CONTENTS

Building Memory Efficient Java Programs

Nick Mitchell Edith Schonberg Gary Sevitsky

PREFACE

Over the past ten years, we have worked with developers, testers, and performance analysts to help fix memory-related problems in large Java applications. During the many years we have spent studying the performance of Java applications, it has become clear to us that the problems related to memory is a topic worthy of a book. In spite of the fact that computers are now equipped with gigabytes of memory, Java developers face an uphill battle getting their applications to fit in memory. Ten years ago, a 500MB heap was considered big. Now it is not unusual to see applications that are having trouble fitting into 2 to 3 gigabyte heaps, and developers are turning to 64-bit architectures to be able to run with even larger heaps.

Java heaps are not just big, but are often bloated, with as much as 80% of memory devoted to overhead rather than "‘real data"’. This much bloat is an indication that a lot of memory is being used to accomplish little. We have seen applications where a simple transaction needs 500K for the session state for one user, or 1 Gigabyte of memory to support only a few hundred users.

By the time we are called in to help with a performance problem, the situation is often critical. The application is either in the final stages of testing or about to be deployed. Fixing problems this late in the cycle is very expensive, and can sometimes require major code refactoring. It would certainly be better if it were possible to deal with memory issues earlier on, during development or even design.

Why ...? Java developers face some unique challenges when it comes to memory. First, much is hidden from view. A Java developer who assembles a system out of reusable libraries and frameworks is truly faced with an "‘iceberg"’, where a single call or constructor may invoke many layers of hidden framework code. We have seen over and over again how easy it is for space costs to pile across the layers. While there is much good advice on how to code flexible and maintainable systems, there is little information available on space. The space costs of basic Java features and higher-level frameworks can be difficult to ascertain. In part this is by design - the Java programmer has been encouraged not to think about physical storage, and instead to let the runtime worry about it. The lack of awareness of space costs, even among many experienced developers, was a key motivation for writing this book.

By raising awareness of the costs of common programming idioms, we aim to help developers make informed tradeoffs, and to make them earlier.

The design of the Java language and standard libraries can also make it more difficult for programmers to use space efficiently. Java's data modeling features and managed runtime give developers fewer options than a language like C++, that allows more direct control over storage. Taking these features out of the hands of developers has been a huge plus for ease of learning and for safety of the language. However, it leaves the developer who wants to engineer frugally with fewer options. Helping developers make informed decisions between competing options was another aim for the book.

This book is a comprehensive, practical guide to memory-conscious programming in Java. It addresses two different and equally important aspects of using memory well. Much of the book covers how to represent your data efficiently. It takes you through common modeling patterns, and highlights their costs and discusses tradeoffs that can be made. The book also devotes substantial space to managing the lifetime of objects, from very short-lived temporaries to longer-lived structures such as caches. Lifetime management issues are a common source of bugs, such as memory leaks, and inefficiency (mostly performance). Throughout the book we use examples to illustrate common idioms. Most of the examples are distilled from more complex examples we have seen in real-world applications. Throughout the book are also guides to Java mechanisms that are relevant to a given topic. These include features in the language proper, as well as the garbage collector and the standard libraries.

While the book is a collection of advice on practical topics, it is also organized so as to give a systematic approach to memory issues. When read as a whole it can be helpful in seeing the range of topics that need to be considered, especially early in design. That does not mean that one must read the whole book in order, or do a comprehensive analysis of every data structure in your design, in order to get the benefit. The chapters are written to stand on their own where possible, so that if a particular pattern comes up in your code you can quickly get some ideas on costs and alternatives. At the same time, familiarizing yourself with a few concepts in the Introduction will make the reading much easier.

The book is appropriate for Java developers (experienced and novice alike), especially framework and applications developers, who are faced with decisions every day that will have impact down the line in system test and production. It is also aimed at technical managers and testers, who need to make sure that Java software meets its performance requirements. This material should be of interest to students and teachers of software engineering, who would like to gain a better understanding of memory usage patterns in real-world Java applications. Basic knowledge of Java is assumed.

Much of the content relies on knowing or measuring the size of objects at runtime. Sizes vary depending which JRE you are using. Our reference JRE is Sun Java 6 update 14. Unless otherwise stated, all sizes are for this reference JRE. The book is self-contained in that it teaches how to calculate object sizes from scratch. We realize, of course, that this can be a tedious endevour, and so the appendix provides a list of tools and resources that can help with memory analysis. Nevertheless, we belief that performing detailed calculations are pedagogically important.

The book is divided into four parts:

Part 1 introduces an important theme that runs through the book: the health of a data design is the fraction of memory devoted to actual data vs. various kinds of infrastructure. In addition to size, memory health can be helpful for gauging the appropriateness of a design choice, and for comparing alternatives. It can also be a powerful tool for recognizing scaling problems early.

Part 2 covers the choices developers face when creating their physical data models, such as whether to delegate data to separate classes, whether to introduce subclasses, and how to represent sparse data and relationships. These choices are looked at from a memory cost perspective. This section also covers how the JVM manages objects and its cost implications for different designs.

Part 3 is devoted to collections. Collection choices are at the heart of the ability of large data structures to scale. This section covers, through examples, various design choices that can be made based on data usage patterns (e.g. load vs. access), properties of the data (e.g. sparseness, degree of fan-out), context (e.g. nested structures) and constraints (e.g. uniqueness). We look closely at the Java collection classes, their cost in different situations, and some of their undocumented assumptions. We also look at some alternatives to the Java collection classes.

Part 4 covers the topic of lifetime management, a common source of inefficiency, as well as bugs. This section examines the costs of both short-lived temporaries and long-lived structures, such as caches and pools. We explain the Java mechanisms available for managing object lifetime, such as ThreadLocal storage, weak and soft references, and the basic workings of the garbage collector. Finally, we present techniques for avoiding common errors such as memory leaks and drag.

CONTENTS

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Managing your Java program's memory couldn't be easier, or so it would seem. Java provides you with automatic garbage collection, and a compiler that runs as your program is running and responds to its operation. There are many standard, open source, and proprietary libraries available that provide powerful functionality, and are written by experts. All you have to do is piece together the parts and let the Java runtime system do the rest for you.

The reality, unfortunately, is very different. If you just assemble the parts, take the defaults, and follow all the good advice to make your program flexible and maintainable, you will likely find that your memory needs are *much* higher than imagined. You may also find that precious memory resources are wasted holding on to data that is no longer needed, or, even worse, that your system suffers from memory leaks. Java is filled with costly memory traps that are easy to fall into. All too often, these problems won't show up until late in the cycle, when the whole system comes together. You may discover, for example, when your product is about to ship, that your design is far from fitting into memory, or that it does not support nearly the number of users it needs to support. Fixing these problems can take a major effort, requiring extensive refactoring or rethinking architectural decisions, such as the choice of frameworks you use.

This book is a guide to using memory wisely in Java. Memory usage, like any other aspect of software, needs to be carefully engineered. Predicting your system's memory needs early in the cycle, and engineering with those needs in mind, can make the difference between success and failure of your project. Engineering means making informed tradeoffs, and, unfortunately, the information to do so isn't always available. While there has been much written on how to build systems that are bug-free, easy to maintain, and secure, there is little guidance available on how to use Java memory efficiently or even correctly. (Too often memory issues are left to chance, or at least put off until there is a crisis.) This book gives you the tools to make informed choices early on. It gives you an approach to looking at your system's uses of memory, and a practical guide to making informed tradeoffs in every part of your design and implementation. (Common patterns that make up your design - helps you understand costs and alternatives. Also relevant Java mechanisms). Three kinds of info: patterns, mechanisms, and an organization/approach to thinking about memory.

(If you are like most Java developers, you probably don't have a good picture of of how much memory your designs use.)

Achieving efficient memory usage takes some effort in any language. There are things about Java that make it especially easy to end up with bloated or even incorrect data designs. One reason is that Java is so easy to program, so it gives you a false sense of security that everything is being handled for you. (Languages like C give you a lot of control over storage, and so it is clear what the consequences of your choices are.) Also, as we shall see, the basic costs of building blocks such as Strings, can be surprisingly expensive, when compared with languages like C++. So much is done for you in Java, from management of the heap, to all the functionality hidden behind framework APIs, that it can be difficult to find out how much memory your data structures need, or how long they are staying around, until you have a fully running system. Compared to many languages, Java also gives you fewer options for designing your data structures and managing their storage. This means that if you find you have problems, you have fewer ways of fixing them without rethinking larger aspects of your design. All of this makes it important to understand what memory costs and alternatives are, as early as possible.

Beyond the technical realities of using memory well in Java there are some commonly held misconceptions that often make things worse. So before getting into how to engineer memory in Java, it is worth dispelling some of these myths, and getting a better understanding of why memory problems can be so common in Java.

1.1 Facts and Fictions

In addition to the technical reasons why managing memory can be a challenge, there are other reasons why memory footprint problems are so common. In particular, the software culture and popular beliefs can lead you to ignore memory costs. Some of these beliefs are really myths — they might have once been true, but no longer. Here are several.

1.2 Memory-conscious Engineering

At a high level, the approach we present in this book is simple. For each part of your data design: - understand the space needs of the various implementation alternatives - determine how long that data should remain alive, and then choose an appropriate implementation to achieve that

The bulk of this book takes you through the most common patterns that come up in practice for each topic. We show you how to recognize these patterns in your designs, and give you relevant information about costs and other pitfalls.

discuss looking at each data structures separately

1.2.1 Estimating Space Costs

discuss estimating (Edith text on counting bytes, etc is great) (including scalability discussion) (and focusing on important stuff)

discuss health very briefly

discuss entities and collections

Entity-Collection Diagrams

Much of this book is about how to implement your data designs to make the most efficient use of space. In this section we introduce a diagram, called the *entity-collection(E-C) diagram*, that helps with that process. It highlights the major elements of the data model implementation, so that the costs and scaling consequences of the design are easily visible. We use these diagrams throughout the book to illustrate various implementation options and their costs.

A data model implementation begins with a conceptual understanding of the entities and relationships in the model. This may be an informal understanding, or it may be formalized in a diagram such as an E-R diagram or a UML class diagram. At some point that conceptual model is turned into Java classes that represent the entities, attributes, and assocations of the model, as well as any auxiliary structures, such as indexes, needed to access the data. The example below shows a simple conceptual model, using a UML class diagram. A Java implementation of that model is also shown, using rectangles for classes and arrows for references.

In these models we make a distinction between the implementations of entities and the implementation of collections. We do this for a number of reasons. First, the kinds of choices you make to improve the storage of your entities are often different from those Collections are also depicted as nodes, using an octagonal shape. This is different from UML class diagrams, where associations are shown as edges. E-C diagrams show

1.2.2 Managing lifetime

discuss overview of approach to lifetime management

1.2.3 Defining Terms

Terms like object can have different meanings in the literature. The following are the conventions used throughout this book.

Since the word *object* can have different meanings, we precisely define the terminology used:

- A *class* is a Java class. A class name, for example `String`, always appears in type-writer font.
- A *data model* is a set of classes that represents one or more logical concepts.

- Finally, an *object* is an instance of a class, that exists at runtime occupying a contiguous section of memory.

Part I

Using Space

Chapter 2

MEMORY HEALTH

A data structure may be big, but how do you know if it is too big? This chapter introduces the concept of memory health, a way to gauge how efficiently a data structure uses memory. Java developers often have to piece together many smaller objects to build a single data structure. Memory health is a way to understand the overhead introduced in each object, the impact of connecting these objects into larger data structures, and, finally, how a data structure scales as the amount of data it holds grows. Understanding memory health can provide valuable insight into each step of the data modeling process.

Later chapters show how to estimate both the size and health of data structure designs. Both of these are important. Memory health is independent of size: a small structure can be unhealthy, and a large structure can be healthy. A healthy data structure will generally scale well, while unhealthy data structures are often the root causes of memory footprint problems.

2.1 Distinguishing Data from Overhead

The size of a data structure is the number of bytes it occupies. The *health* of a data structure tells you what these bytes are accomplishing. Is the memory being spent on something important? A very useful way to classify bytes is as either data or overhead. Bytes classified as data represent the real data intrinsic to your program. All other bytes are overhead of the particular representation you have chosen for that data. A healthy data structure, from a space standpoint, is one with a low overhead. Sometimes there are tradeoffs; a larger design may be necessary for performance. Sometimes, though, choosing a design with lower memory overhead actually improves performance.

In Java, it is all too easy to design a data structure that is bloated by a factor of more than 50%; this means that over half of the data structure is devoted to things unrelated to the data that the application cares about. To understand why there is so much overhead, it is useful to dig deeper into its different sources:

- *JVM overhead.* Each object has a header imposed by the JVM to facilitate various administrative tasks, such as garbage collection. In addition, the JVM may impose additional object alignment overhead.

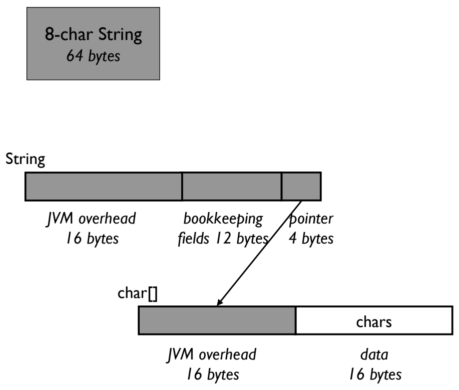


Figure 2.1. An eight character string in Java 6.

