12

GARBAGE COLLECTION IN MRI, JRUBY, AND RUBINIUS

Garbage collection (GC) is the process high-level languages like Ruby use to manage memory for you. Where do your Ruby objects live while you're using them? How does Ruby clean up objects your program no longer uses? Ruby's GC system solves these problems.

Garbage collection is not unique to Ruby. The first implementation of garbage collection was in the Lisp programming language, invented by John McCarthy around 1960. Like Ruby, Lisp manages memory for you automatically using garbage collection. Since its invention, garbage collection has been the subject of decades of computer science research and has become an important feature of numerous computer languages, including Java, C#, and, of course, Ruby.

Computer scientists have invented many different algorithms for performing garbage collection. As it turns out, MRI uses the same GC algorithm John McCarthy invented over 50 years ago: *mark-and-sweep garbage collection*. JRuby and Rubinius, on the other hand, use a different algorithm, invented just a few years later in 1963: *copying garbage collection*. They also employ another innovation called *generational garbage collection* and can even

perform GC tasks in a separate thread while your application continues to run using *concurrent garbage collection*. In this chapter we'll touch on the basic ideas behind these complex GC algorithms. The MRI, JRuby, and Rubinius garbage collectors use more complex versions of these algorithms, but the same fundamental principles apply.

ROADMAP

Garbage Collectors Solve Three Problems

Despite its name, garbage collection is not only the process of cleaning up garbage objects. Garbage collectors, in fact, solve three problems:

- They *allocate* memory for use by new objects.
- They *identify* which objects your program is no longer using.
- They *reclaim* memory from unused objects.

Ruby's GC system is no different. When you create a new Ruby object, the garbage collector allocates memory for that object. Later, Ruby's garbage collector determines when your program has stopped using the object so it can reuse that memory to create new Ruby objects. Allocating memory and reclaiming memory are two sides of the same coin; it makes sense for Ruby's garbage collector to perform both tasks.

Garbage Collection in MRI: Mark and Sweep

A great place to start learning about garbage collection is MRI's relatively simple GC algorithm, which is similar to the one used by John McCarthy in 1960 with his groundbreaking work on Lisp. Once we understand how the algorithm works, we'll look at the more complex garbage collection in JRuby and Rubinius and explore how MRI is adopting some of their techniques.

MRI's *mark-and-sweep* algorithm hands your program memory for new objects until the available memory, or *heap*, is exhausted, at which point MRI stops your program and *marks* the objects that variables or other objects in your code still hold a reference to as *live objects*. Ruby then *sweeps* up the remaining objects, called *garbage objects*, allowing their memory to be reused. Once this process is complete, Ruby allows your program to continue again.

The Free List

Standard MRI Ruby uses McCarthy's original allocation solution, which is called the *free list*. Figure 12-1 shows what a free list looks like conceptually.

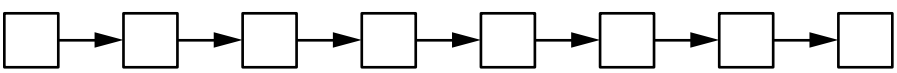

Figure 12-1: A conceptual view of the free list inside MRI

Each white square in the diagram represents a small piece of memory that is available for creating new objects. Think of this diagram as a linked list of unused Ruby objects. When you create a new Ruby object, MRI pulls a free memory block from the head of the list and uses it to create a new Ruby object, as shown in Figure 12-2.

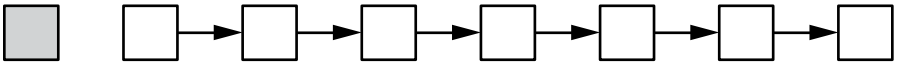

Figure 12-2: Ruby has taken the first memory block from the free list and used it to create a new Ruby object.

The gray box in this figure is an allocated, live object. The remaining white boxes are still available. Internally all Ruby objects are represented by a C structure called `RVALUE`. MRI uses a C *union* inside `RVALUE` to encompass all of the C structures we've seen so far in MRI, such as `RArray`, `RString`, `RRegexp`, and so on. In other words, each square could be any kind of Ruby object or an instance of a custom Ruby class (via `RObject`). The contents of each object, such as the characters in a string, are often stored in a separate memory location.

As your program starts to allocate more new objects, MRI takes more new `RVALUE` structures from the free list, and the list of unused values shrinks, as shown in Figure 12-3.

Figure 12-3: As your program creates more objects, MRI starts to use up the free list.

MRI'S USE OF MULTIPLE FREE LISTS

When MRI starts to execute a Ruby script, it allocates memory for use in the free list. It sets the length of the initial free list to about 10,000 `RVALUE` structures, which means that MRI can create 10,000 Ruby objects without allocating more memory. As more objects are needed, MRI allocates more memory, placing more empty `RVALUE`s onto the free list.

Rather than create a single, long linked list with 10,000 elements, Ruby divides the allocated memory into subsections known as *heaps* in the MRI source code, each about 16k in size. It then creates a free list for each of these heaps, initially creating 24 lists of 407 objects each, using some of the remaining memory for other internal data structures.

Because there are multiple free lists, MRI repeatedly returns `RVALUE` structures from one free list until it's empty and then steps to another free list, returning more structures from that second list. In this way, MRI iterates over the available free lists until they are all empty.

Marking

As your program runs, it creates new objects, and eventually MRI uses up all remaining objects on the free list. At that point, the GC system stops your program, identifies objects that your code is no longer using, and reclaims their memory for allocation to new objects. If no unused objects are found, Ruby asks the operating system for more memory; if there is none to be had, Ruby throws an out-of-memory exception and stops.

Objects that your program allocated but that are no longer being used are known as *garbage objects*. To identify garbage objects, MRI traverses pointers in your objects' C structures, following references from one to another in order to find all active objects (see Figure 12-4). MRI knows your code is no longer using an object if it finds no references to them.

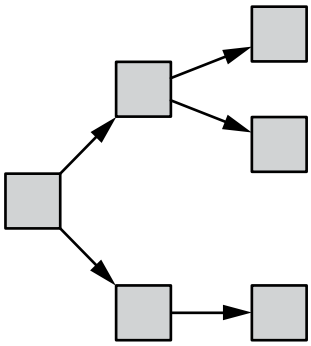

Figure 12-4: Ruby follows pointers, or references, from one object to another, starting with a root object on the left.

The gray box on the left is a *root object*, a global variable that you create or an internal object that Ruby knows your application must be using. There are typically many root objects at any given time. The arrows represent references from this root object to other objects, which in turn may contain references to other objects. This network of objects and references is known as the *object graph*. MRI marks each Ruby object that it finds as it traverses the object graph, stopping your program during the marking process in order to insure that no new object references are created.

Once the marking process completes, the heap contains a series of objects, both marked and unmarked, as shown in Figure 12-5. The marked objects are *live*, which means your code is actively using them. The unmarked objects are garbage, meaning Ruby can release or reclaim their memory. Your code is still using the marked objects, so their memory must be preserved.

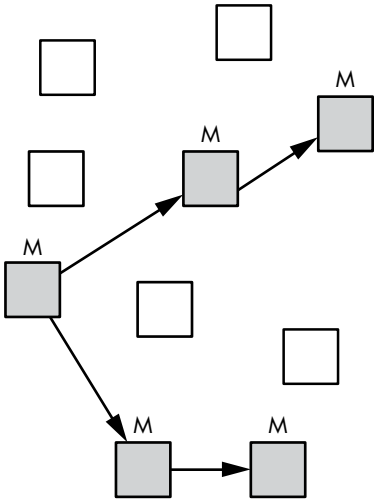

Figure 12-5: MRI has marked five active objects (gray) with five garbage objects remaining in the heap (white).

How Does MRI Mark Live Objects?

MRI saves the information about marked and unmarked objects using a technique known as *bitmap marking*. Bitmap marking refers to the technique

of saving the live object marks as a series of bits in a data structure known as the *free bitmap* (see Figure 12-6). MRI uses a separate memory structure to hold the free bitmap and doesn't save the marks near the objects.

