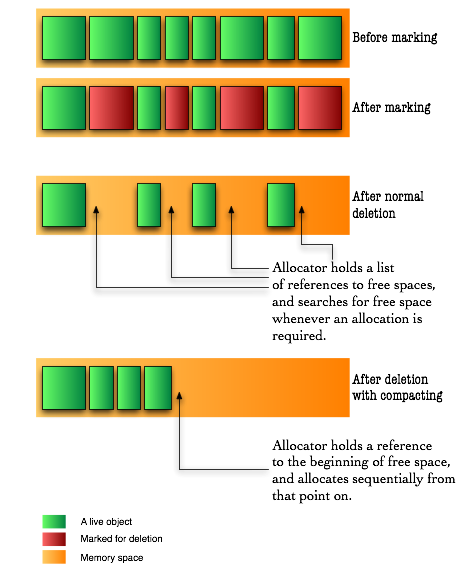
**The basics of garbage collection**

The garbage collector first performs a task called *marking*. The garbage collector traverses the application graph, starting with the *root objects*; those are objects that are represented by all active stack frames and all the static variables loaded into the system. Each object the garbage collector meets is marked as being used, and will not be deleted in the *sweeping* stage.

The *sweeping* stage is where the deletion of objects take place. There are many ways to delete an object: The traditional C way was to mark the space as free, and let the allocator methods use complex data structures to search the memory for the required free space. This was later improved by providing a defragmenting system which compacted memory by moving objects closer to each other, removing any fragments of free space and therefore allowing allocation to be much faster:



For the last trick to be possible a new idea was introduced in garbage collected languages: even though objects are represented by references, much like in C, they donâ€™t really reference their real memory location. Instead, they refer to a location in a dictionary which keeps track of where the object is at any moment.

Fortunately for us – but unfortunately for these garbage collection algorithms – our servers and personal computers got faster (and multiple) processors and bigger memory capacities. Compacting memory areas this large often was very taxing on the application, especially considering that when doing that, the whole application had to freeze due to the changes in the virtual memory map. Fortunately for us though, some smart people improved those algorithms in three ways: concurrency, parallelization and generational collection.

**Generational garbage collection**

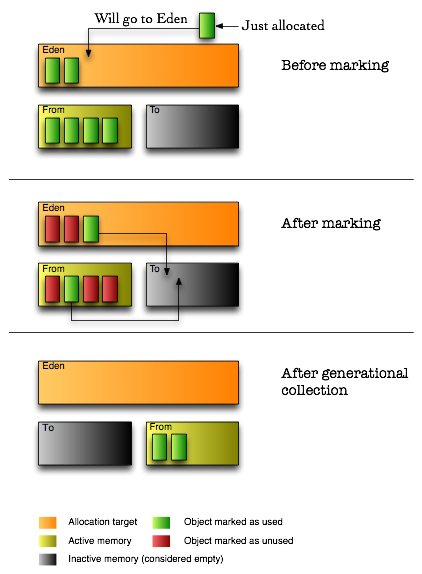
In any application, objects could be categorized according to their life-line. Some objects are short-lived, such as most local variables, and some are long-lived such as the backbone of the application. The thought about generational garbage collection was made possible with the understanding that in an applicationâ€™s lifetime, **most instantiated objects are short-lived**, and that there are **few connections between long-lived objects to short-lived objects**.

In order to take advantage of this information, the memory space is divided to two sections: *young generation* and *old generation*. In Java, the long-lived objects are further divided again to *permanent objects* and *old generation objects*. Permanent objects are usually objects the Java VM itself created for caching like code, reflection information etc. Old generation objects are objects that survived a few collections in the young generation area.

Since we know that objects in the young generation memory space become garbage early, we collect that area frequently while leaving the old generationâ€™s memory space to be collected in larger intervals. The young generation memory space is much smaller, thus having shorter collection times.

An additional advantage to the knowledge that objects die quickly in this area, we can also skip the compacting step and do something else called *copying*. This means that instead of seeking free areas (by seeking the areas marked as unused after the marking step), we copy the live objects from one young generation area to another young generation area. The originating area is called the From area, and the target area is called the To area, and after the copying is completed the roles switch: the From becomes the To, and the To becomes the From.

In addition, the Java VM splits the young generation to *three* areas, by adding an area called Eden which is where all objects are allocated into. To my understanding this is done to make allocation faster by always having the allocator reference to the beginning of Eden after a collection.



By using the copying method, garbage collection achieves defragmentation without seeking for dead memory blocks. However, this method proves itself to be more efficient in areas where most objects are garbage, so it is not a good approach to take on the old generation memory area. Indeed, that area is still collected using the compacting algorithm – but now, thanks to the separation of young and old generations, it is done in much larger intervals.

**Next up**

I didnâ€™t expect the amount of information Iâ€™ve found. I especially didnâ€™t expect the amount of information Iâ€™ve found regarding how the garbage collector makes use of the multiple processors platforms available today in almost all new computers. Iâ€™m not a big believer in extremely long, 3,000 words posts, so all the information regarding parallel and concurrent garbage collectors can be found on the next post, allowing me to upload this one now and have a couple of days to edit the next one before sending it online.