- *Pointers.* Pointers glue objects together into data structures. Pointers can be null or non-null.
- *Bookkeeping fields.* Not all primitive fields store real application data. Some are bookkeeping fields needed by the representation.

The Memory Bloat Factor

An important metric for evaluating the health of a data model design is the fraction of overhead in the objects of that data model. The *memory bloat factor*, or simply *bloat factor*, is the ratio of overhead to total size.

Example: An 8-Character String You learned in the quiz in Chapter 2 that an 8-character string occupies 64 bytes, which seems surprisingly high. The actual character data takes up 16 bytes, since Java supports the international character set, which uses 2-bytes per character. So it costs 16 bytes to represent the eight characters themselves. The other 48 bytes are pure overhead. This structure has a *bloat factor* of 75%. The actual data occupies only 25%. These numbers vary from one JVM to another, but the overhead is always very high. (Unless otherwise noted, all measurements in this book are from the 32-bit IBM Java 6 JVM.)

Figure 2.1 shows why the bloat factor is so high. Strings have all three kinds of overhead – JVM overhead, pointers, and bookkeeping fields. Every Java string is really two objects: a `String` that serves as a wrapper, and a `char` array with the actual characters. Two objects means two object headers, plus the pointer glueing the two objects together. The `String` object is entirely overhead, containing no real data. Part of its cost is three bookkeeping fields: a length and an offset, which have meaning only when this string is the result of a substring operation, and a saved hashcode. Adding all of this together, the overhead is 48 bytes. If you were to design a string from scratch, 20% overhead might seem a reasonable target. For such a common data type, it is important to obtain a low overhead. Unfortunately, for a string to have only 20% overhead with its current implementation, it would need to be 96 characters long. As bad as this seems, this string representation does at least have the benefit that it amortizes away its overhead. The overhead cost is always 48 bytes, so overhead of a string approaches 0% for larger and larger strings. For some data structures it is not possible to amortize away overhead costs, as discussed in Section 2.4.

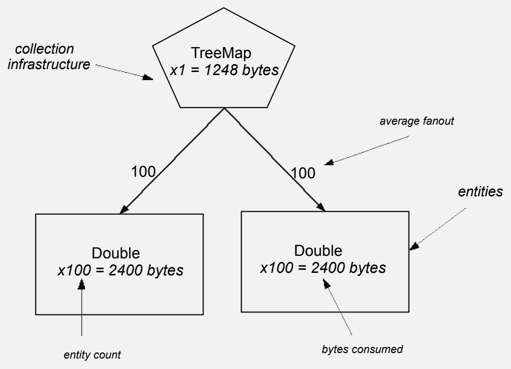
Strings, in particular short strings, are pervasive. It is important to be aware of the overhead of strings when incorporating them into your data design. Given the overhead of Java strings, the choices you make in Java need to be different than in a language like C, where the string overhead is minimal. Making informed choices is critical for the overall memory health of an application. There will be more on strings in later chapters.

2.2 Entities and Collections

A string is an example of a very simple data structure. When modeling a more complex data structure, you design classes to represent application entities, and choose collections in which to store them. Seemingly small choices you make in both cases can have a big impact in memory bloat. For example, the choice of the initial capacity of a collection can make or break the scalability of your design.

To help understand the health of more complex data structures, it is useful to have a diagram notation that spells out the impacts of various choices. An Entity-Collection (EC) diagram is a cousin of an Entity-Relation (ER) diagram, that exposes just the right level of implementation detail to help calculate the memory bloat factor of a complex data structure. As its name implies, an EC diagram shows the entities and collections of a design. It is important to remember that the entities and collections in the diagram are abstracted from the actual data structure objects. A diagram box may be hiding other objects that help implement the entity or collection. For example, a diagram box for a `String` entity represents both the `String` object and its underlying character array.

The Entity-Collection (EC) Diagram



In an EC diagram, there are two types of boxes, pentagons and rectangles. Each box summarizes the implementation details of a portion of a data structure. Pentagons represent collection infrastructure, and rectangles represent the entities of your application.

An EC diagram represents the *containment* structure of your data model. So, an edge from a collection to a rectangular entity indicates membership of the entity in the collection. This also means that, if a certain type of collection or entity is used in multiple ways in the same data structure, then this type will appear in multiple locations — one in each location in the data structure in which that type is used.

The notation $xN = M$ inside each node means there are N objects of that type in that location in the data structure, and in total these objects occupy M bytes of memory. Each edge in a content schematic is labeled with the average fanout from the source entity to the target entity.

Where an ER or UML diagram would show the attributes of each entity, but ignore the implementation costs, an EC diagram summarizes the total cost in the single sizing number shown in each node. Where these other diagrams would show relations or roles as edges, an EC diagram shows a node summarizing the collections implementing this relation.

Example: A Monitoring System A monitoring program collects samples from a distributed system. Each sample it collects consists of a unique timestamp and a value, each represented as a double. The samples arrive out of order. The task

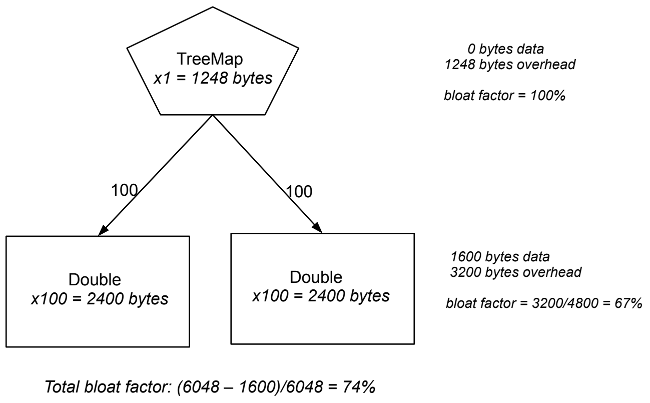


Figure 2.2. EC Diagram for 100 samples stored in a `TreeMap`

is to display samples in chronological order, after all of the data has been collected. The solution requires a data structure to store the samples.

A map is a convenient way to store these timestamp-value pairs. A regular `HashMap` only solves part of the monitoring system problem. To be able to display the samples in chronological order, the entries have to be sorted by timestamp. The first idea that leaps forward is to use a `TreeMap`. A `TreeMap` is a map that maintains its entries in sorted order, so this appears to be a perfect choice. The only problem is the memory cost.

An EC diagram of a `TreeMap` storing 100 samples is shown in Figure 2.2. This diagram gives a schematic view of the `TreeMap` and the entities it contains, along with their costs. The total cost, obtained by adding up the entity and collection costs, is 6,048 bytes. The real data consumes only 1,600 bytes of this, since a double occupies 8 bytes and there are 200 of them. Therefore, the bloat factor is 74%.

Looking at the individual parts of the EC diagram can provide some insight into the source of this overhead. First, the sample timestamps and values are stored as `Double` objects. This is because the standard Java collection APIs take only `Objects` as parameters, not scalars, forcing you to box your scalars before putting them in a collection. Even with Java's autoboxing, the compiler still generates a boxed scalar in order to store a value in a standard collection. A single instance of a `Double` is 24 bytes, so 200 `Doubles` occupy 4,800 bytes. Since the data is only 1,600 bytes, 33% of the `Double` objects is actual data, and 67% is overhead. This is a high price for a basic data type.

The `TreeMap` infrastructure occupies an additional 1,248 bytes of memory. All of this is overhead. What is taking up so much space? `TreeMap`, like every other collection in Java, has a wrapper object, the `TreeMap` object itself, along with other internal objects that implement the collection functionality. Each type of collection in the standard library has a different kind of infrastructure, some are built from arrays, some from separate entry objects linked together, and some use a combination of both. Internally, `TreeMap` is a self-balancing search tree. The tree nodes maintain pointers to parents and siblings. In newer releases of Java 6, each node in the tree can store up to 64 key-value pairs in two arrays. This example uses this newer implementation, which is more memory-efficient for this case, but still expensive.

Using a `TreeMap` is not *a priori* a bad design. It depends on whether the overhead is buying something useful. `TreeMap` has a high memory cost because it maintains its entries in sorted order while they are being randomly inserted and deleted. It constantly maintains a sorted order. If you need this capability, for instance, if you have a real time monitor that is constantly being updated, and you need to look at the data in order, then `TreeMap` is a great data structure. But if you do not need this capability, there are other data structures that could provide sorted-map functionality with less memory consumption. In this example, the data needs to be sorted only after data collection is complete. There is an initial load phase followed by a use phase. The sorted order is only needed during the second phase, after all the data is loaded. This load-and-use behavior is a common pattern, and it can be exploited to choose a more memory-efficient representation.

Of course, another nice aspect of `TreeMap` is that it can be pulled off the shelf. It would be ideal if there were another collection that provides just the needed functionality. If not, maybe there is a way to easily extend another collection class by adding a thin wrapper around it. Writing your own collection class from scratch should rarely be necessary, and is not recommended.

2.3 Two Memory-Efficient Designs

This section describes two other data structure designs to store the monitoring system samples. These designs use less memory and do not maintain sorted order while the data is being loaded. This is not a problem, since the data can be sorted after loading.

The first design stores the samples in an `ArrayList`, where each entry is a `Sample` object containing a timestamp and value. Both values are stored in primitive `double` fields of `Sample`. There is a bit more code that has to be written, but it is not excessive. Fortunately, the standard Java `Collections` class has some useful static methods so that new sort and search algorithms do not have to be implemented. The `sort` and `binarySearch` methods from `Collections` each can take an `ArrayList` and a `Comparable` object as parameters. To take advantage of these methods, the new Sample class has to implement the Comparable interface, so that two sample timestamps can be compared:

```
public class Sample implements Comparable<Sample> {

    private final double timestamp;
    private final double value;

    public Sample(double timestamp, double value) {
        this.timestamp = timestamp;
        this.value = value;
    }

    public double getTimestamp() {
        return timestamp;
    }

    public double getValue() {
        return value;
    }

    public int compareTo(Sample that) {
        return Double.compare(this.getTimestamp(), that.getTimestamp());
    }
}
```

Additionally, the ArrayList needs to be stored in a wrapper class that implements map operations. Here are two methods, getValue and sort, of a wrapper class Samples.

```
public class Samples {

   public final static double NOT_FOUND = -1d;
    private ArrayList<Sample> samples = new ArrayList<Sample>();
    ....

    public double getValue(double timestamp) {
        Sample sample = new Sample(timestamp, 0.0);
        int result = Collections.binarySearch(samples, sample);
        if (result < 0) {
         return NOT_FOUND;
```

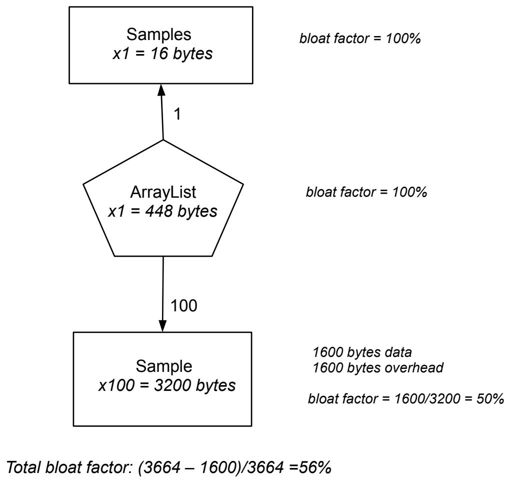


Figure 2.3. EC Diagram for 100 samples stored in an `ArrayList` of `Samples`

```
        }
        return samples.get(result).getValue();
    }

    public void sort() {
        Collections.sort(samples);
        samples.trimToSize();
    }
}
```

The EC diagram for 100 entries is shown in Figure 2.3. This design uses less memory and has better health than the `TreeMap` design. The memory cost is reduced from 6,048 to 3,664 bytes, and the overhead is reduced from 74% to 56%. There are two reasons for this improvement. First, the timestamps and values are not boxed. They are stored as primitive double fields in each `Sample`. This reduces the number of objects needed to store the actual data from 200 to 100. Fewer objects means fewer object headers and pointers, which is significant in this case. Secondly, an `ArrayList` has lower infrastructure cost than a `TreeMap`. `ArrayList` is one of the most memory-efficient Java collections. It is implemented using a wrapper object and an array of pointers. This is much more compact than the heavy-weight `TreeMap`.

While this is a big improvement, 56% overhead still seems high. Over half the memory is being wasted. How hard is it to get rid of this overhead completely?

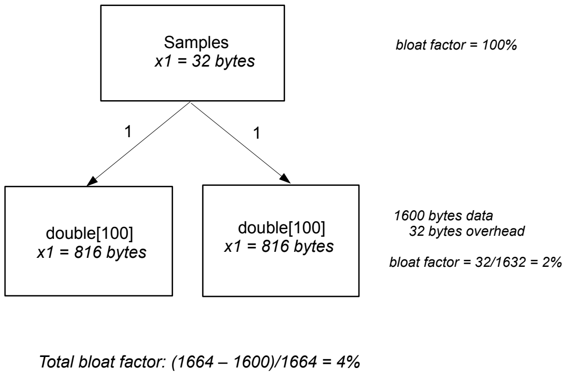


Figure 2.4. EC Diagram for 100 samples stored in two parallel arrays

Eliminating overhead means eliminating objects altogether. This is not a recommended practice, unless memory is extremely tight, and there is no other choice. It is none the less an interesting exercise to compare a best case design with the other designs. The most space-efficient design uses two parallel arrays of doubles. One array stores all of the sample timestamps, and the second stores all of the values. These arrays can be stored in a wrapper class that implements a map interface.