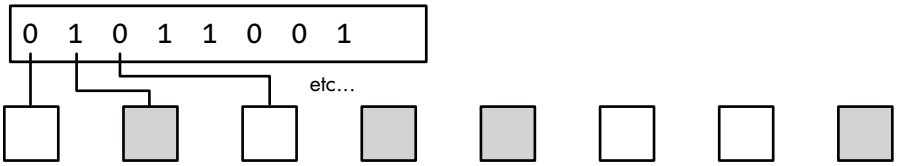

Figure 12-6: MRI saves the GC mark flags in a separate data structure known as the free bitmap.

The reason to use a separate memory structure for the mark bits has to do with a Unix memory optimization technique called *copy-on-write* (see page 265). Similar to how Ruby shares memory between different strings that contain the same letters, copy-on-write allows Unix processes to share memory that contains the same values. By saving the mark bits separately, MRI maximizes the amount of memory that will contain the same values across processes. (In Ruby 1.9 and earlier, the mark bits were saved inside each `RVALUE` structure, causing the garbage collector to modify almost all of Ruby's shared memory while marking live objects and rendering the copy-on-write optimization ineffective.)

Sweeping

Having identified garbage objects, it's time to reclaim them. Ruby's GC algorithm places the unmarked objects back on the free list, as shown in Figure 12-7.

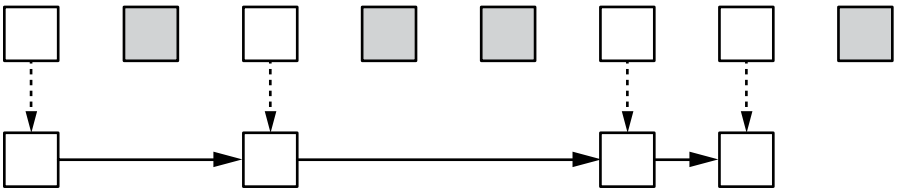

Figure 12-7: While sweeping, MRI places unused `RVALUE` structures back on the free list.

The process of moving unused objects back onto the free list is referred to as *sweeping* the objects. Normally this process runs very quickly because MRI doesn't actually copy objects; it simply adjusts the pointers in each `RVALUE` to create the free linked list (the solid arrows in Figure 12-7).

Lazy Sweeping

Beginning with version 1.9.3, MRI introduced an optimization known as *lazy sweeping*. The lazy sweep algorithm reduces the amount of time a program is stopped by the garbage collector. (Remember, during the normal mark and sweep, MRI stops executing your code.)

Lazy sweeping sweeps only enough garbage objects back to the free list to create a few new Ruby objects and to allow your program to continue, thus reducing the amount of time required to sweep. Ruby sweeps all of the garbage RVALUE objects found in only one of MRI's internal heap structures back to that heap's free list. If no garbage objects are found in the current heap, Ruby tries a lazy sweep on the next heap and works its way through the remaining heaps. (We'll see this algorithm at work in Experiment 12-1.)

Lazy sweeping can reduce the amount of time your program is paused waiting for garbage collection; however, it doesn't reduce the overall amount of garbage collection work to do. Lazy sweeping amortizes the same total amount of sweeping work over multiple GC pauses.

THE RVALUE STRUCTURE

You can find the definition of the RVALUE C structure in the *gc.c* MRI source code file, which contains the implementation of MRI's garbage collector. Listing 12-1 shows part of the RVALUE definition.

```
typedef struct RVALUE {
❶  union {
❷    struct {
       VALUE flags;        /* always 0 for freed obj */
       struct RVALUE *next;
     } free;
❸    struct RBasic  basic;
     struct RObject object;
     struct RClass  klass;
     struct RFloat  flonum;
     struct RString string;
     struct RArray  array;
     struct RRegexp regexp;

--snip--

    } as;
#ifdef GC_DEBUG
    const char *file;
    int   line;
#endif
} RVALUE;
```

Listing 12-1: Part of the RVALUE definition from gc.c

Notice at ❶ that RVALUE uses a union to hold one of many different types of values internally. The first possible value is the `free` structure, defined at ❷, which represents RVALUEs still on the free list. MRI includes every other possible type of Ruby object in the union starting at ❸: `RObject`, `RString`, and so forth.

Disadvantages of Mark and Sweep

The chief disadvantage of mark and sweep is that it requires your program to stop and wait while the marking and sweeping processes take place. Beginning with version 1.9.3, however, MRI's lazy sweeping technique shortens the GC pauses somewhat.

Another disadvantage is that the time required to perform a mark-and-sweep garbage collection is proportional to the total size of the heap. During the marking phase, Ruby needs to visit every active object in your program. During the sweeping phase, Ruby needs to iterate over all of the unused garbage objects left in the heap. As the number of objects created by your program and the total heap size grows, both tasks become more time intensive.

The final issue with mark and sweep is that all of the free list elements—all of the unused objects available for your program to use—must be the same size. MRI doesn't know ahead of time when you allocate a new object whether it will be a string, an array, or a simple number. This is why the `RVALUE` structure MRI uses in the free list must encompass any possible type of Ruby object.

Experiment 12-1: Seeing MRI Garbage Collection in Action

You've learned how the MRI GC algorithm works at a theoretical level. Let's switch gears now to see how MRI performs actual garbage collection. The script in Listing 12-2 creates 10 Ruby objects.

```
10.times do
  obj = Object.new
end
```

Listing 12-2: Creating 10 Ruby objects using `Object.new`

If it's true that MRI assigns unused space from the free list to new objects, Ruby should remove 10 `RVALUE` structures from the free list and assign them to these 10 new objects when we run Listing 12-2. To see this in action, we use the `ObjectSpace#count_objects` method, as shown in Listing 12-3.

```
def display_count
❶  data = ObjectSpace.count_objects
❷  puts "Total: #{data[:TOTAL]} Free: #{data[:FREE]} Object: #{data[:T_OBJECT]}"
end

10.times do
  obj = Object.new
❸  display_count
end
```

Listing 12-3: Using `ObjectSpace#count_objects` to display information about MRI's heap

Now we call `display_count` at ❸ each time around the loop. `display_count` uses `ObjectSpace#count_objects` at ❶ to display information at ❷ about the total number of objects, the number of free objects, and the number of `RObject` structures active each time around the loop.

Running Listing 12-3 gives the output shown in Listing 12-4.

```
Total: 17491 Free: 171 Object: 85
Total: 17491 Free: 139 Object: 86
Total: 17491 Free: 132 Object: 87
Total: 17491 Free: 125 Object: 88
Total: 17491 Free: 118 Object: 89
Total: 17491 Free: 111 Object: 90
Total: 17491 Free: 104 Object: 91
Total: 17491 Free: 97 Object: 92
Total: 17491 Free: 90 Object: 93
Total: 17491 Free: 83 Object: 94
```

Listing 12-4: The output produced by Listing 12-3

The `Total:` field displays the value that MRI returns for `ObjectSpace.count_objects[:TOTAL]`. This value (17491) is the total number of objects currently active inside Ruby. It includes objects we create; objects Ruby creates internally while parsing, compiling, and executing our program; and objects on the free list. This number does not change when we create new objects because it already includes the entire free list.

The `Free:` field displays the value returned by `ObjectSpace.count_objects[:FREE]` for the length of the free list. Notice that the value drops by about 7 each time around the loop. We create only one object per iteration, but Ruby creates 6 other objects each time around the loop while running the code in the `display_count` method.

The `Object:` field displays the count of `RObject` structures currently active in Ruby. Notice that this value increases by 1 each time around the loop, even though we don't keep an active reference to the new objects. That is, we don't save the value returned by `Object.new` anywhere. The `RObject` count includes active and garbage objects.

