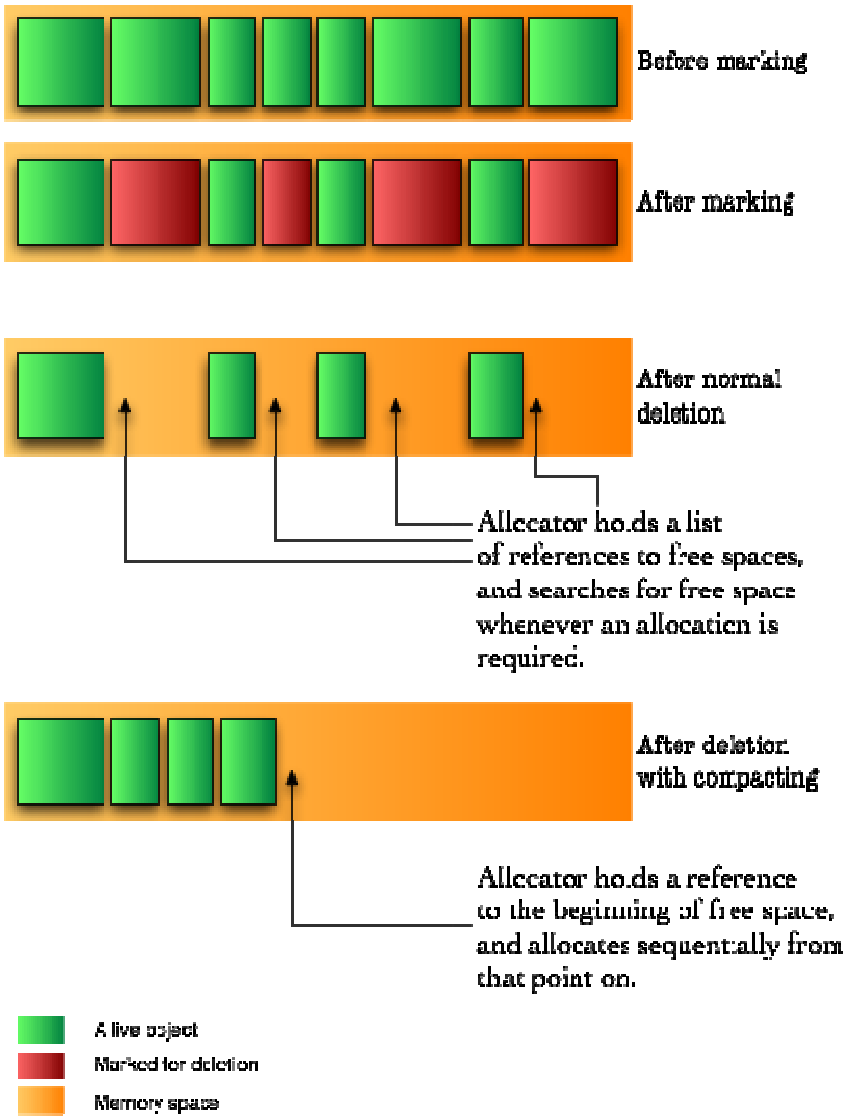
GARBAGE COLLECTION

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 takes 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 don'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.

Java's Garbage Collection

The JVM's heap stores all objects created by an executing Java program. Objects are created by Java's *"new"* operator, and memory for new objects is allocated on the heap at run time. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to keep track of when to free allocated memory, thereby preventing many potential bugs and headaches.

The name *"garbage collection"* implies that objects that are no longer needed by the program are *"garbage"* and can be thrown away. A more accurate and up-to-date metaphor might be *"memory recycling".* When an object is no longer referenced by the program, the heap space it occupies must be recycled so that the space is available for subsequent new objects. The garbage collector must somehow determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects. In the process of freeing unreferenced objects, the garbage collector must run any *"finalizers"* of objects being freed.

Why garbage collection?

Garbage collection relieves programmers from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the JVM has several advantages. First, it can make programmers more productive. When programming in non-garbage-collected languages the programmer can spend many late hours (or days or weeks) chasing down an elusive memory problem. When programming in Java the programmer can use that time more advantageously by getting ahead of schedule or simply going home to have a life.

Invoking the garbage collector

Invoking the garbage collector requires a simple two-step process. First you create a Java *Runtime* object. If you haven't used them before, Runtime objects let you interface with the environment in which your application is running. Then, after creating the Runtime object, you'll invoke the *gc()* method ("garbage collector") of the Runtime class.

Written in Java, these two steps look like this:

```
Runtime r = Runtime.getRuntime();
r.gc();
```

Java's Garbage Collector: A simple test program

```
public class GarbageCollectorTest
{
      final int REQUIRED_MEMORY = 50000;

      void consumeMemory()
      {
            int[] intArray = new int[REQUIRED_MEMORY];

            for (int i=0; i<REQUIRED_MEMORY; i++)
            {
                  intArray[i] = i;
            }
      }

public static void main (String[] args)
{
      GarbageCollectorTest gct = new GarbageCollectorTest();

      //get a Runtime object
      Runtime r = Runtime.getRuntime();

      //determine the current amount of free memory
      long freeMem = r.freeMemory();
      System.out.println("free memory before creating array: " + freeMem);

      //consume some memory
      gct.consumeMemory();

      //determine amount of memory left after consumption
      freeMem = r.freeMemory();
      System.out.println("free memory after creating array:  " + freeMem);

      //run the garbage collector, then check freeMemory
      r.gc();
      freeMem = r.freeMemory();
      System.out.println("free memory after running gc():    " + freeMem);
   }
}
```

Run it and observe the behavior of this program.