This design requires a bit more code, but, again, it is not excessive. Like the `Collections` class, the `Arrays` class provides static `sort` and `binarySearch` methods. However, these methods apply only to a single array. If you sort the timestamp array, you will lose the association between the timestamps and their corresponding values. You can use an extra temporary array to get around this problem. The implementation is left as an exercise. The EC diagram for this design using two double arrays is shown in Figure 2.4. The overhead is only 4%.

For the three designs presented, there is a tradeoff between ease of programming and memory efficiency. More programming is needed as the design becomes more memory efficient. However, in many cases, memory efficient designs are just as easy to implement as inefficient designs. When there are two distinct execution phases, as in this example, the data structure can be trimmed after the first phase, to eliminate empty space reserved for growth. Another option is to use one data structure in the first phase, and copy it into a second data structure for the second phase. This approach can sometimes be a good compromise between ease-of-programming and memory efficiency.

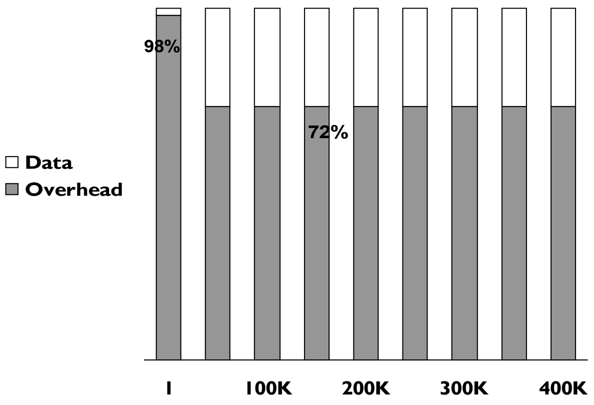


Figure 2.5. Health Measure for the `TreeMap` Design Shows Poor Scalability

2.4 Scalability

The evaluation of the three monitoring system designs was based on only 100 samples. In a real scenario, a monitoring system might have to handle hundreds of thousands, or millions, of samples. Stress testing is usually performed late in a development cycle, when it is very costly to fix a memory footprint problem. Using a memory health approach, it possible to predict how well a data structure design will scale much earlier.

The basic question for predicting scalability is what happens to the bloat factor as a data structure grows. The `TreeMap` design has 74% overhead with 100 samples. With 100,000 samples, maybe the high overhead will be amortized away, that is, maybe this design will scale well, even if it is inefficient for small data sizes. The bar graph in Figure 2.5 shows how the `TreeMap` design scales as the number of samples increase. Each bar is split into two parts: the percentage of overhead and the percentage of data. For a single sample, the overhead is 98%! As more samples are added, the bloat factor drops to 72%. Unfortunately, with 200,000 samples, and 300,000 samples, the bloat factor is still 72%. The `TreeMap` design is not only bloated, but it also does not scale. It is constantly bloated.

To understand why this happens, recall that the infrastructure of `TreeMap` is made up of nodes, with two 64-element arrays hanging off of each node. As samples are added, the infrastructure grows, since new nodes and arrays are being created. Also, each additional sample has its own overhead, namely the JVM overhead in each `Double` object. When the `TreeMap` becomes large enough, the *per-entry overhead* dominates and hovers around 72%. The bloat factor is larger when the `TreeMap`

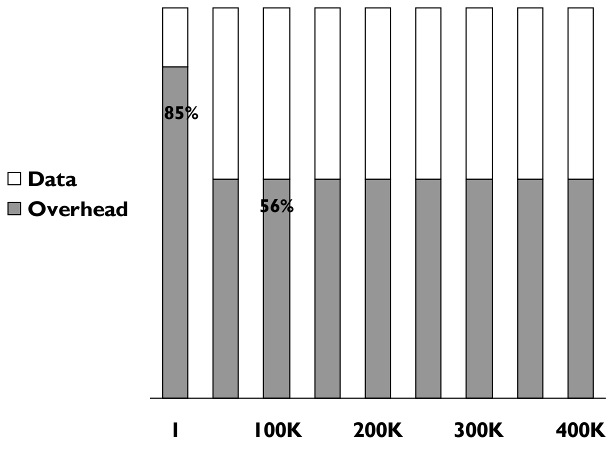


Figure 2.6. Health Measure for the `ArrayList` Design

is small. In contrast, for small `TreeMaps`, the fixed cost of the initial `TreeMap` infrastructure is relatively big. The `TreeMap` wrapper object alone is 48 bytes. This initial fixed cost is quickly amortized away as samples are added.

Fixed vs Per-Entry Overhead

The memory overhead of a collection can be classified as either *fixed* or *per-entry*. Fixed overhead stays the same, no matter how many entries are stored in the collection. Small collections with a large fixed overhead have a high memory bloat factor, but the fixed overhead is amortized away as the collection grows. Per-entry overhead depends on the number of entries stored in the collection. Collections with a large per-entry overhead do not scale well, since per-entry costs cannot be amortized away as the collection grows.

Figure 2.6 shows similar data for the `ArrayList` design. Like the `TreeMap` design, there is a fixed overhead, which is significant when the `ArrayList` is small. As the `ArrayList` grows, the fixed overhead is amortized away, but there is still a per-entry cost of 56%, that remains constant.

For the last design that uses arrays, there is only fixed overhead, namely, the `Samples` object and JVM overhead for the arrays. There is no per-entry overhead at all. Figure 2.7 shows the initial 80% fixed overhead is quickly amortized away. When more samples are added, the bloat factor becomes 0. The samples themselves are pure data.

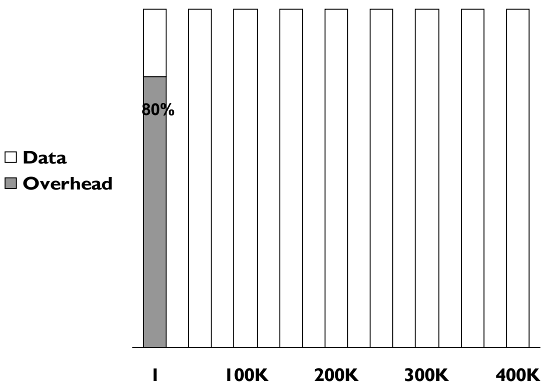


Figure 2.7. Health Measure for the Array-Based Design Shows Perfect Scalability

This analysis helps predict the memory requirements for large sample sizes. Suppose the monitoring system needs to process 10 million samples. The data alone takes up 160 million bytes (153MB). The graphs shown in this section can be used to quickly calculate how much memory you will need. For the `TreeMap` design, the overhead cost is 72%, so you will need 546MB to store the samples. For the `ArrayList` design, you will need 347MB. For the `array` design, you will need only 153MB. As these numbers show, the design choice can make a huge difference.

2.5 Summary

When you are developing a large application, closing your eyes to quantitative considerations and thinking everything is going to be fine is risky. Taking the time to measure the cost and health of your data design can pinpoint problems early on. This chapter described three conceptual tools to help evaluate the memory-efficiency of a design.

- The *memory bloat factor* measures how much of your the memory usage of your design is real data, and how much is an artifact of the way you have chosen to represent it. This metric tells you how much room there is for improvement, and if in fact you are paying a reasonable price for what you are getting.
- Complex data structures are built up from data entities and collections of these entities. The *Entity-Collection (EC) Diagram* shows the costs associated with the different parts of a complex data structure. These diagrams help you to compare the memory efficiency of different representation choices.
- By classifying the overhead of a collection as either *fixed* or *per-entry*, you can predict how much memory you will need to store very large collections. Being able to predict scalability is critical to meeting the requirements of larges applications.

These are the basic tools used in the rest of this book. To estimate the cost of an entity, you will need to know how many bytes each primitive type needs, what is the pointer size, and what the JVM overhead is. You will learn how to estimate the cost of entities in Chapter 4. To estimate scalability, you will need to know what the fixed and per-entry costs are for the collection classes you are using. These are given in Chapter 7.

Chapter 3

DELEGATION

An important design decision when modeling data is how many Java classes to use to represent a logical concept. The choices made at this design stage can impact memory costs significantly. At one extreme, a single class may store all attributes. At the other extreme, a main class may delegate many of its attributes to other classes, resulting in a fine-grained design. Delegation is a very popular pattern because of the flexibility it provides. It is also one of the few ways Java allows you to reuse existing classes. Yet, overly fine-grained data models can result in poor memory health. This chapter explains how to evaluate the granularity of a design from a memory perspective. It begins with the costs of basic objects, and moves on to examples from real applications.

3.1 The Cost of Objects

There is no Java library method that returns the size of an object. This is by design. Unlike systems languages like C, a Java programmer is not supposed to know either the size of an object or how it is laid out in memory. The JRE, not the programmer, manages storage, and the JRE has the freedom to implement its own storage policy. However, knowing the sizes of objects is necessary for understanding memory requirements and scalability. Fortunately, it is not hard to estimate the size of an object, and a good estimate of the most prevelant classes is usually sufficient for making intelligent design choices. You need to know just a few basics, starting with the number of bytes needed to store primitive data types. These sizes are defined in the Java language specification [**?**], and are given in Table 3.1.

Primitive data type	Number of bytes
boolean, byte	1
char, short	2
int, float	4
long, double	8

Table 3.1. The number of bytes needed to store primitive data.

	Sun Java 6 (u14)	IBM Java 6 (SR4)
Object Header size	8 bytes	12 bytes
Array Header size	12 bytes	16 bytes
Object alignment	8 byte boundary	8 byte boundary

Table 3.2. Object overhead used by the Sun and IBM JREs for 32-bit architectures.

		Sun Java 6 (u14)			IBM Java 6 (SR4)		
Class	Data size	Header	Alignment fill	Total bytes	Header	Alignment fill	Total bytes
Boolean, Byte	1	8	7	16	12	3	16
Character, Short	2	8	6	16	12	1	16
Integer, Float	4	8	4	16	12	0	16
Long, Double	8	8	0	16	12	4	24

Table 3.3. The sizes of boxed scalar objects, in bytes.

Objects are bigger than the sum of their fields, and their size depends on the specific JRE implementation. The JRE allocates a header with each object that stores information such as the object's class, an identity hashcode, a monitor used for locking, and various flags. For array objects, the header has an additional integer to store the number of array elements. Additionally, the JRE requires objects to be aligned on specific address boundaries, for example, addresses that are multiples of 8. To show how implementations differ, Table 3.2 gives object header and alignment costs imposed by two JREs, SUN Java 6 (update 14) and IBM Java 6 (SR4), both for 32-bit architectures.

The simplest objects are the boxed scalars, which are objects with a single primitive data type field. Since both of these JREs align objects on 8-byte boundaries and object headers are at least 8 bytes, a boxed scalar is at least 16 bytes. Table 3.3 gives the sizes of boxed scalars.

Estimating the size of an object with more than one field is a bit more complicated, since the hardware imposes additional alignment requirements on specific data types. For example, integer fields are usually aligned on 4 byte boundaries. This field alignment potentially introduces more padding.

For example, consider a class `SimpleEmployee`, which has all primitive fields. The comments show the number of bytes needed to store the primitive data.

```
class SimpleEmployee {
    int hoursPerWeek;   // 4 bytes
    boolean exempt;     // 1 byte
    double salary;      // 8 bytes
    char jobCode;       // 2 bytes
    int yearsOfService; // 4 bytes
}
```

If the fields are laid out one after the other with no fill, then the field `salary` would have to begin on a 5-byte boundary, which is not allowed for `doubles`. To avoid adding 3 more bytes of padding, the JRE could rearrange the fields to take up the smallest amount of space. In fact, the Sun JRE does a good job rearranging and packing fields, and the IBM JRE does not. This means that field sizes depend on the specific JRE also.

Using the Sun JRE, you can assume each field is the size of its primitive data type because the fields are packed. The size of a `SimpleEmployee` object is 32 bytes:

```
8 + (4+1+8+2+4) = 27 bytes, rounds up to 32 bytes
```

In contrast, using the IBM JRE, all fields in non-array objects are either 4 or 8 bytes. Here is the `SimpleEmployee` class with field sizes adjusted accordingly.

```
class SimpleEmployee {
    int hoursPerWeek;      // 4 bytes
    boolean exempt;        // 4 byte
    double salary;         // 8 bytes
    char jobCode;          // 4 bytes
    int yearsOfService;    // 4 bytes
}
```

The size of a `SimpleEmployee` is 40 bytes:

```
12 + (4+4+8+4+4) = 36, rounds up to 40 bytes
```

For arrays, the data is always packed with no spaces. For example, here is a formula that estimates the size of an array of 100 `chars`:

```
header + 100*2, round up to a multiple of the alignment
```

Estimating Object Sizes

The size of an object can be estimated as follows:

1. Add together the sizes of all of the fields and the object header. Fields include those in the object's class and all of its superclasses.

2. Round up the result to the next multiple of the object alignment.

The object header size and alignment depend on the JRE. Field sizes also depend on the JRE. For example, the JRE may pack fields to obtain the minimum possible size, or align fields on word boundaries. Array data is always packed.

For objects with only a small amount of primitive data, the overhead is relatively high. But as the amount of primitive data increases, the bloat factor decreases. The overhead cost consists of a fixed object header cost and a bounded alignment cost, which are amortized when the object is big. For example, for the Sun JRE, the bloat factors for the boxed scalars range from 94% for a `Boolean` down to 50% for a `Double`. The bloat factor for a `SimpleEmployee` is 46%. The bloat factor for an array of 100 `chars`s is insignificant. This is not the case for objects with other kinds of overhead, like references.