Seeing MRI Perform a Lazy Sweep

Now if we increase the number of iterations from 10 to 30 and rerun Listing 12-3, we see the following output in Listing 12-5.

```
Total: 17493 Free: 166 Object: 85
Total: 17493 Free: 134 Object: 86
Total: 17493 Free: 127 Object: 87
Total: 17493 Free: 120 Object: 88

--snip--

Total: 17493 Free: 29 Object: 101
Total: 17493 Free: 22 Object: 102
Total: 17493 Free: 15 Object: 103
```

```
❶ Total: 17493 Free: 8 Object: 104
❷ Total: 17493 Free: 246 Object: 104
  Total: 17493 Free: 239 Object: 105
  Total: 17493 Free: 232 Object: 106
  Total: 17493 Free: 225 Object: 107
```

Listing 12-5: Running Listing 12-3 with 30 iterations instead of 10

This time the free list count drops to 8 at ❶. Then at ❷ the free count increases to 246, but the object count remains at 104. This must be a full garbage collection. But it's not! If Ruby had collected all available garbage objects, it would have reduced the `RObject` count when it increased the free count because all of our objects become garbage immediately. What's going on here?

This was a lazy sweep. Ruby first marked all active objects, indirectly identifying the garbage ones. Instead of moving all the garbage objects to the free list, however, it swept only a portion of them: the garbage objects it found in one of its internal heap structures. The free count increased, but the `RObject` count remained the same because MRI reused an `RObject` structure created by one of the previous iterations in order to create the new object.

Seeing MRI Perform a Full Collection

We can see the effect of a full garbage collection by triggering one manually with the `GC.start` method (see Listing 12-6).

```
def display_count
  data = ObjectSpace.count_objects
  puts "Total: #{data[:TOTAL]} Free: #{data[:FREE]} Object: #{data[:T_OBJECT]}"
end

30.times do
  obj = Object.new
  display_count
end

❶ GC.start
❷ display_count
```

Listing 12-6: Triggering a full garbage collection

Here, we again iterate 30 times, creating new objects and calling `display_count`. Then, we call `GC.start` at ❶, which triggers MRI to run a full garbage collection. Finally, at ❷ we call `display_count` again to display the same technical information. Listing 12-7 shows the new output.

```
--snip--

Total: 17491 Free: 26 Object: 101
Total: 17491 Free: 19 Object: 102
Total: 17491 Free: 12 Object: 103
❶ Total: 17491 Free: 251 Object: 103
Total: 17491 Free: 244 Object: 104
Total: 17491 Free: 237 Object: 105
Total: 17491 Free: 230 Object: 106
Total: 17491 Free: 223 Object: 107
Total: 17491 Free: 216 Object: 108
Total: 17491 Free: 209 Object: 109
Total: 17491 Free: 202 Object: 110
Total: 17491 Free: 195 Object: 111
Total: 17491 Free: 188 Object: 112
Total: 17491 Free: 181 Object: 113
❷ Total: 17491 Free: 9527 Object: 43
```

Listing 12-7: The output generated by Listing 12-6

Most of Listing 12-7 shows output similar to Listing 12-5. The total remains the same, while the free count gradually decreases. At ❶ we see the lazy sweep occur again, increasing the free count to 251. But at ❷ we see a dramatic change. The total number of objects remains at 17491, but the free count jumps to 9527 and the number of objects reduces dramatically to 43!

From this observation, we know the following:

- The free count increased dramatically at ❷ because Ruby swept all of the garbage objects onto the free list in one large operation. This garbage included the objects our code created in previous iterations as well as objects that Ruby created internally during the parsing and compilation phases.
- The `RObject` count reduced to 43 because all of the objects created in previous iterations were garbage (because we didn't save them anywhere). The 43 count includes only objects Ruby created internally and none of the objects our code created. If we had saved our new objects somewhere, the `RObject` count would have remained the same. (We'll try this next.)

Interpreting a GC Profile Report

So far in this experiment we've allocated just a few objects from the free list. Of course, your Ruby programs will typically create many more than 30 objects. How does MRI's garbage collector behave when we create thousands or even millions of objects? How can you find out how much time is being taken by the garbage collector in a complex Ruby application?

The answer is to use the `GC::Profiler` class. If you enable it, MRI's internal GC code will collect statistics about each GC run. Listing 12-8 shows how to use `GC::Profiler`.

```
❶ GC::Profiler.enable

  10000000.times do
    obj = Object.new
  end

❷ GC::Profiler.report
```

Listing 12-8: Displaying a GC usage profile using `GC::Profiler` (gc-profile.rb)

We first enable the profiler at ❶ by calling `GC::Profiler.enable`. The following code creates 10 million Ruby objects. At ❷ we display the GC profile report by calling `GC::Profiler.report`. Listing 12-9 shows the report generated in Listing 12-8.

```
$ ruby gc-profile.rb
GC 1046 invokes.
Invoke Time(sec)       Use Size(byte)     Total Size(byte)         Total Object      GC Time(ms)
           0.036               690920              700040                17501         0.694000
           0.039               695200              700040                17501         0.433999
           0.041               695200              700040                17501         0.585000
           0.046               695200              700040                17501         0.577000
           0.049               695200              700040                17501         0.466000
           0.051               695200              700040                17501         0.516999
           0.054               695200              700040                17501         0.419000
           0.056               695200              700040                17501         0.535000
           0.059               695200              700040                17501         0.410000
           0.062               695200              700040                17501         0.426999
--snip--
```

Listing 12-9: A portion of the GC profile report generated in Listing 12-8

To save space, I've removed the first column from the report, a simple counter. Here's what the other columns mean:

- *Invoke time* shows when the garbage collection occurred, measured as seconds after the Ruby script started to run.
- *Use size* shows how much heap memory is used by all live Ruby objects after each collection is finished.
- *Total size* shows the total size of the heap after collection—in other words, the memory taken by live objects plus the size of the free list.
- *Total object* shows the total number of Ruby objects, either live or on the free list.
- Finally, *GC time* shows the amount of time each collection took.

Notice in this experiment that, aside from *invoke time*, none of the values change. The amount of memory used by live Ruby objects, the total size of the heap, and the total number of objects all remain the same. This is because we don't save the new Ruby objects anywhere. They all immediately become garbage. The *GC time* value fluctuates somewhat but more or less remains the same. The amount of time required by the collector to sweep all of the new objects back to the free list remains about the same because the collector sweeps about the same number of objects each time.

However, if we save all of the new objects in an array, they will remain live and not become garbage. Listing 12-10 shows code that saves each object into a single, large array.

```
GC::Profiler.enable

❶ arr = []
10000000.times do
❷   arr << Object.new
end

GC.start

GC::Profiler.report
```

Listing 12-10: Saving 10 million Ruby objects in an array (gc-profile-array.rb)

Here, we create an empty array at ❶ and save each of the new objects in it at ❷. Because the array holds a reference to all of the new objects, they remain active. The garbage collector can't reclaim memory from any of them. Listing 12-11 shows the GC profile report produced by Listing 12-10.

```
$ ruby gc-profile-array.rb
❶ GC 17 invokes.
Invoke Time(sec)       Use Size(byte)     Total Size(byte)         Total Object      GC Time(ms)
           0.031              690920              700040                17501         0.575000
           0.034              708480              716320                17908         0.689000
           0.037             1261680             1269840                31746         1.077000
           0.043             2254280             2262920                56573         1.994999
           0.054             4044200             4053720               101343         3.454999
           0.074             7266080             7277160               181929         5.288000
           0.108            13058920            13072840               326821         9.417000
           0.170            23489240            23508320               587708        14.465000
           0.279            42267080            42311720              1057793        26.015999
           0.478            76096560            76157840              1903946        45.910000
```

Listing 12-11: Ruby has to increase the heap size to accommodate all the new, live objects.

This time the profile report is very different! The garbage collector can't free any of the new objects because they remain active in the array. This means Ruby has no choice but to repeatedly allocate more memory to hold them. When you read Listing 12-11, notice that all three important values—*use size*, *total size*, and *total object*—increase exponentially. This increase is

why at ❶ we see the garbage collector was called only 17 times. (Ruby also ran a few collections before we called `GC::Profiler.enable` as it parsed and compiled our script.) Each time the collector more or less doubled the size of the heap, allowing the script to continue to run for longer and longer periods of time. Instead of running many collections quickly, as we saw in Listing 12-9, Ruby ran just a few slow collections.

