.



It does a fairly good job of describing the two ways of allocating and freeing memory in a stack and a heap. Yum!

* To what extent are they controlled by the OS or language runtime?

As mentioned, heap and stack are general terms, and can be implemented in many ways. Computer programs typically have a stack called a [call stack](http://en.wikipedia.org/wiki/Call_stack) which stores information relevant to the current function such as a pointer to whichever function it was called from, and any local variables. Because functions call other functions and then return, the stack grows and shrinks to hold information from the functions further down the call stack. A program doesn't really have runtime control over it; it's determined by the programming language, OS and even the system architecture.

A heap is a general term used for any memory that is allocated dynamically and randomly; i.e. out of order. The memory is typically allocated by the OS, with the application calling API functions to do this allocation. There is a fair bit of overhead required in managing dynamically allocated memory, which is usually handled by the OS.

* What is their scope?

The call stack is such a low level concept that it doesn't relate to 'scope' in the sense of programming. If you disassemble some code you'll see relative pointer style references to portions of the stack, but as far as a higher level language is concerned, the language imposes its own rules of scope. One important aspect of a stack, however, is that once a function returns, anything local to that function is immediately freed from the stack. That works the way you'd expect it to work given how your programming languages work. In a heap, it's also difficult to define. The scope is whatever is exposed by the OS, but your programming language probably adds its rules about what a "scope" is in your application. The processor architecture and the OS use virtual addressing, which the processor translates to physical addresses and there are page faults, etc. They keep track of what pages belong to which applications. You never really need to worry about this, though, because you just use whatever method your programming language uses to allocate and free memory, and check for errors (if the allocation/freeing fails for any reason).

* What determines the size of each of them?

Again, it depends on the language, compiler, operating system and architecture. A stack is usually pre-allocated, because by definition it must be contiguous memory (more on that in the last paragraph). The language compiler or the OS determine its size. You don't store huge chunks of data on the stack, so it'll be big enough that it should never be fully used, except in cases of unwanted endless recursion (hence, "stack overflow") or other unusual programming decisions.

A heap is a general term for anything that can be dynamically allocated. Depending on which way you look at it, it is constantly changing size. In modern processors and operating systems the exact way it works is very abstracted anyway, so you don't normally need to worry much about how it works deep down, except that (in languages where it lets you) you mustn't use memory that you haven't allocated yet or memory that you have freed.

* What makes one faster?

The stack is faster because all free memory is always contiguous. No list needs to be maintained of all the segments of free memory, just a single pointer to the current top of the stack. Compilers usually store this pointer in a special, fast [register](http://en.wikipedia.org/wiki/Stack_register) for this purpose. What's more, subsequent operations on a stack are usually concentrated within very nearby areas of memory, which at a very low level is good for optimization by the processor on-die caches.