3.2 The Cost of Delegation

The `SimpleEmployee` class is not very realistic, since it has only primitive fields. Usually, a class has some fields that are objects. In Java, a field of type `ObjectType` is implemented using *delegation*, that is, the field stores a reference to another object of type `ObjectType`.

Here is a more realistic employee class with several reference fields. An employee now has a name, which is a `String`, and a start date, which is a `Date`. The type of `salary` has been changed from `double` to `BigDecimal`. `BigDecimal` avoids potential roundoff errors.

```
class EmployeeWithDelegation {
    int hoursPerWeek;         // 4 bytes
    String name;              // 4 bytes
    BigDecimal salary;        // 4 bytes
    Date startDate;           // 4 bytes
    boolean exempt;           // 1 byte
    char jobCode;             // 2 bytes
    int yearsOfService;       // 4 bytes
}
```

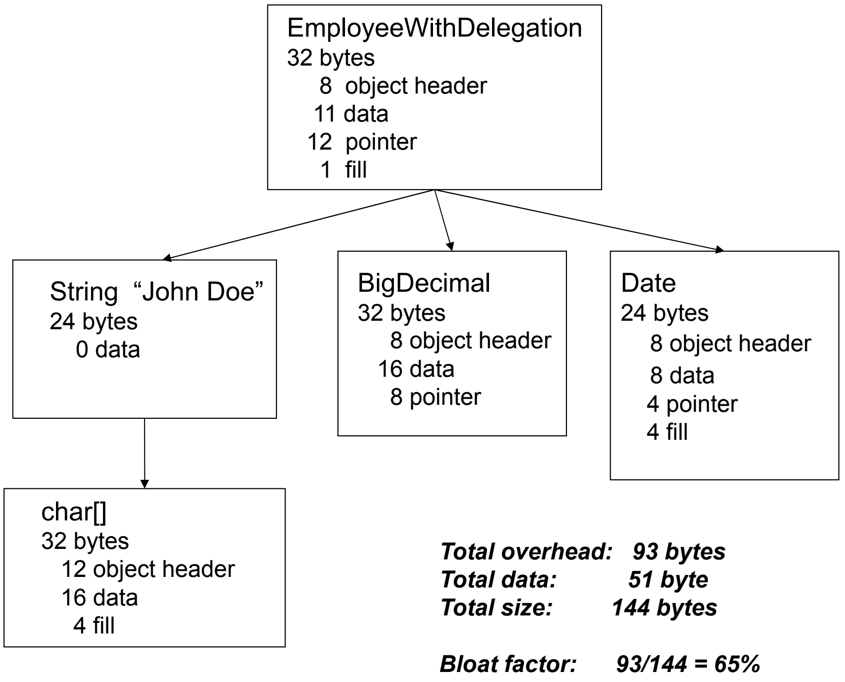


Figure 3.1. The memory layout for an employee "John Doe".

On 32-bit architectures, a pointer is 4 bytes. You can calculate the size of any object as described in in Section 3.1, plugging in 4 bytes for each reference field. Assuming the Sun JRE, the size of an EmployeeWithDelegation object is 32 bytes:

```
(4+4+4+4+1+2+4) + 8 = 31, rounds up to 32 bytes
```

While an instance of the `EmployeeWithDelegation` class is a single 32 byte object, an entire employee, including name, salary, and start date consists of five objects. Two of these objects are used to store the name. (Recall from Section 2.1 that a string is represented by a wrapper `String` object and a `char` array.) The memory layout for a specific employee "John Doe" is shown in Figure 3.1.

A comparion of a `EmployeeWithDelegation` object with a `SimpleEmployee` object from Section 3.1 shows that the size has increased from 32 to 144 bytes, and the bloat factor has increased from 46% to 65%. There is more information stored in the new version, so it is no surprise that it is bigger. The increase in the bloat factor is more significant. Delegation increases memory bloat. Delegation introduces additional object headers, a pointer for each delegated object, and empty pointer slots for uninitialized object fields. Delegation may also force additional alignment costs, since each new delegated object has to be aligned to an 8-byte boundary.

In the spirit of keeping things simple, Java does not allow you to nest objects inside other objects, to build a single object out of other objects. You cannot nest an array inside an object, and you cannot store objects directly in an array. You can only point to other objects. Even the basic data type `String` consists of two objects. This means that delegation is pervasive in Java programs, and it is difficult to avoid a high level of delegation overhead. Single inheritance is the only language feature that can be used instead of delegation to compose two object, but single inheritance has limited flexibility. In contrast, C++ has many different ways to compose objects. C++ has single and multiple inheritance, union types, and variation. C++ allows you to have `struct` fields, you can put arrays inside of structs, and you can also have an array of structs.

Because of the design of Java, there is a basic delegation cost that is hard to eliminate it. This is the cost of object-oriented programming in Java. While it is hard to avoid this basic delegation cost, it is important not to make things a lot worse, as discussed in the next section.

3.3 Fine-Grained Data Models

The software engineering culture tends to promote delegation, and for good reasons. Delegation provides a loose coupling of objects, making refactoring and reuse easier. Replacing inheritance by delegation is often recommended, especially if the base class has extra fields and methods that the subclass does not need. In languages with single inheritance, once you have used up your inheritance slot, it becomes hard to refactor your code. Therefore, delegation can be more flexible than inheritance for implementing polymorphism. However, overly fine-grained data models can be expensive both in execution time and memory space.

There is no simple rule that can always be applied to decide when to use delegation. Each situation has to be evaluated in context, and there may be tradeoffs among different goals. To make an informed decision, it is important to know what the costs are.

Suppose an emergency contact is needed for each employee. An emergency contact is a person along with a preferred method to reach her. The preferred method can be email, cell phone, work phone, or home phone. All contact information for the emergency contact person must be stored, just in case the preferred method does not work in an actual emergency. Here are class definitions for an emergency contact, written in a highly delegated style that is not uncommon in real applications.

```
class EmployeeWithEmergencyContact {
    ...
    EmergencyContact contact;
}

class EmergencyContact {
    ContactPerson contact;
    ContactMethod preferredContact;
}

class ContactPerson {
   String name;
   String relation;
   EmailAddress email;
   PhoneNumber phone;
   PhoneNumber cell;
   PhoneNumber work;
}

class ContactMethod {
   ContactPerson owner;
}

class PhoneNumber extends ContactMethod {
   byte[] phone;
}

class EmailAddress extends ContactMethod {
   String address;
}
```

The memory layout for a sample employee is shown in Figure 3.2. There are 15 objects used to store emergency contact information, with a bloat factor of 77%, which seems excessive. The objects are all small, containing only one or two meaningful fields, which is a symptom of an overly fine grained data model. Also, all of the data is at the bottom of object chains, 4 to 5 objects long. Refactoring can flatten this structure somewhat, undoing a few of the delegation.

One object that looks superfluous is `EmergencyContact`, which encapsulates the contact person and the preferred contact method. Reversing this delegation involves inlining the fields of the `EmergencyContact` class into other classes, and eliminating the `EmergencyContact` class. Here are the refactored classes:

```
class EmployeeWithEmergencyContact {
   ...
   ContactPerson contact;
}

class ContactPerson {
   String name;
   String relation;
   EmailAddress email;
```

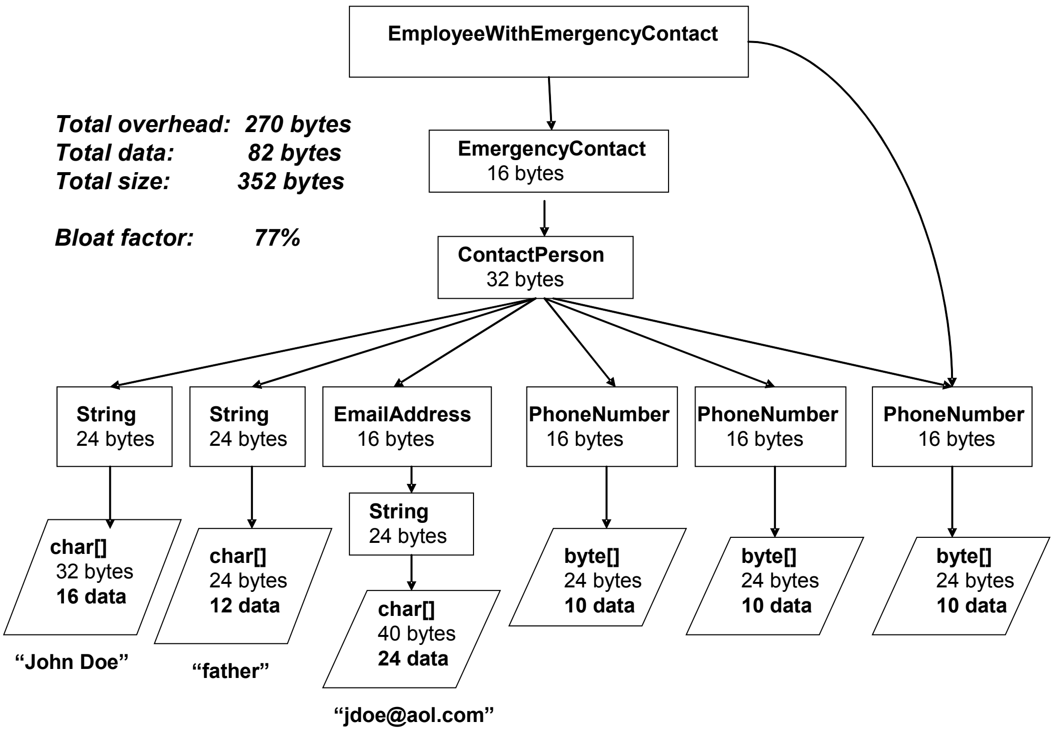


Figure 3.2. The memory layout for an employee with an emergency contact.

```
        PhoneNumber phone;
        PhoneNumber cell;
        PhoneNumber work;
        ContactMethod preferredContact;
    }
```

This change eliminates an object from Figure 3.2, but it only recovers 8 bytes, since a 16 byte object is removed and `ContactPerson` is 8 bytes bigger. You can save considerably more space by inlining the four `ContactMethod` classes into the `ContactPerson` class, which removes 64 bytes of overhead. In a system with many instances of employees stored in memory, this reduction is significant. In order to make this change, the preferred contact method must be encoded somehow in `ContactPerson`. A simple way to achieve this is to use an enumeration type field, which has the same size as a reference field, to discriminate among the different contact methods:

```
    enum PreferredContactMethod {
        EMAIL, HOME_PHONE, CELL_PHONE, WORK_PHONE;
    }

    class ContactPerson {
        PreferredContactMethod preferred;
        String name;
        String relation;
        String email;
        byte[] cellPhone;
        byte[] homePhone;
        byte[] workPhone;
    }
```

Figure 3.3 shows the memory layout after these changes. Both the size and the bloat factor have been reduced.

When many objects are used to represent one logical concept, this is an indication that the data model may be using too much delegation. Delegation is good, but it is possible to overuse a good thing. A design with fewer, bigger objects has less overhead and is more scalable. Whenever you use delegation, there should be a good reason. This is especially true for the important data entities in an application, those that will determine the scalability of the program.

3.4 Large Base Classes

As discussion in the last section, highly-delegated data models can result in too many small objects. Occasionally, you run across a highly-delegated data model where the delegated objects are large. This can happen when delegated classes inherit from a

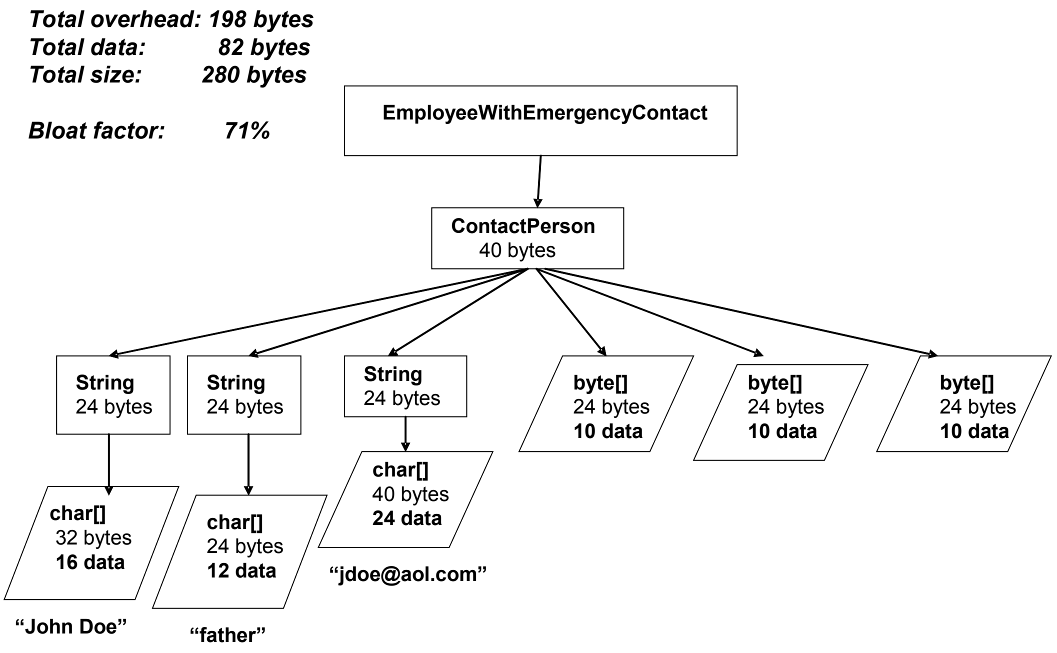


Figure 3.3. Memory layout for refactored emergency contact.

large base class. When fine grained data modeling is combined with inheriting from large base classes, memory costs multiply and can become prohibitive.