If we draw a graph of the time required for each collection (*GC Time*) against the total size of the heap (*Total Heap Size*), as shown in Figure 12-8, we can draw another interesting conclusion.

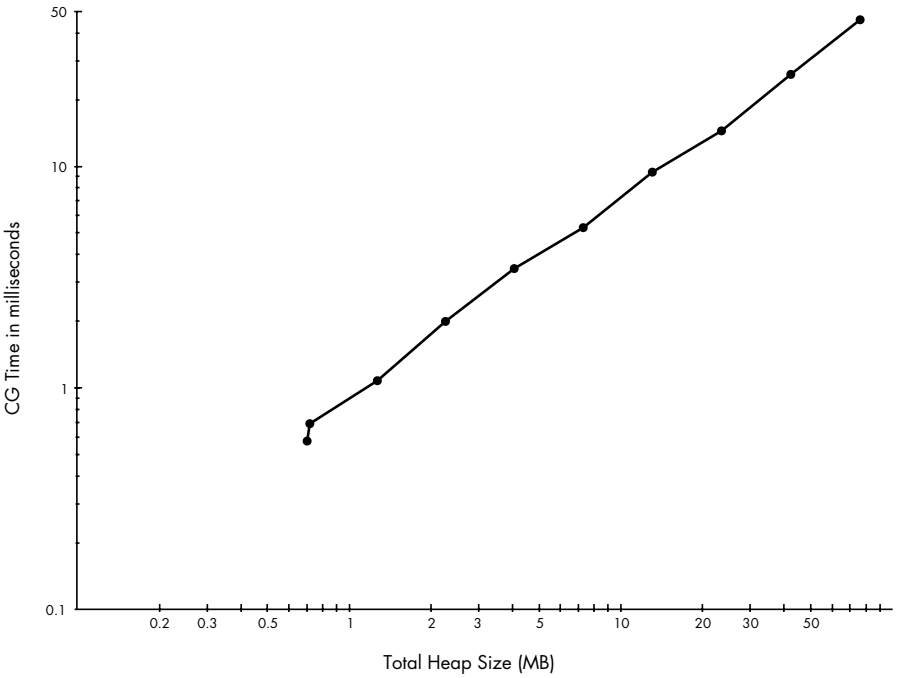

Figure 12-8: The time required to perform mark and sweep increases linearly with the heap size.

Figure 12-8 uses a logarithmic scale for both the x-axis (*Total Heap Size*) and the y-axis (*GC Time*). Because Ruby doubled the heap size during each collection, the data points are more or less evenly spaced across the logarithmic x-axis scale. They are also evenly spaced along the logarithmic y-axis because the time increases exponentially.

Most importantly, note the data points form a straight line: This straight line means the time required to perform a garbage collection increases linearly as a function of the total heap size. As you create more Ruby objects, it takes longer to mark them. Sweeping also takes longer when there are more garbage objects; however, in this example, we don't see any sweep time because all our objects remain live.

Garbage Collection in JRuby and Rubinius

Because JRuby uses the Java Virtual Machine (JVM) to implement Ruby, it's able to use the JVM's sophisticated GC system to manage memory for Ruby objects. In fact, garbage collection is one of the primary benefits of using the JVM platform: The JVM garbage collector has been refined over many years.

The Rubinius C++ virtual machine also includes a sophisticated, efficient garbage collector that uses some of the same underlying algorithms as the JVM. One of the benefits of choosing Rubinius as your Ruby platform is its sophisticated GC system.

The garbage collectors used by JRuby and Rubinius differ from MRI's garbage collector in three ways:

- Instead of using a free list, they allocate memory for new objects and reclaim memory from garbage objects using an algorithm called *copying garbage collection*.
- They handle old and young Ruby objects differently using *generational garbage collection*.
- They use *concurrent garbage collection* to perform some GC tasks at the same time that your application code is running.

NOTE *Although the GC systems used by JRuby and Rubinius are dramatically different from MRI's mark-and-sweep garbage collector, MRI has begun to incorporate some of these ideas as well. Specifically, the GC system in Ruby 2.1 has begun to use generational garbage collection.*

In the following sections, we'll explore the basic algorithms underpinning copying, generational, and concurrent garbage collection, as we learn more about how garbage collection works in Rubinius and JRuby.

Copying Garbage Collection

In 1963, three years after John McCarthy built the first Lisp garbage collector, Marvin Minsky developed a different way of allocating and reclaiming memory known as *copying garbage collection*. (Minsky's research was also originally used for Lisp. The algorithm was later refined by Fenichel and Yochelson in 1969 and by Baker in 1978.) Instead of using a free list to track available objects, copying garbage collectors allocate memory for new objects from a single large heap or memory segment. When that memory segment is used up, these collectors *copy* only the live objects over to a second memory segment, leaving the garbage objects behind. The two segments are then swapped, immediately reclaiming all of the memory from the garbage objects. (Rubinius and the JVM both use complex algorithms based on this original idea.)

Bump Allocation

When you allocate memory for a new object using a copying garbage collector, such as the collectors in the JVM and Rubinius, the garbage collector uses an algorithm called *bump allocation*. Bump allocation allocates adjacent memory segments from a large, continuous heap by *bumping*, or incrementing, a pointer to keep track of where the next allocation will occur. Figure 12-9 shows how this process works for three repeated allocations. (The large rectangle represents the Rubinius or JVM heap.)

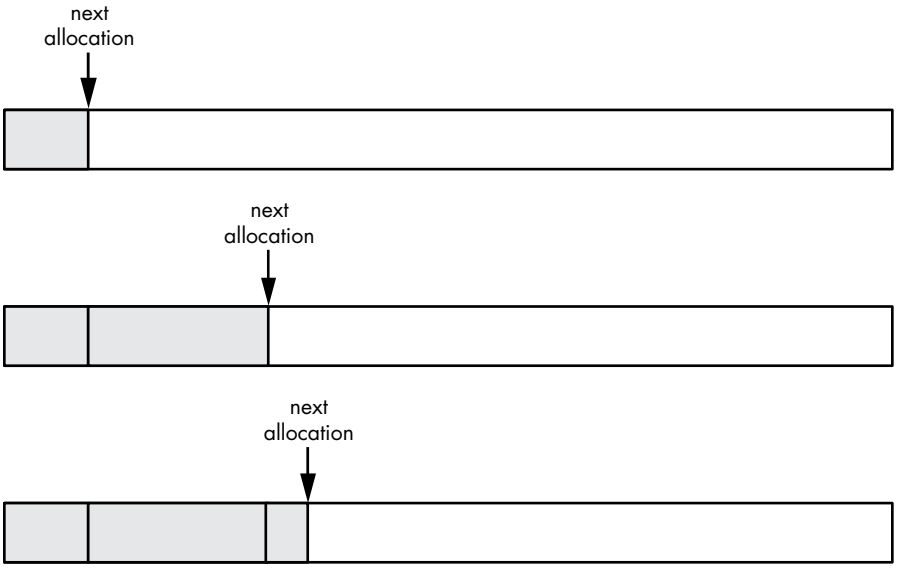

Figure 12-9: Allocating three objects using bump allocation

A copying collector keeps a pointer that tracks where in the heap the next allocation will occur. Each time the collector allocates memory for a new object, it returns some memory from the heap and moves this pointer to the right. As more objects are created, the memory allocated from the heap also moves to the right. Notice, too, that the new objects are not all the same size; each object uses a different number of bytes. As a result, the objects are not spaced evenly across the heap.

The advantages of this technique are that it's very fast and simple to implement and it provides good *locality of reference*, meaning that related values in your program should be located near each other in memory. Locality is important because if your code repeatedly accesses the same area of memory, your CPU can cache that memory and access it much more quickly. If your program often accesses very different areas of memory, the CPU must continually reload the memory cache, slowing down your program's performance.