A frequent data management requirement is to track creation and update information, that is, when data is created or updated and by whom. Here is a base class, taken from a real application, that stores create and update information.

```
class UpdateInfo {
    Date createDate;
    Party enteredBy;
    Date updateDate;
    Party updateBy;
}
```

You can track changes by subclassing from `UpdateInfo`. Update tracking is a *cross-cutting feature*, since it can apply to any class in a data model.

Returning to `EmployeeWithEmergencyContact` in Section 3.3, suppose that updates to employee emergency contacts need to be tracked. You need to decide how precise the tracking should be. Should every update to every phone number and email address be tracked, or is it sufficient to track the fact that some contact information was changed for an emergency contact? If you decide to track changes to every contact phone number or email address, you can easily achieve this by extending the `ContactMethod` class defined in the fine-grained data model from Section 3.3:

```
class ContactMethod extends UpdateInfo {
    ContactPerson owner;
}
```

Figure 3.4 shows an instance of an contact person with update information associated with every `ContactMethod`. Not only is this a highly delegated structure with multiple `ContactMethod` objects, but each one has an additional 16 bytes. Furthermore, there are potentially four more objects of type `Date` and `Party` for each of the four `ContactMethod` objects. A far more scalable solution is to move up a level, and track changes to each `ContactPerson`. With this solution, you do not need such a fine-grained data model, and the update tracking functionality is 1/4 the cost.

The solution in Figure 3.4 provides a very fine granularity of functionality. An argument can be made in favor of this solution, since you lose functionality and flexibility if you only track updates to `ContactPerson`. However, if the program hits a scalabity problem, it may not be possible to be this casual with memory. Also, an alternate design may be available that gives the desired functionality in a more memory-efficient way. In this example, you could implement an update log instead of tracking updates in the objects themselves. Assuming updates are sparse, this is a much better solution. It is very easy define a subclass without looking closely at the memory size of a superclass, especially if the inheritance chain is long.

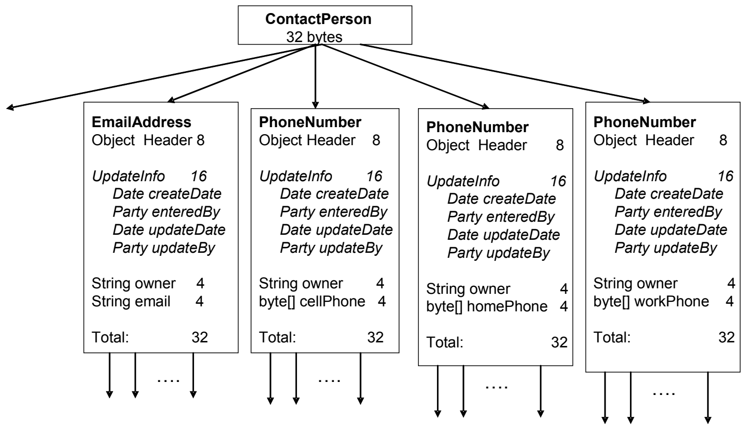


Figure 3.4. The cost of associating `UndateInfo` with every `ContactMethod`.

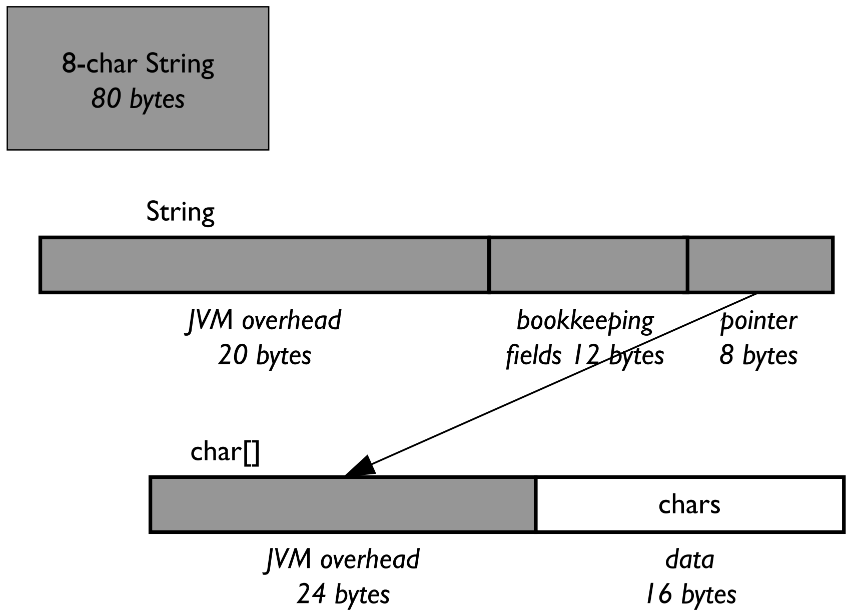


Figure 3.5. The memory layout for an 8 character string by a 64-bit JRE.

3.5 64-bit Architectures

If your application does not fit into memory, perhaps moving to a 64-bit architecture will save you. However, to support a 64-bit address space, more memory is required. Object header sizes double, and pointers are 8 bytes instead of 4. Some studies [?] show that memory consumption can increase by 40%-50% going from a 32-bit to a 64-bit address space for the same Java program.

Consider what happens to the 8-character string from Section 2.1 in a 64-bit JRE. The memory layout is shown in Figure 3.5. The 64-bit string is 50% bigger than the 32-bit string. All of the additional cost is overhead.

In reality, things are not so bad. Both the Sun and the IBM JREs have implemented a scheme for compressing addresses that avoids this code size blowup, provided that the heap is less than 32 gigabytes. Address compression is implemented by stealing a few unused bits from 32-bit pointers. During execution, 32-bit compressed addresses are converted into 64-bit native addresses, and vice-versa.

Address compression is available in the Sun Java 6 (update 14) release, enabled with the option -XX:+UseCompressedOops. It is available in IBM Java 6 J9 SR 4 with the option -Xcompressedrefs.

3.6 Summary

The decision to delegate functionality to another object sometimes involves making a tradeoff between flexibility and memory cost. You need to decide how much flexibility is really needed, and you also need to be aware of the actual memory costs. This chapter provides the basic knowledge for estimating memory costs.

- An object size depends on the object header size, field alignment, object alignment, and pointer size. These can vary, depending on the JRE and the hardware. The size of an object is the sum of the header and the field sizes, rounded up to an alignment boundary.

If you need the exact size of objects, there are various tools available. A list of resources is provided in the Appendix.

This chapter also describes several costly anti-patterns to avoid.

- A *highly-delegated data model* results in too many small objects and a large bloat factor. Typically, each object has only a few fields, which is excessive data granularity.
- A *highly-delegated data model with large base classes* results in too many big objects. Often, the data model is providing a fine granularity of function, which may no be needed.

Both the design of Java and software engineering best practices encourage highly delegated data models with many objects. This cost is often considered to be insignificant — delegating to another object is just a single level of indirection. But the costs of the pointers and object headers needed to implement delegation indirection add up quickly, and contribute significantly to large bloat factors in real applications.

Chapter 4

REDUCING OBJECT BLOAT

In many applications, the heap is filled mostly with instances of just a few important classes. You can increase scalability significantly by making these objects as compact as possible. This chapter describes field usage patterns that can be easily optimized for space, for example, fields that are rarely needed, constant fields, and dependent fields. Simple refactoring of these kinds of fields can sometimes result in big wins.

4.1 Rarely Used Fields

Chapter 3 presents examples where delegating fields to another class increases memory cost. However, sometimes delegation can actually save memory, if you don't have to allocate the delegated object all the time.

As an example, consider an on-line store with millions of products. Most of the products are supplied by the parent company, but sometimes the store sells products from another company:

```
class Product {
    String sku;
    String name;
    ..
    String alternateSupplierName;
    String alternateSupplierAddress;
    String alternateSupplierSku;
}
```

When there is no alternate supplier, the last three fields are never used. By moving these fields to a separate side class, you can save memory, provided the side object is allocated only when it is actually needed. This is called *lazy allocation.* Here are the refactored classes:

```
class Product {
    String sku;
    String name;
    ..
```

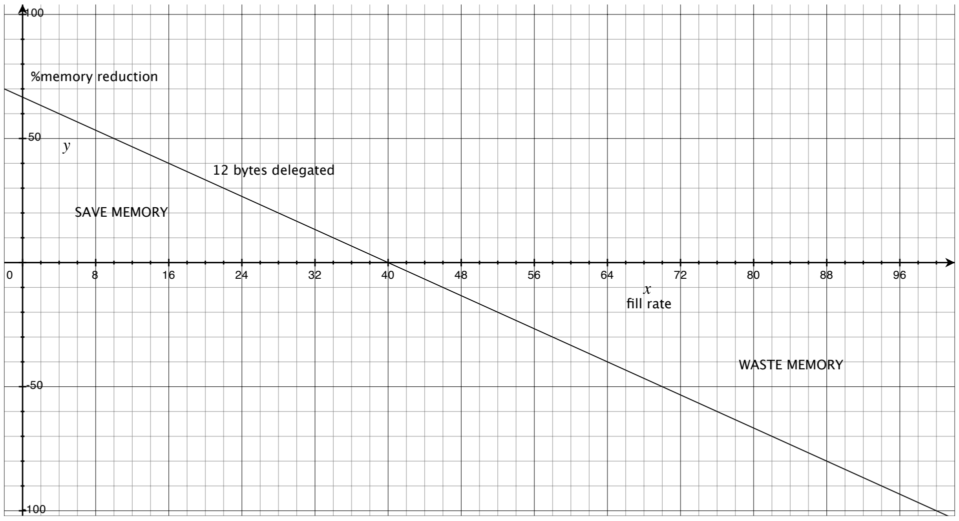


Figure 4.1. This plot shows how much memory is saved or wasted by delegating 12 bytes of memory to a side object. The x-axis is the fill rate, and the y-axis is the percent of memory saved.

```
    Supplier alternateSupplier;
}

class Supplier {
    String supplierName;
    String supplierAddress;
    String sku;
}
```

For products with no alternate supplier, eight bytes are saved per product, since three fields are replaced by one. Of course, products with an alternate supplier pay a delegation cost, including an extra pointer and object header. An interesting question is how much total memory is actually saved? The answer depends on the percentage of products that have an alternate supplier and need a side object, which we'll call the *fill rate*. The higher the fill rate, the less memory is saved. In fact, if the fill rate is too high, memory is wasted.

Figure 4.1 shows the memory saved for different fill rates, assuming three fields (12 bytes) are delegated. The most memory that can be saved is 66%, when the fill rate is 0%. When the fill rate is 10%, 50% of memory is saved. When the fill rate is over 40%, the memory saved is negative, that is, memory is wasted. The lesson here is that if you aren't sure what the fill rate is, then using delegation to save memory may end up backfiring.

In addition to the fill rate, the memory savings also depends on the number of fields delegated. The more bytes delegated, the larger the memory savings, assuming

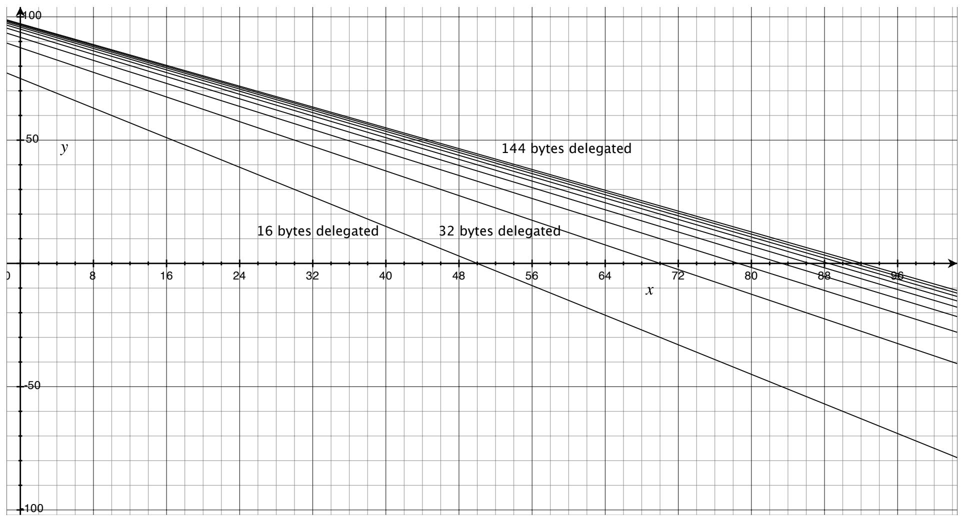


Figure 4.2. This plot shows how much memory is saved or wasted for different delegated field sizes. The x-axis is the fill rate, and the y-axis is the percent of memory saved. Each line represents a different delegated size, starting from 16 bytes, going up to 144 bytes by increments of 16 bytes.

the same fill rate. Figure 4.2 shows the memory saved for different fill rates and different delegated-field sizes. Each line represents a different delegated-field size. The bottom-most line represents a delegated field size of 16 bytes, the next line represents 32 bytes, the next represents 48 bytes, and so on, up to 144 bytes. As the delegated object size increases, you can worry less about the fill rate. For example, if 32 bytes are delegated, there is almost 90% savings with a low fill rate, and some memory savings with a fill rate up to 70%. As the delegation size increases, the lines start to converge, since the fixed delegation cost becomes relatively less important.

Delegation Cost Calculation

Assume the cost of a pointer is 4 bytes, and the cost of an object header is 8 bytes, and

$$\begin{aligned} B &= \textit{the size in bytes of the delegated fields} \\ F &= \textit{the fill rate} \end{aligned}$$

The memory cost per object with the delegated implementation is:

$$D = F(B+8)+4$$

Note that every object pays a 4 byte pointer cost, and only F of the objects pay for the side object. The proportional improvement is:

$$\begin{aligned} \frac{(B-D)}{B} &= 1-\frac{D}{B} \\ &= 1-\frac{(F(B+8)+4)}{B} \end{aligned}$$

The following equation can be used to obtain the plots in Figures 4.1 and 4.2, for different values of B:

$$y = 100 * (1-\frac{\frac{x}{100}(B+8)+4}{B})$$

(Note that this calculation does not take alignment fill into account, which can add additional delegation overhead.)

A common error is to delegate rarely-used fields to a side class, but forget to lazily allocate it, that is, always allocate a side object. In this case, instead of saving memory, you pay the full cost of delegation as well as the cost of unused fields. Lazy allocation can be error-prone, since it may require testing whether the object exists at every use. This complexity has to be weighed against potential memory savings.

4.2 Attribute Table

If a field is very rarely used, then it might make sense to delete it from its class altogether, and store it in a separate attribute table that maps objects to the attribute values. For example, suppose that a few of the products have won major awards, and you want to record this information. Rather than maintaining a field `majorAward` in every product, you can define a table:

```
class Product {
   static HashMap<String, String> majorAward = new HashMap
      ();
```

```
    ..
}
```

Even though a HashMap has it's own high overhead, this design will come out ahead if there are a small number of major awards. Whenever you add any kind of new table like this, however, you have to be careful you do not introduce a memory leak. If products are garbage collected, the corresponding entries in the attribute table should be cleaned up. This topic is discussed at length in ???.

4.3 Constant Fields

Declaring a constant field static is a simple way to save memory. Programmers usually remember to make constants like *pi* static. There are other situations that are a bit more subtle, for example, when a field is constant because of how it is used in the context of an application.

Returning to the product example, suppose that each product has a field catalog that points to a store catalog. If you know that there is always just one store catalog, then the field catalog can be turned into a static, saving 4 bytes per product.

As a more elaborate example, suppose that a Product has a field referencing a Category object, where a category may be books, music, clothes, toys, etc. Clearly, different products belong to different categories. However, suppose we define subclasses Book, Music, and Clothing of Product, and all instances of a subclass belong to the same category. Now the category field has the same value for products in each subclass, so it can be declared static:

```
class Book extends Product {
    static Category bookCategory;  // points to the book
        category object
    ..
}

class Music extends Product {
    static Category musicCategory; // points to the music
        category object
    ..
}

class Clothing extends Product {
    static Category clothingCategory; // points to the
        clothing category object
    ..
}
```

Knowing the context of how objects are created and used, and how they relate to other objects, is helpful in making these kinds of memory optimizations.

4.4 Mutually Exclusive Fields

Sometimes a class has fields that are never used at the same time, and therefore they can share the same space. Two mutually exclusive fields can be conflated into one field if they have the same type. Unfortunately, Java does not have anything like a union type to combine fields of different types. However, if it makes sense, mutually exclusive field types can be broadened to a common base type to allow this optimization.

For example, suppose that each women's clothing product has a size, and there are different kinds of sizes: xsmall-small-medium-large-xlarge, numeric sizes, petite sizes, and large women's sizes. One way to implement this is to introduce a field for each kind of size:

```
class WomensClothing extends Product {
   static Category clothingCategory; // points to the
      clothing category object
   ..
   SMLSize    smlSize;
   NumericSize numSize;
   PetiteSize  petiteSize;
   WomensSize  womensSize;
}
```

Each type is an enum class, such as:

```
enum SMLSize {
   XSMALL, SMALL, MEDIUM, LARGE, XLARGE;
}

enum NumericSize {
   ZERO, TWO, FOUR, SIX, EIGHT, TEN, TWELVE, FOURTEEN,
      SIXTEEN;
}

enum PetiteSize {
   ZERO, TWO, FOUR, SIX, EIGHT, TEN, TWELVE, FOURTEEN,
      SIXTEEN;
}

enum WomensSize {
   ONEX, TWOX, THREEX, FOURX;
}
```

These four size fields are mutually exclusive — a clothing item cannot have both a petite size and a women's size, for example. Therefore, you can replace these fields by one field, provided that the four enum types are combined into one enum type:

```
class Clothing extends Product {
    static Category clothingCategory; // points to the
        clothing category object
    ..
    ClothingSize   size;
}

enum ClothingSize {
    XSMALL, SMALL, MEDIUM, LARGE, XLARGE,
    ZERO, TWO, FOUR, SIX, EIGHT, TEN, TWELVE, FOURTEEN,
        SIXTEEN,
    PETITE_ZERO, PETITE_TWO, PETITE_FOUR, PETITE_SIX,
        PETITE_EIGHT, PETITE_TEN,
    PETITE_TWELVE, PETITE_FOURTEEN, PETITE_SIXTEEN,
    ONEX, TWOX, THREEX, FOURX;
}
```

If the types of two mutually exclusive fields are different classes, then you generalize these types by defining a common superclass, if possible. As a last resort, you can always combine these fields into a single field of type `Object`. Finally, if there are sets of fields that are mutually exclusive, then you can define a side class for each set of fields, where all side classes have a common superclass. In this case, you need to do the math to make sure that you actually save memory, given the extra cost of delegation.

4.5 Redundant Fields

A field is redundant if it can be computed on the fly from other fields, and, in principle, can be eliminated. In the simplest case, two fields store the same information but in different forms, since the two fields are used for different purposes. For example, product IDs are more efficiently compared as `ints`, but more easily printed as `Strings`. Since it is possible to convert one representation into the other, storing both representations is not necessary, and only makes sense if there is a real cost penalty from performing the data conversion. In the more general case, a field may depend on many other values. For example, you could allocate a field to store the number of items in a shopping cart, or simply compute it by adding up all of the shopping cart items.

There is a trade-off between the performance cost of a conversion or computation and the memory cost of an extra field, which has to be weighed in context. How often is the information needed and how expensive is it to compute? What's the total memory cost? Comparing performance cost to memory cost is a bit like apples and oranges, but often it is clear which resource is most constrained. Here are several considerations to keep in mind:

- Redundant `String` fields should be avoided, since strings have a very high overhead in Java, as we have seen.
- Computed fields are very useful when storing partial values avoids expensive quadratic computation. For example, if you need to support finding the number of children of nodes in a graph, then caching this value for each node is a good idea.

4.6 Optimizing Framework Code

The storage optimizations described in this chapter assume that you are familiar with the entire application you are working on. You need to understand how objects are created and used, and therefore know enough to determine whether these optimizations make sense. However, if you are programming a library or framework, you have no way of knowing how your code will be used. In fact, your code may be used in a variety of different contexts with different characteristics. Premature optimization — making an assumption about how the code will be used, and optimizing for that case — is a common pitfall when programming frameworks.

For example, suppose the online store is designed as a framework that can be extended to implement different kinds of stores. For some stores, most products may have an alternate supplier. For other stores, most products may not. There is no way of knowing. If the `Product` class is designed so that the alternate supplier is allocated as a side object, then sometimes memory will be saved and sometimes wasted. One possibility is to define two versions of the `Product` class, one that delegates and one that doesn't. The framework user can then use the version that is appropriate to the specific context. However, this is generally not practical.

Frequently, decisions are made that trade space for time. There are many instances of this trade-off in the Java standard library. For example, lets look at `String`, which has three bookkeeping fields, an offset, a length, and a hashcode. These 12 bytes of overhead consume 21% of an eight character string. The offset and length fields implement an optimization for substrings. That is, when you create a substring, both the original string and substring share the same character array, as shown in Figure 4.3. The offset and length fields in the substring `String` object specify the shared portion of the character array. This scheme optimizes the time to create a substring, since there is no new character array and no copying. However, every string pays the price of the offset and length field, whether or not they are

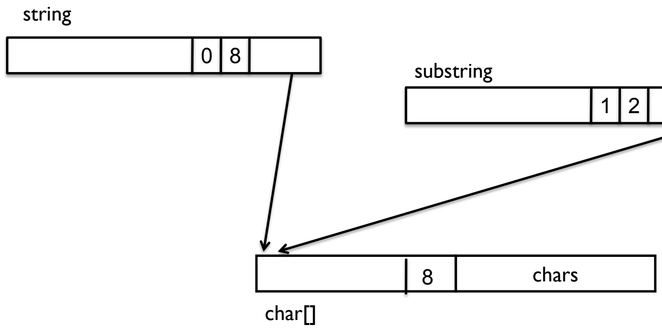


Figure 4.3. A string and a substring share the same character array. The length and offset fields are needed for the substring, but are redundant in the original string object.

used. Across all Java applications, there are many more strings than substrings, so a lot of memory is wasted. Even when there are many substrings, if the original strings go away, you have a different footprint problem, namely, saving character arrays which are too big.

The third bookkeeping field in `String` is a hashcode. Storing a hashcode seems like a reasonable idea, since it is expensive to compute it repeatedly. However, you have to be puzzled by the space-time trade-off, since a string only needs its hashcode when it is stored in a `HashSet` or a `HashMap`. In both of these cases, the `HashSet` or `HashMap` entry already has a field for hashcode for each element. (As if this redundancy isn't enough, there are also four bytes reserved in every object header for the identity hashcode.)

This is a cautionary tale of premature optimization. Framework decisions can have a long-lived impact. For these `String` optimizations, it's not clear that there is any performance gain beyond some unrealistic benchmarks. But it's too late and expensive to change the implementation, so all applications must pay the price in memory footprint.

4.7 Summary

Even though Java does not let you control the layout of objects, it is still possible to make objects smaller by recognizing certain usage patterns. Optimization opportunities include:

- Rarely used fields can be delegated to a side object, or stored in a completely separate attribute table.
- Fields that have the same value in all instances of a class can be declared static.
- Mutually exclusive fields can share the same field, provided they have the same type.
- Redundant fields whose value depends on the value of other fields can be eliminated, and recomputed each time they are used.
-

Every field eliminated saves around 4 bytes per object, which may seem small. However, often several optimizations can be applied to a class, and if the class has the most objects in the heap, then these small optimizations turn out to be significant.

Chapter 5

REPRESENTING VALUES

5.1 Character Strings

Strings vs. Compiled Forms

Strings and StringBuffers

5.2 Dates

5.3 BigInteger and BigDecimal

Chapter 6

SHARING IMMUTABLE DATA

So far, we have been concerned with the wasteful overhead that results from data representation. But what about the data itself? If you examine any Java heap, you will find that a large amount of the data is duplicated. At one extreme, there are often thousands of copies of the same boxed integers, especially 0 and 1. At the other extreme, there may be many small data structures that have the same shape and data. And, of course, duplicate strings are extremely common. This chapter describes various techniques for sharing data to avoid duplication, including a few low-level mechanisms that Java provides.

6.1 String Literals

Duplicate strings are not only one of the most common sources of memory waste, they are also very expensive, since even small strings incur a large overhead. Fortunately, it is not hard to eliminate string duplication.

One technique is to represent strings as literals whenever possible. Duplication problems arise because dynamically created `String`s are stored in the heap without checking whether they already exist. `String` literals, on the other hand, are stored in a *string constant pool* when classes are loaded, where they are shared. Therefore, there is a big advantage to `String` literals.

As an example, suppose an application reads in property name-value pairs from files into tables:

```
class ConfigurationProperties {
    ..
    void handleNextEntry() {
        String propertyName = getNextString();
        String propertyValue = getNextString();
        propertyMap.put(propertyName, propertyValue);
    }
}
```

The `String`s stored in `propertyMap` are created dynamically. If there are just a few distinct property names in all of the input pairs, these property names will be duplicated many times in the heap.

However, if you know in advance what all of the property names are, then you can define them once as `String` literals, which can be shared among the entries of `propertyMap`.

```
class PropertyNames {
   public static String numberOfUnits = ‘‘NUM_UNITS’’;
   public static String minWidgets = ‘‘MIN_WIDGETS’’;
   ..
}

class ConfigurationWithStaticProperties {
   void handleNextEntry() {
      String propertyName = getNextPropertyName();
      String propertyValue = getNextString();
      propertyMap.put(propertyName, propertyValue);
   }
}
```

The `getNextPropertyName` method reads in a property name, and returns a pointer to a property name literal, stored in the JVM string constant pool. Alternatively, defining an enumeration type to encode property names may be a better stylistic choice.

A common mistake is to create a new `String` from a `String` literal, which is usually completely unnecessary:

```
class PropertyNames {
   public static String numberOfUnits =
                             new String(‘‘NUM_UNITS’’);
   public static String minWidgets =
                             new String(‘‘MIN_WIDGETS’’);
   ..
}
```

Even though the standard library is smart enough to share character arrays in this case, this code still creates redundant `String` objects in the heap.

Using `String` literals to avoid dulication is only possible when the `String` values are known in advance. Section 11.1 introduces the notion of a sharing pool for sharing dynamic data. Section 5.3 describes the Java string interning mechanism, which uses a built-in string sharing pool to eliminate duplication.