Another benefit of copying garbage collection is the ability to create objects of different sizes. Unlike the `RVALUE` structure in MRI, JRuby and Rubinius can allocate new objects of any size.

The Semi-Space Algorithm

The real benefit and elegance of copying garbage collectors becomes evident when the initial heap is used up and a garbage collection occurs. Copying garbage collectors identify live and garbage objects the way that mark-and-sweep collectors do—by traversing the object graph following object references or pointers. Once the garbage objects have been identified, however, copying garbage collectors work very differently.

Copying garbage collectors actually use two heaps: one to create new objects with bump allocation and a second, empty one, as shown in Figure 12-10.

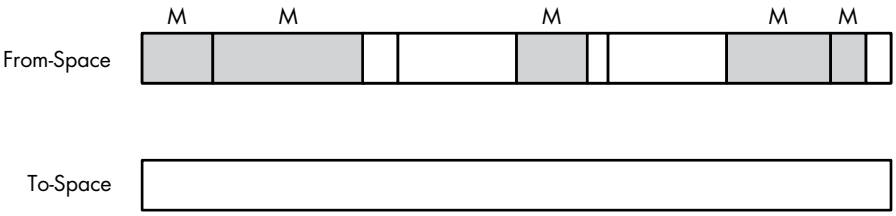

Figure 12-10: The semi-space algorithm uses two heaps, one initially empty.

The heap at the top contains the objects already created and is known as the *from-space*. Note that the objects in the from-space were already marked as live (gray with an *M*) or garbage (white). The lower heap is the *to-space*, and it's initially empty. The algorithm I'm about to describe is known as the *semi-space* algorithm because the total available memory is divided between the from-space and the to-space.

When the from-space becomes completely full, copying garbage collectors copy all of the live objects down into the to-space, leaving the garbage objects behind. Figure 12-11 shows the copying process.

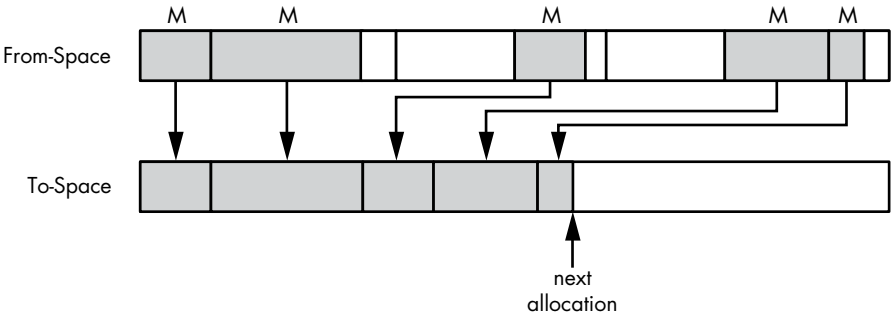

Figure 12-11: The semi-space algorithm copies only live objects to the second heap.

The from-space again appears at the top of the diagram and the to-space below. Notice how the live objects are copied down into the to-space. The arrows pointing down indicate this copying process. A pointer similar to the one used for bump allocation keeps track of where the next live object should be copied to.

Once the copying process is finished, the semi-space algorithm swaps heaps, as shown in Figure 12-12.

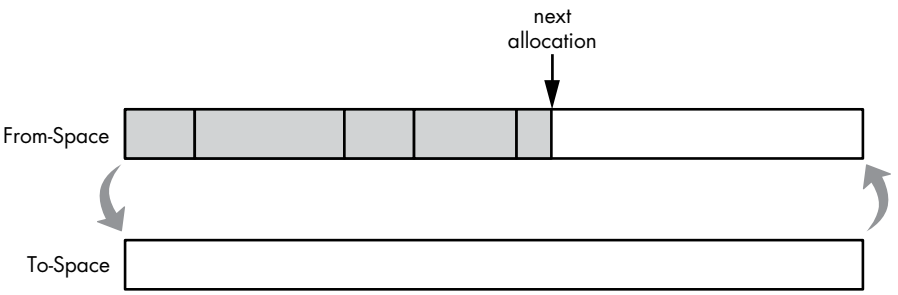

Figure 12-12: After copying the live objects, the semi-space algorithm switches heaps.

In Figure 12-12, the to-space has become the new from-space and is now ready to allocate more memory for new objects using bump allocation. You might expect the algorithm to be slow because so much copying is involved, but it's not, because only active, live objects are copied. Garbage objects are left in place and then reclaimed.

NOTE *All of the live objects were copied to the left side of the heap; this allows the garbage collector to allocate the remaining unused memory most efficiently. This compaction of the heap is a natural result of the semi-space algorithm.*

While the semi-space algorithm is an elegant way to manage memory, it is somewhat memory inefficient. It requires the collector to allocate twice as much memory as it actually uses because all of your objects might remain active and could be copied into the second heap. The algorithm is also somewhat difficult to implement because when the collector moves live objects, it also has to update references and pointers to them internally.

The Eden Heap

As it turns out, both Rubinius and the JVM use a variation of the semi-space algorithm with a third heap structure for allocating new objects called the *Garden of Eden,* or *Eden heap*. Figure 12-13 shows the three memory structures.

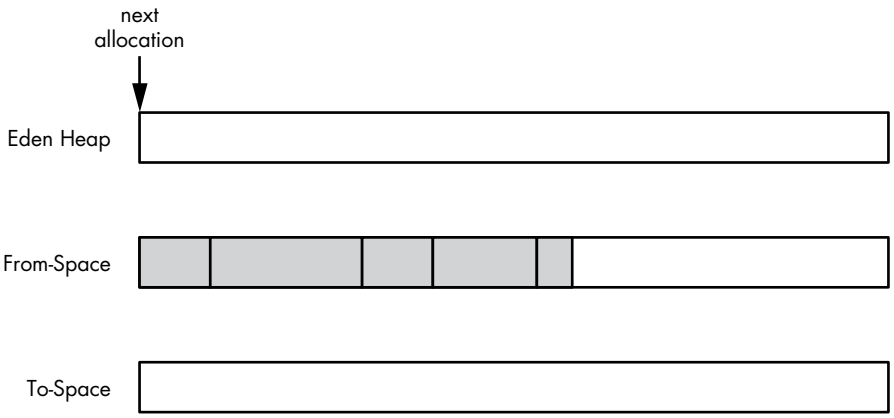

Figure 12-13: The Eden heap is for allocating memory for brand-new objects.

The Eden heap is where the JVM and Rubinius allocate memory for new objects; the from-space contains all of the live objects copied in the previous garbage collection process; and the to-space remains empty until the next garbage collection runs. Each time the garbage collection process runs, the collector copies your objects from both the Eden heap and from-space into the to-space, thereby allowing more memory to be available for new objects because the Eden heap will always be empty after each semi-space copy operation.

Generational Garbage Collection

Many modern garbage collectors, including the collectors in the JVM and the Rubinius VM, use *generational GC* algorithms, a technique that treats new objects differently than older ones. A new, or *young*, object is one that your program has just created, while an old, or *mature*, object is one that your program is continuing to use. The time that an object has to remain active for in order for it to be considered mature is usually measured by the number of times the garbage collection system has run.

The Weak Generational Hypothesis

The reason objects are categorized as either young or mature is based on the assumption that most young objects will have a short lifetime while mature objects are likely to continue to live for a long time. This assumption is known as the *weak generational hypothesis*. In simple terms, new objects are likely to die young. Because young and mature objects have different life expectancies, different GC algorithms are appropriate for each category, or *generation*.

For example, consider a Ruby on Rails website. To generate a web page for each client request, a Rails application creates many new Ruby objects. However, once a web page has been generated and returned to the client, all of those Ruby objects are no longer needed and the GC system can reclaim their memory. At the same time, the application might also create a few Ruby objects that live between requests, such as ones that represent a controller, some configuration data, or a user session. These few mature objects would have a longer lifetime.

Using the Semi-Space Algorithm for Young Objects

According to the weak generational hypothesis, young objects are created continually by your program but also become garbage quite frequently. Because of this, both the JVM and Rubinius run the GC process more frequently for young objects than for mature ones (you'll see just how much more frequently in Experiment 12-2). The semi-space algorithm is ideal for young objects because it copies only live objects. When the Eden heap fills up with new objects, the garbage collector identifies most of them as garbage because new objects usually die young. Because there are fewer live objects, the collector has less copying to do. The JVM refers to these objects as *survivors* and calls the from-space and the to-space *survivor spaces*.

Promoting Objects

When a new object becomes old (that is, when it has survived a certain number of runs of the GC system), it is *promoted*, or copied, into the mature generation heap during the semi-space copy process, as shown in Figure 12-14.

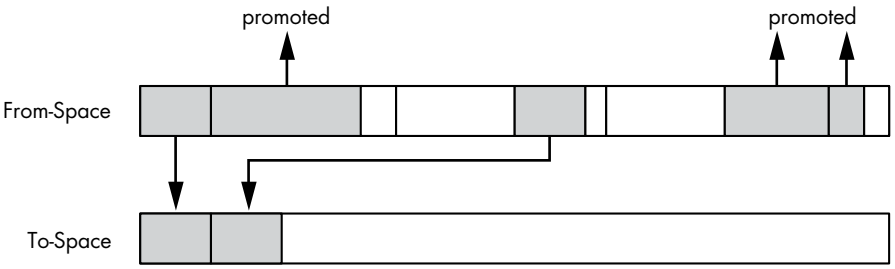

Figure 12-14: Generational garbage collectors promote old objects from the young heap to the mature one.

Notice that the from-space contains five active objects, shown as gray rectangles. Two of these are copied down to the to-space by the semi-space algorithm, but the other three are promoted. Their age has exceeded the *new object lifetime* because they have remained active for a certain number of GC runs.

In Rubinius, the new object lifetime is set to 2 by default, meaning that a young object becomes mature once the GC system has run twice with your code still holding a reference to that object. (This means that Rubinius will copy a live object twice between the from- and to-space, using the semi-space algorithm.) Over time, Rubinius adjusts the object lifetime value, based on various statistics, to optimize garbage collection as much as possible.

The JVM's garbage collector internally calculates the new object lifetime, attempting to keep the from- and to-space heaps about half full. If these heaps start to fill up, the new object lifetime will decrease, and objects will be promoted more quickly. If the spaces are mostly empty, the JVM will increase the new object lifetime, allowing new objects to remain there longer.

Garbage Collection for Mature Objects

Once your objects are promoted into the mature collection, they will likely live on for a long time due to the weak generational hypothesis. As a result, both the JVM and Rubinius need to run garbage collection on the mature generation much less frequently. Garbage collection on the mature generation runs once the heap allocated for mature objects fills up. Because most new objects don't live past the new object lifetime, the mature collection fills up slowly.

The JVM offers many command-line options that allow you to configure the relative or absolute sizes of young and mature generation heaps (the JVM documentation refers to the mature generation as the *tenured generation*). The JVM also maintains a third generation for internal objects created by the JVM itself: the *permanent generation*. Garbage collection on the young generation is called a *minor collection*, and on the tenured generation, it's a *major collection*.

Rubinius uses a sophisticated GC algorithm called *Immix* for the mature generation of objects. Immix attempts to reduce the amount of total memory used and the amount of heap fragmentation by collecting active objects into continuous regions. Rubinius also uses a third generation for very large objects and collects them using a standard mark-and-sweep process.

NOTE *MRI Ruby version 2.1 implements a generational GC algorithm for standard Ruby like the one the JVM and Rubinius have used for years. Its primary challenge is also detecting which mature objects reference young ones (see "References Between Generations" on page 316). MRI uses a solution common to many implementations of generational GC: it tracks each time a mature object references a young one using write barriers. However, implementing write barriers in MRI is complex because existing C extensions won't contain them.*

REFERENCES BETWEEN GENERATIONS

In addition to the new object lifetime, generational garbage collectors have to track another important detail: young objects that are active because of a reference from an old object. Because collections on the young generation will not mark mature objects, the collector might assume that certain young objects are garbage when they are not. Figure 12-15 shows an example of the problem.

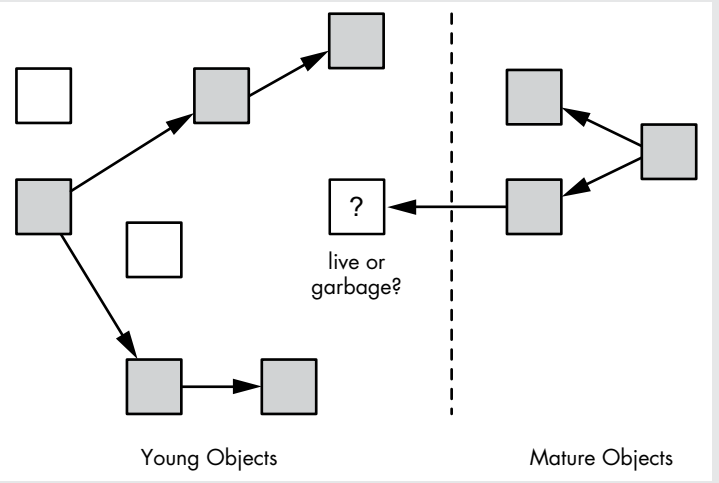

Figure 12-15: Generational garbage collectors need to find mature objects that reference young objects.

The young collection contains several live objects (gray) and garbage objects (white). During the young object marking phase, the generational garbage collector follows only references from young objects in order to speed up the process, which occurs frequently. Notice, however, the center object marked with a question mark: Is it live or garbage? There are no references to it from other young objects, but there is a reference to it from a mature object. If Rubinius or the JVM were to run the semi-space algorithm on the young objects at left after marking them, the center object would be incorrectly considered garbage and its contents overwritten!

Write Barriers

Generational garbage collectors can solve this problem using *write barriers*. These are bits of code that keep track of when your program adds a reference from a mature object to a young one. When the garbage collector encounters such a reference, it considers that one mature object to be another root for use in marking young objects, thereby allowing the object in question to be considered live and to be copied properly by the semi-space algorithm.

Concurrent Garbage Collection

Both Rubinius and the JVM use another sophisticated technique to reduce the amount of time your application spends waiting for garbage collection: *concurrent garbage collection*. When using concurrent garbage collection, the garbage collector runs at the same time as your application code. This eliminates, or at least reduces, pauses in your program due to garbage collection because your application doesn't have to stop and wait while the garbage collector runs.

Concurrent garbage collectors run in a separate thread from the primary application. Although in theory this could mean that your application will slow a bit because part of the CPU's time has to be spent running the GC thread, most computers today contain microprocessors with multiple cores, which allow different threads to run in parallel. This means one of the cores can be dedicated to running the GC thread, leaving the other cores to run the primary application. (In practice, this still might slow down your application because fewer cores are available.)

NOTE *MRI Ruby 2.1 also supports a form of concurrent garbage collection by performing the sweep portion of the mark-and-sweep algorithm in parallel while your Ruby code continues to run. This helps to reduce the amount of time your application is paused while garbage collection runs.*

Marking While the Object Graph Changes

Marking objects while your application is running presents one large obstacle for concurrent garbage collectors: What if your application changes the object graph while the collector is marking it? To better understand this problem, see the example object graph in Figure 12-16.

This figure shows a small set of objects being marked by a concurrent garbage collector. On the left is a root object, and to the right are various child objects referenced by the root object. All of the live objects are marked with *M* and shown in gray. The garbage collector, indicated by the large arrow, has already marked the live objects and is now processing the objects near the bottom. The collector is about to mark the two remaining white objects at the bottom right.