6.2 Sharing Pools

Suppose an application generates a lot of duplicated data and the values are unknown before execution. You can eliminate data duplication by using a *sharing pool*,

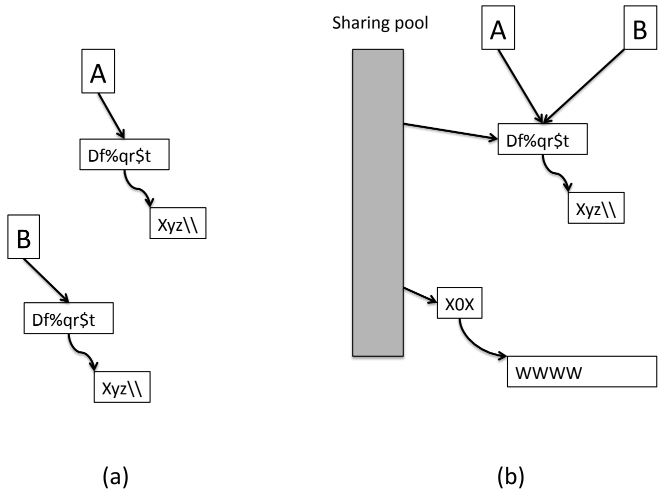


Figure 6.1. (a) Objects A and B point to duplicate data. (b) Objects A and B share the same data, stored in a sharing pool.

as shown in Figure 5.1. In Figure 5.1(a), objects A and B point to identical data structures. Figure 5.1(b) shows objects A and B sharing the same data structure, which is stored in a sharing pool.

Sharing Pool

A *sharing pool* is a centralized structure that stores canonical data values that would otherwise be replicated in many objects. A sharing pool itself is usually some sort of hash table, although it could be implemented in other ways.

There are several issues that you need to be aware of before using a sharing pool.

Shared objects must be immutable. Changing shared data can have unintended side effects. For example, changing A in Figure 5.1(b), also changes the value of B.

The result of equality testing should be the same, whether or not objects are shared. In particular, you should never use == on shared objects. In Figure 5.1, `A==B` is false in Figure 5.1(a) and true in Figure 5.1(b), which can lead to very subtle bugs. It is dangerous to use == in any case, unless exact object identity is really required.

Sharing pools should not be used if there is limited sharing. A sharing pool itself adds memory costs, including additional per-entry costs. If there is not much sharing, then the memory saved from eliminating duplicates isn't enough to compensate for the extra cost, and memory will be wasted instead of saved.

Shared objects should be garbage collected. In Figure 5.1(b), the sharing pool stores an object that no other object is pointing to. Over time, the sharing pool can fill up with garbage, that is, items that were once needed but not any more. If the sharing pool is not purged of these unused items, there is a memory leak that can eventually use up all of memory.

Fortunately, Java provides a few built in mechanisms that take care of some of these concerns.

6.3 String Interning

Since `String` duplication is so common, Java provides a built-in string pool for sharing `String`s, implemented by the native JVM and maintained in its internal *perm space*. To share a `String`, you simply call the method `intern` on it, and everything is taken care of automatically. Since `String`s are immutable, sharing is safe. However, the rule about not using == still holds on shared `String`s.

In the example from section **??**, `ConfigurationWithStaticProperties` eliminates property name duplication but not property value duplication. Suppose you know that there are not too many distinct values, but you don't know what they are. In this case, property values are perfect candidates for interning.

```
  class ConfigurationPropertiesWithInterning {
     void handleNextEntry() {
        PropertyName propertyName = getNextPropertyName();
        String propertyValue = getNextString().intern();
        propertyMap.put(propertyName, propertyValue);
     }
 }
```

The call to `intern` adds the new property value `String` to the internal string pool if it isn't there already, and return a pointer to it. Otherwise, the new `String` is a duplicate, and a previously saved `String` is returned.

Even though interned `String` are not stored in the heap, they do incur native memory overhead, and native memory is not free. Interning `String`s indiscriminately wastes memory and can result in an exception: `java.lang.OutOfMemoryError:PermGen Space`. There are JVM parameters to adjust the perm space size: XX:PermSize=128m sets perm size to 128 megabytes, and -XX:MaxPermSize=512m sets the maximum perm size to 512 megabytes. Fortunately, the JVM performs garbage collection on the internal string pool, so there is no danger of a memory leak.

6.4 Integer Sharing Pool

The Java library provides a sharing pool for `Integer`s. Unlike the string pool, the `Integer` pool is initialized at class load time to store all `Integer`s in a fixed range, from -128 to 127 by default. The method `Integer.valueOf(int value)` returns a pointer to an `Integer` in the pool, provided `value` is in range.

Because the `Integer` sharing pool is pre-initialized and fixed in size, it's always a good idea to call `Integer.valueOf` instead of the constructor to create a new `Integer`. For example, the following code stores `Integer`s from 1 to 500 in an array:

```
for (int i = 1; i <= 500; i++) {
    numbers[i] = Integer.valueOf(i);
}
```

For the first 127 numbers, `valueOf` returns an existing `Integer`. For the rest of the numbers, `valueOf` returns a new `Integer`. Calling `Integer.valueOf` incurs no extra overhead, and you never have to worry about wasting memory or getting a memory exception. The only precaution is avoid using == to compare potentially shared `Integer`s.

There is a JVM parameter to change the size of the `Integer` sharing pool: -XX:AutoBoxCacheMax=100 sets the high value in the pool to 100.

6.5 Sharing Objects

Beyond strings and boxed `Integer`s, there are often other kinds of duplicated objects and data structures consuming large portions of the heap. There is no built-in Java mechanism to share objects or data structures in general, so you have to implement a sharing pool for them from scratch. All of the sharing pool issues from Section 11.1 need to be addressed. The shared objects or structures must be immutable, they must not be compared using ==, there must be sufficient memory savings from sharing to justify the sharing pool, and the sharing pool must be not cause a memory leak. Note that, in general, `equals` is implemented as ==, so sharing data structures typically requires writing a new `equals` method.

To illustrate a user-written sharing pool, consider a graph where the nodes have annotations, many of which are duplicates. Both the graph and the annotations are modified dynamically. The two basic requirements are 1) the ability to find existing annotations quickly to share them, and 2) the ability to release annotations that are no longer associated with any node, so they can be garbage collected. The second requirement prevents a memory leak.

Interestingly, none of the common collection classes meet these requirements out-of-the-box. A `HashSet` can store `Annotation`s uniquely, but retrieving an existing `Annotation` is not easy. The first requirement is best implemented as a `HashMap` mapping `Annotation`s to themselves:

```
HashMap<Annotation><Annotation> (1)
```

For the second requirement, Java provides some support for releasing unused items from collections, namely, `WeakReferences` and `WeakHashMaps`. A `WeakReference` is an object wrapper. An object that is referenced by a `WeakReference` can be reclaimed by the garbage collector if there are no other strong references to it. A `WeakHashMap` is a hashmap that stores keys as `WeakReferences`. That is, when there are no strong references to a key, the entire entry is freed for garbage collection. For example, we want an `Annotation` to go away when its associated `Nodes` are no longer used. This can be implemented by a `WeakHashMap`:

```
WeakHashMap<Node><Annotation> nodeAnnotations;   (2)
```

This looks close to what we need. So let's modify (1) to be a `WeakHashMap`, so that when an `Annotation` is no longer used, its sharing pool entry is freed for garbage collection:

```
WeakHashMap<Annotation><Annotation> sharingPool;
```

This should do it. Wrong! Both the key and the value of the sharingPool reference the same `Annotation` object. The key is a weak reference, but the value is a strong reference, which prevents an `Annotation` object from every being released. The value must also be a `WeakReference`. Here is the correct implementation of a sharing pool for `Annnotations`:

```
class AnnotationFactory {
  static WeakHashMap<Annotation, WeakReference<Annotation
     >>sharingPool =
                new WeakHashMap<Annotation, WeakReference<
                   Annotation>>();

  public Annotation getAnnotation(Annotation annotation) {
     if (annotation == null) return null;
     WeakReference<Annotation> wref = sharingPool.get(
        annotation);
     if (wref != null) {
        Annotation oldAnnotation = wref.get();
        if (oldAnnotation != null) {
           return oldAnnotation;
        }
     }
     sharingPool.put(annotation, new WeakReference(
        annotation));
     return annotation;
  }
```

```
        ..
    }
```

As this example shows, Java's weak referencing capability has subtle semantics and is not easy to use. Weak referencing is a lifetime management facility, and is discussed in greater detail in Chapter ?? .

6.6 StringBuffer vs. String

It is well-known that `StringBuffer` is more efficient than `String` for performing string concatenation. Since `String`s are immutable, concatenating `String`s involves allocating a temporary `char` array, copying the `String`s into it, and then constructing a result `String`. A `StringBuffer`, on the other hand, is mutable. If the `StringBuffer` capacity is sufficient, then `String`s can be concatenated by simply appending them to the `StringBuffer`.

However, long-lived `StringBuffer`s can waste memory. Usually a `StringBuffer` is 40% empty space, since they double in size whenever they need to be reallocated. Typically, after a string is built up in the `StringBuffer`, it is stable, at which point it should be converted to a `String`, so that the `StringBuffer` can be garbage collected. Using `StringBuffer`s to facilitate building a `String` is fine, but they should be used only as temporaries.

6.7 Summary

Not only are Java heaps bloated from too much overhead, they are also bloated from duplicated data. If you know that your application generates many copies of the same data, then you should find a way to share the data. Java provides several built-in sharing mechanisms:

- The JVM maintains a native sharing pool for `Strings`. Use `String` interning to make use of this sharing pool.
- The standard library maintains a fixed size pool for a fixed range of `Integer`s. You should use `valueOf` to create `Integer`s instead of a constructor.

Additionally, Java provides a weak referencing mechanism, which can be used to implement your own sharing pool.

These mechanisms appear to be clumsy addons that were necessary to solve problems that came up in practice. But they are better than nothing, and without them, it would be much harder to share data. Whether or not you use these mechanisms, you should remember these four rules:

- Shared objects must be immutable.
- The result of equality testing should be the same, whether or not objects are shared.

- Sharing pools should not be used if there is limited sharing.
- Shared objects should be garbage collected.

Chapter 7

A BRIEF OVERVIEW OF COLLECTIONS

Chapter 8

USING COLLECTIONS: RELATIONSHIPS

Relationships in an entity-relationship model are typically implemented in Java using the standard library collection classes. It is common for a Java application to create hundreds of thousands, even millions, of collections, where the vast majority are empty or contain only a few entries. Our guess is that the collection class developers would be surprised by this usage pattern. Why bother implementing expandable structures and clever hashing algorithms for only a few entries? This mismatch between collection implementation and usage is a leading cause of memory bloat. The basic cost of a collection, even an empty collection, is remarkably high. Creating millions of small collections multiplies this basic infrastructure cost, which is all overhead, filling the heap. This chapter shows how to mitigate the many-small-collections problem to reduce memory bloat.

8.1 Choosing The Right Collection

The standard Java collections vary widely in terms of memory consumption. Not surprisingly, the more functionality a collection provides, the more memory it consumes. Collections range from simple, highly efficient `ArrayLists` to very complex `ConcurrentHashMaps`, which offer sophisticated concurrent access control at an extremely high price. Using overly general collections, that provide more functionality than really needed, is a common pattern leading to excessive memory bloat. Since collection implementations are hidden, it's easy to see how this happens.

To illustrate, consider a graph with 100,000 nodes that have four edges each on average. A straight-forward implementation is to use a `HashMap`, where the keys are nodes and the values are `HashSets` of edges. In this example, a node is an `Integer`, and an edge consists of two `Integers`, a node number and an edge weight. Figure 6.1 shows an entity-collection diagram for the graph.

This is a hugely bloated structure with a bloat factor of 95.7%. There are multiple problems here, that we will address in this chapter one by one. The first problem is a collection choice problem, namely, the edges are represented by 100,000 very small `HashSets`, each consuming 232 bytes, which is all overhead. It's hard

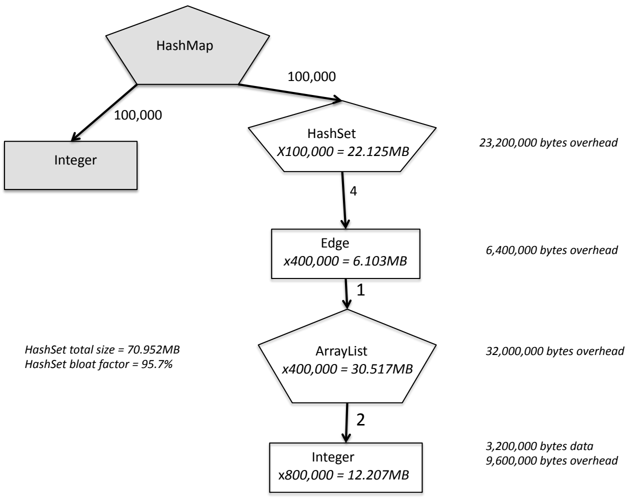


Figure 8.1. A 100,000-node graph, stored as a `HashMap` from nodes to `HashSets` of edges.