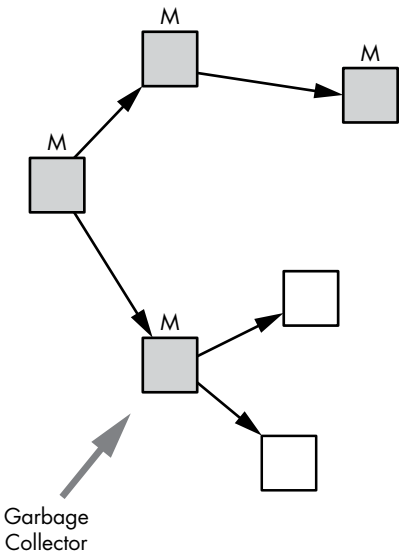

Figure 12-16: A garbage collector marking an object graph

Now suppose your application, which is also running while the marking process is underway, creates a new object and adds it as a child of one of the previously marked objects. Figure 12-17 shows the new situation.

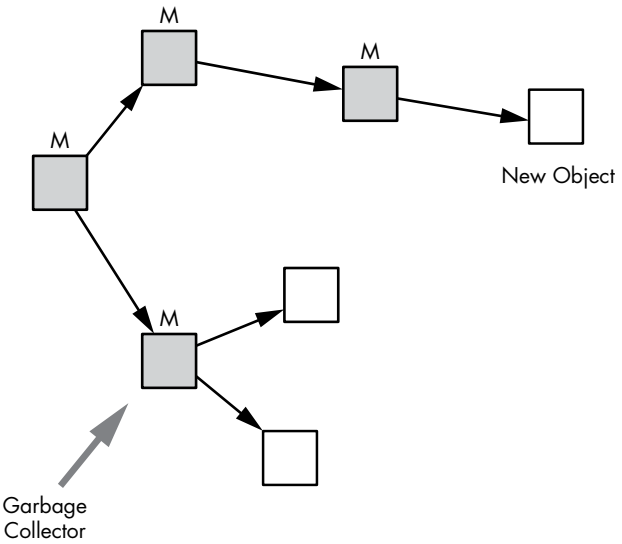

Figure 12-17: Your application creates a new object while the marking process is underway.

Notice that one of the live, marked objects points to a new object that hasn't been marked yet.

Now suppose the garbage collector finishes marking the object graph. It has marked all of the live objects, meaning that any remaining objects are assumed to be garbage. Figure 12-18 shows how the object graph appears at the end of the marking process.

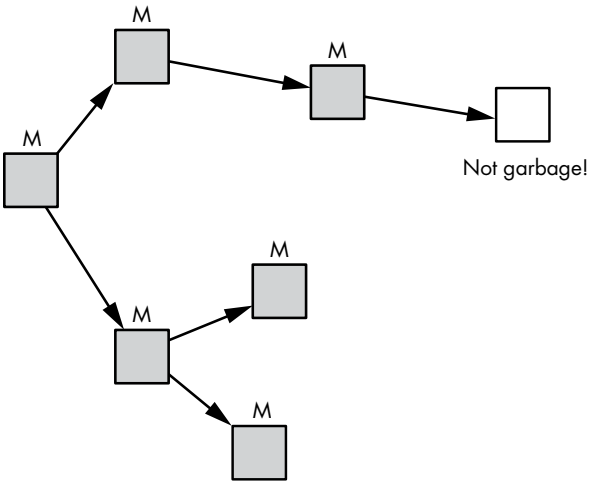

Figure 12-18: The collector incorrectly considers the new live object to be garbage.

The garbage collector has finished marking all live objects, but it missed the new object. The collector will now reclaim its memory, but the application will have lost valid data or will have garbage data added to one of its objects!

Tricolor Marking

The solution to this problem is to maintain a *mark stack*, or a list of objects that still need to be examined by the marking process, as shown in Figure 12-19.

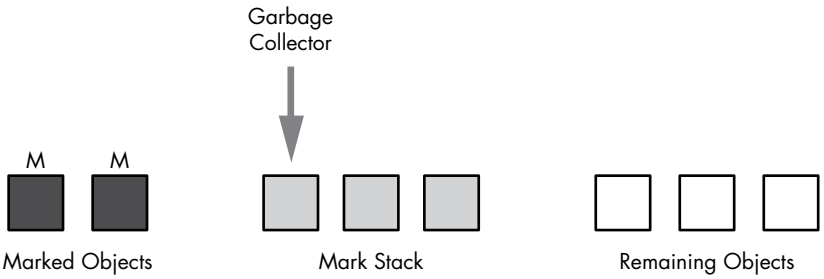

Figure 12-19: The marking process works through the objects in the mark stack.

Initially all of the root objects are placed on the mark stack. As the garbage collector marks objects, it moves them from the mark stack to the list of marked objects on the left, and it adds any child objects it finds to the mark stack. When the mark stack is exhausted, the garbage collector is finished; it has identified all live objects and any remaining objects on the right are assumed to be garbage. But with this scheme, if the application modifies one of the objects during marking, the collector can move the modified object back to the mark stack, even if it was previously marked, as shown in Figure 12-20.

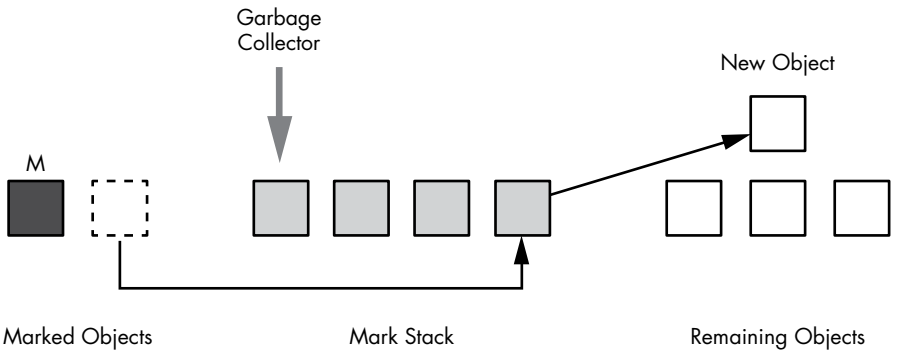

Figure 12-20: The collector moves a marked object back to the mark stack because the application modified it.

The application has added a new object to the system, as shown at right in the figure's remaining objects list. This time, however, the collector notices that an existing object was modified because it now contains a

reference to the new object and it moves the modified object to the mark stack in the center. As a result, the collector will eventually find and mark the new object as it works through the mark stack.

This modified marking algorithm is known as *tricolor marking*: Objects already processed are considered "black"; objects on the mark stack, "gray"; and the remaining objects, "white," as shown in Figures 12-19 and 12-20.

NOTE *Concurrent garbage collectors can use write barriers to detect when an application changes the object graph. Write barriers are used by both generational and concurrent garbage collectors.*

Three Garbage Collectors in the JVM

In order to support different types of applications and server hardware, the JVM includes three separate garbage collectors that implement concurrent garbage collection differently. You can use command-line parameters to choose which collector to run in your JRuby program. The three collectors are as follows:

Serial This collector stops your application and performs garbage collection while your application is waiting. It doesn't use concurrent garbage collection at all.

Parallel This collector performs many GC tasks, including minor collections, in a separate thread while your application is running.

Concurrent This collector performs most GC tasks in parallel with your application. It's optimized to reduce GC pauses as much as possible, but its use may slow down your application's overall throughput.

NOTE *In addition to these three, a variety of new, experimental garbage collectors are also available for the JVM. One of these is the garbage-first (G1) collector, and another is the continuously concurrent compacting (C4) collector.*

Unless you direct it to do otherwise, the JVM automatically selects one of these garbage collectors, depending on the type of hardware being used. For most computers, the JVM uses the parallel collector by default; for server-class machines, it uses the concurrent collector instead. You can change the JVM's default garbage collection choice by using command-line options when you start your JRuby program. See the article "Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning" (*http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html*) for more details.

The ability to choose from these different GC algorithms and to further tune the behavior of the collector using many other configuration options is one of the great benefits of using JRuby. The effectiveness and performance of a garbage collector depends on your application's behavior as well as the underlying algorithms used.

To help make sense of the myriad GC-related options provided by the JVM, Charles Nutter, one of the lead developers behind the JRuby project, suggests using the following rules of thumb:

- When in doubt, stick with the JVM's default settings. These settings work well in most cases.
- If you have a lot of data that need to be collected frequently or periodically, the concurrent or experimental G1 collectors may do a better job than the parallel collector.
- Try to improve your code so it uses less memory before tuning garbage collection. Tuning the JVM's garbage collector when you are allocating too much memory solves only half the problem.

Experiment 12-2: Using Verbose GC Mode in JRuby

Experiment 12-1 explored garbage collection in MRI. In this experiment, we'll see how garbage collection works in JRuby by asking the JVM to display technical information about what the JVM's garbage collector is doing. Listing 12-12 shows the code from Experiment 12-1 that creates 10 Ruby objects.

```
10.times do
  obj = Object.new
end
```

Listing 12-12: Creating 10 Ruby objects using `Object.new` (jruby-gc.rb)

When we run this simple program using the `-J-verbose:gc` option, the JVM displays internal debugging information about garbage collection. Here's the command to use:

```
$ jruby -J-verbose:gc jruby-gc.rb
```

But this command doesn't produce any output. Perhaps we aren't creating enough objects to trigger a garbage collection.

Let's increase the number of new objects to 10 million, as shown in Listing 12-13.

```
10000000.times do
  obj = Object.new
end
```

Listing 12-13: Creating 10 million Ruby objects using `Object.new` (jruby-gc.rb)

The new output is shown in Listing 12-14.

```
$ jruby -J-verbose:gc jruby-gc.rb
[GC 17024K->1292K(83008K), 0.0072491 secs]
[GC 18316K->1538K(83008K), 0.0091344 secs]
[GC 18562K->1349K(83008K), 0.0006953 secs]
[GC 18373K->1301K(83008K), 0.0006876 secs]
[GC 18325K->1289K(83008K), 0.0004180 secs]
[GC 18313K->1285K(83008K), 0.0006950 secs]
[GC 18309K->1285K(83008K), 0.0006597 secs]
[GC 18309K->1285K(83008K), 0.0007186 secs]
[GC 18309K->1285K(83008K), 0.0005617 secs]
[GC 18309K->1285K(83008K), 0.0006873 secs]
[GC 18309K->1285K(83008K), 0.0004944 secs]
[GC 18309K->1285K(83008K), 0.0006644 secs]
[GC 18309K->1285K(83008K), 0.0006448 secs]
[GC 18309K->1285K(83008K), 0.0007203 secs]
```

Listing 12-14: The output produced by running Listing 12-13 with `-J-verbose:gc`

The JVM displays a line of information each time garbage collection occurs while running our Ruby program. There are 14 GC events shown here. Each line contains the following information:

`[GC...` The *GC* prefix means this event was a minor collection. The JVM cleaned up only new objects in the Eden heap or young objects in the survivor spaces.

`17024K->1292K` These values show the amount of data used by live objects before (left of the arrow) and after (right of the arrow) the garbage collection. In this example, the amount of space taken up by live objects in the young collection dropped from about 17MB or 18MB to about 1.3MB each time.

`(83008K)` The value in parentheses shows the total size of the JVM heap for this process. This value has not changed.

`0.0072491 secs` This value shows the amount of time taken to perform each garbage collection.

Listing 12-14 shows that the JVM's young heap repeatedly fills up as we create more Ruby objects. Notice that each time the JVM garbage collector usually takes less than 1 millisecond to clean up the many thousands of garbage objects.

Notice, too, that there were no major garbage collections. Why? Because we don't save our Ruby objects. Listing 12-13 creates 10 million objects but doesn't use them, so the JVM's garbage collector determines that they are all garbage and reclaims their memory immediately before they are promoted to become mature objects.

Triggering Major Collections

In order to trigger major collections, we need to create some mature objects by creating Ruby objects that don't die young but that live on for some time. We can achieve this by saving our new objects in an array, as we did in Experiment 12-1. Listing 12-15 repeats the same script again here for convenience.

```
❶ arr = []
  10000000.times do
❷   arr << Object.new
  end
```

Listing 12-15: Saving 10 million Ruby objects in an array

Notice at ❶ that we create an empty array and then insert all 10 million new objects into it at ❷. Because the array contains a reference to all objects, the objects will all remain live.

Now let's rerun our experiment using the `-J-verbose:gc` command. Listing 12-16 shows the result.

```
$ jruby -J-verbose:gc jruby-gc.rb
❶ [GC 16196K->8571K(83008K), 0.0873137 secs]
  [GC 25595K->20319K(83008K), 0.0480336 secs]
  [GC 37343K->37342K(83008K), 0.0611792 secs]
  [GC 37586K(83008K), 0.0029985 secs]
  [GC 54366K->54365K(83008K), 0.0617091 secs]
  [GC 65553K->65360K(83008K), 0.0586615 secs]
  [GC 82384K->82384K(100040K), 0.0479422 secs]
  [GC 89491K(100040K), 0.0124503 secs]
  [GC 95890K->95888K(147060K), 0.0795343 secs]
  [GC 96144K(147060K), 0.0030345 secs]
  [GC 130683K->130682K(148020K), 0.0941640 secs]
  [GC 147706K->147704K(165108K), 0.0925857 secs]
  [GC 150767K->151226K(168564K), 0.0226121 secs]
❷ [Full GC 151226K->125676K(168564K), 0.5317203 secs]
  [GC 176397K->176404K(236472K), 0.0999831 secs]

  --snip--
```

Listing 12-16: The beginning of the output produced by running Listing 12-15 with `-J-verbose:gc`

Notice at ❷ that the output `[Full GC...]` first appears after 13 young collections. (The output continues past what is shown in Listing 12-16.) This tells us that many Ruby objects were promoted, filling up the mature generation and forcing a mature collection to run.

We can draw some other interesting conclusions from this output. First, the size of the young collection gradually grew from the first GC run at ❶ to the mature collection at ❷. This tells us that the JVM was automatically increasing the total heap size as more objects were created. Notice that the total heap size value in parentheses started at around 83MB and grew to over 200MB, as shown in bold. Also, each young collection was still relatively fast at under 0.1 seconds, though much slower than the ones we saw in Listing 12-14, which took less than 1 millisecond. Remember that the semi-space algorithm copies only live objects. This time all of our Ruby objects remained alive, and the JVM had to copy them repeatedly. Finally, notice that the mature, or full, collection at ❷ took about 0.53 seconds, which was much longer than any of the young collections.

Further Reading

There's a vast amount of information available on the topic of garbage collection. To learn more about John McCarthy's original free list implementation, see his article on Lisp: "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1" (*Communications of the ACM*, 1960).

For a taste of modern GC research, you can read about the Immix algorithm used by Rubinius in Stephen M. Blackburn and Kathryn S. McKinley's "A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance" (*ACM SIGPLAN Notices*, 2008). The following article from Oracle both explains the JVM's overall GC algorithm and serves as a good reference for the many command-line options you can use to customize and tune the JVM's garbage collector's behavior: "Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning" (*http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html*).

Finally, two definitive sources on GC algorithms in general and how they have changed over the years are Jones and Lins's *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* (Wiley, 1996) and Jones, Hosking, and Moss's, *The Garbage Collection Handbook: The Art of Automatic Memory Management* (CRC Press, 2012).

Summary

This chapter has covered one of the most important but least understood areas of Ruby internals: garbage collection. We learned that garbage collectors allocate memory for new objects and clean up unused garbage objects. We examined the basic algorithms used by MRI, Rubinius, and JRuby for garbage collection and discovered that MRI allocates and reclaims memory using a free list, while Rubinius and the JVM use the semi-space algorithm. We also saw how Rubinius and JRuby employ concurrent and generational GC techniques, which MRI starts to use in Ruby 2.1.

But we've only scratched the surface of garbage collection. Since its invention in 1960, many complex GC algorithms have been developed; indeed, garbage collection is still an active area of computer science research. The GC implementations in MRI, Rubinius, and JRuby are likely to continue to evolve and improve over time.