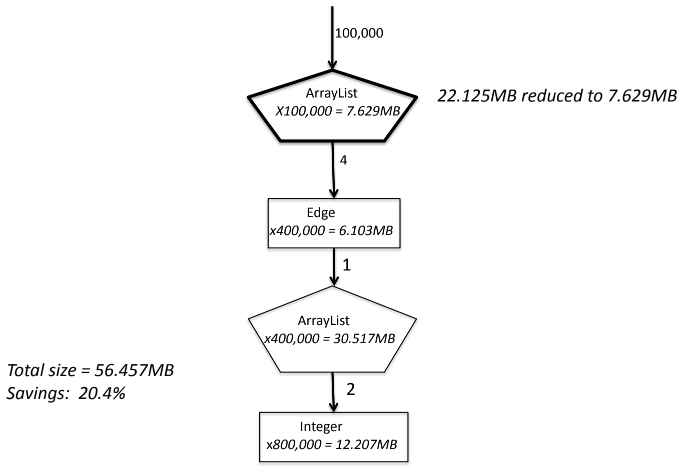


Figure 8.2. A 100,000-node graph, stored as a `HashMap` from nodes to `ArrayLists` of edges.

to think of a good reason why such a heavy-weight collection should ever be used for storing just a few entries, and yet, this pattern arises in almost all applications. For small sets, `ArrayList` is almost always a better choice. `HashSet` maintains uniqueness and provides fast access, but enforcing uniqueness is not always needed. If uniqueness is important, it can be enforced for an `ArrayList` with a little extra checking code, and usually without significant performance loss when sets are small. Figure 6.2 shows improved memory usage with `ArrayList`. Each `ArrayList` incurs 80 bytes of overhead, approximately a third the size of a `HashSet`. This simple change saves 14.49MB, which is a reduction of 20.4%.

8.2 The Cost Of Collections

Let's look at why a `HashSet` is so much bigger than an `ArrayList`. Some of the `HashSet` overhead is Java-related and unavoidable. Other overhead is the result of hard-coded assumptions, for example, the `HashSet` implementation assumes `HashSets` will be very large, and sacrifices memory space in favor of performance. Here is a breakdown of `HashSet` design decisions leading to unnecessary bloat, as illustrated in Figure 6.3.

Reusing HashMap. Internally, a `HashSet` is just a wrapper, delegating all of its work to a `HashMap`. This decision to reuse the `HashMap` code instead of specializing

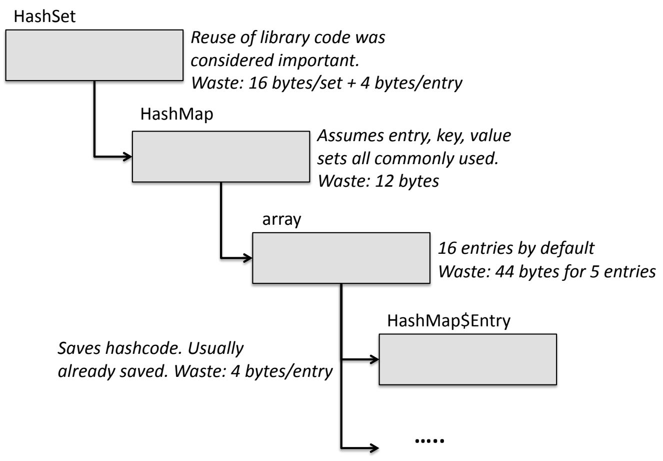

HashSet
Reuse of library code was considered important. Waste: 16 bytes/set + 4 bytes/entry
HashMap
Assumes entry, key, value sets all commonly used. Waste: 12 bytes
array
16 entries by default Waste: 44 bytes for 5 entries
HashMap$Entry
Saves hashcode. Usually already saved. Waste: 4 bytes/entry
.....

Figure 8.3. The internal structure of a `HashSet` showing how implementation assumptions waste memory.

HashSet costs an extra 16 bytes, which is reasonable if HashSets are big and the fixed overhead costs are amortized away. Unfortunately, the fixed cost is multiplied when there are many small HashSets. Also, HashMap is more general than what is needed for a HashSet. Each HashMap$Entry stores a key and a value, and HashSet only uses the key, so four bytes are wasted per entry. Sometimes specialization is a better option than reuse, especially for library classes, where the usage patterns are not known in advance.

Open Chaining. HashMap itself is fairly expensive. First, the HashMap object is just a container pointing to the actual array of entries. This delegation is necessary in Java. Secondly, HashMap uses an open chaining algorithm, which means that clashing entries are chained in a linked list. With open chaining, each entry requires its own HashMap$Entry object and an extra level of indirection.

Default Array Size. A HashMap, which is used to implement a HashSet, has a default size of 16 entries, which means that its array of entries has an initial size of 16. This wastes space when most HashSets have only a few entries. For example, a HashSet with five entries wastes 44 bytes.

Bookkeeping Fields. HashMap allows callers to iterate over its set of keys, set of values, and set of entries. These sets are cached once they are created. Every HashMap has three pointers to store these sets, which is an extra 12 bytes. However, its not that common to use any of these sets, and very rare to use more than one set at a time. Certainly when the HashMap is part of a HashSet, there is never a need for a set of values.

Extra Per-Entry Costs. Each HashMap$Entry stores a hashcode, which is an unnecessary redundant field. The most common keys are either Strings, which store their own hashcode, or Integers, whose hashcodes are easy to compute.

In contrast, ArrayList has a smaller fixed cost and a smaller per-entry cost than HashSet. It is really just an expandable array, consisting of a wrapper object and an array of entries, as shown in Figure 6.4. Lower fixed cost means that an ArrayList with just a few elements is smaller than a HashSet with the same elements. Fixed costs include wrapper objects and unitialized array elements. The fact that HashSet delegates to HashMap dramatically inflates its fixed cost. Lower per-entry cost means that ArrayList scales much better than HashSet. The per-entry cost of an ArrayList entry is an entry array pointer, which is 4 bytes. The per-entry cost of a HashSet is an entry array pointer plus a HashMap$Entry, which is 28 bytes. So ArrayList is better both for small and large sets.

Table 6.1 shows the empty-collection costs of four basic collections, ArrayList, LinkedList, HashMap, and HashSet. These costs have been calculated based on the Sun JVM, using the techniques described in Chapter 3. The various other Java standard library implementations in circulation have costs similar to these. You can calculate them using the same methodology.

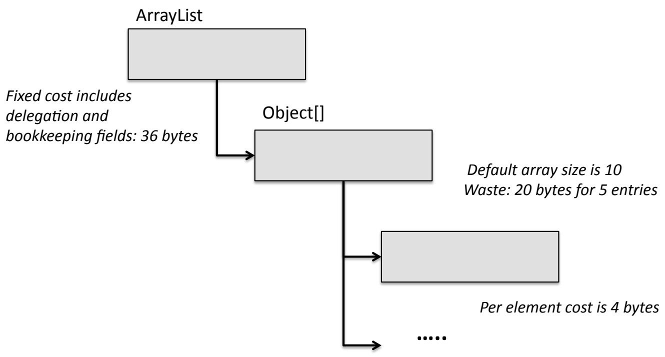

ArrayList
Fixed cost includes delegation and bookkeeping fields: 36 bytes
Object[]
Default array size is 10
Waste: 20 bytes for 5 entries
Per element cost is 4 bytes
.....

Figure 8.4. The internal structure of an `ArrayList`, which has a relatively low fixed overhead, and is scalable.

Collection	Fixed Cost	Per-Entry Cost	Default Capacity
ArrayList	80 bytes	4 bytes	10 entries
LinkedList	48 bytes	24 bytes	1 sentinel entry
HashMap	120 bytes	28 bytes	16 entries
HashSet	136 bytes	28 bytes	16 entries

Table 8.1. The cost breakdown of four basic Java standard library collections with default size and no entries. The fixed cost includes an array whose size equals the default capacity. The per-entry cost is used to determine scalability.

8.3 Properly Sizing Collections

Collections such as `HashMap` and `ArrayList` store their entries in arrays. When these arrays become full, a larger array is allocated and the entries are copied into the new array. Since allocation and copying can be expensive, these entry arrays are always allocated with some extra capacity, to avoid paying these growth costs too often. This is why the initial capacity of an `ArrayList` is 10 rather than zero or one, and why its capacity increases by 50% when it is reallocated. Similarly, the capacity of a `HashMap` starts at 16, and grows by a factor of 2 when the `HashMap` becomes 75% full.

These default policies trade space for time, on the assumption that collections grow. However, typical applications have hundreds of thousands of small collections that don't grow. As a result, there is no performance gain, and the extra empty array slots can add up to significant bloat problem. Unless you take explicit action, element arrays are almost always too big.

Fortunately, for `ArrayLists`, it is possible to right-size them. If you know that an `ArrayList` has a maximum size x less than the default size, then it's worth passing x as a parameter to the constructor. This sets the initial capacity of the `ArrayList` to x. However, if you are wrong and the `ArrayList` grows bigger than x, then it will grow by 50%, which may be worse than just taking the default initial size.

Alternatively, you can call the `trimToSize` method which shrinks the entry array by eliminating the extra growth space. Trimming reallocates and copies the array, so it is expensive to keep calling `trimToSize` while an `ArrayList` is still growing. Trimming is appropriate after it has been fully constructed and will never grow again. In fact, applications often have a build phase followed by a used phase. `ArrayLists` can be trimmed between these two phases, so that the cost of reallocation and copying is paid only once.

Returning to the graph example, all `ArrayLists` in Figure 6.2 have the default capacity. If we assume that the the graph is built in one phase, and used in another phase, then it is possible to trim all of the `ArrayLists`. This should save quite a bit of space, since there are 100,000 `ArrayLists` with four entries on average, and 400,000 `ArrayLists` with two entries each. In fact, trimming these `ArrayLists` saves 15.2 million bytes, as shown in Figure 6.5. The total size is reduced by 25.6%.

`HashSets` and `HashMaps` do not have `trimToSize` methods, but it is possible to pass the initial capacity and load factor as constructor parameters when creating a `HashSet` or `HashMap`. However, before changing the initial capacity, you should ask yourself whether using a `HashSet` or `HashMap` is a wise decision in the first place. If you are going to end up with many collections with fewer than 16 elements, perhaps there is a more memory-efficient solution, like `ArrayList`.

A `LinkedList` is another alternative for small collections, and are better than `ArrayLists` if the collections are changing a lot. The 24 byte per-entry cost is larger, but there is no element array, and only one extra entry, which is a sentinal.

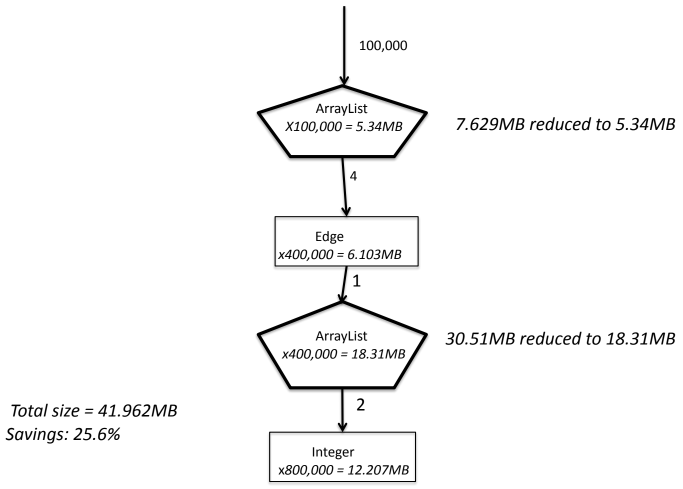


Figure 8.5. The graph example after all of the `ArrayLists` have been trimmed by calling the `trimToSize` method.

8.4 Avoiding Empty Collections

Too many empty collections is another common problem that fills up space in the heap. A quick look inside an empty collection shows that it is not all that empty. According to Table 6.1, an empty `HashMap` consumes 120 bytes, and an empty `ArrayList` consumes 80 bytes, assuming a default initial size. Even if the initial size is zero, empty collections are still large. A zero-sized `HashMap` consumes 56 bytes, and a zero-sized `ArrayList` consumes 40 bytes. Empty `HashSets` are even bigger.

Empty collection problems are generally caused by eager initialization, that is, allocating collections before they are actually needed. Exacerbating this problem, collections themselves also allocate their internal objects in an eager fashion. For example, `HashMap` allocates its entry array before any entries are inserted. You might think that eager initialization is not such a big problem, since entries will be added eventually. However, often collections are allocated just in case they are needed later, and remain empty throughout the execution.

Suppose the graph in Figure 6.5 is initialized by the code:

```
ArrayList nodes;
HashMap graph;
int numNodes;
   ..
   public void initGraph() {
       ..
       for (i = 0; i < numNodes; i++) {
          graph.put(nodes.get(i), new ArrayList());
       }
   }
```

Initially, each node is mapped to an empty edge `ArrayList`, so there are 100,000 empty `ArrayLists` before any edges are inserted. As the graph is populated, many of these `ArrayLists` will become non-empty, but it is likely that a good number of nodes have no edges. If 25% of the nodes in the final graph still have no edges, there will be 25,000 empty `ArrayLists`, which consume .95MB after calling `trimToSize`. Figure 6.6 shows the entity-collection diagram after removing 25,000 empty edge `ArrayLists`. Note that there are now only 75,000 edge `ArrayLists` shown, and since there are no more empty `ArrayLists`, the average fanout increases to 5.33.

Delaying allocation prevents creating too many empty collections. That is, instead of initializing all of the collections that you think you may need, allocate them on-demand, just before inserting an edge. On-demand allocation requires more checking code, to avoid `NullPointerExceptions`.

Alternatively, you can initialize collection fields to reference static empty collections, including EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. For example, calling static method `Colletions.emptyList` to initialize edge `ArrayLists` maps

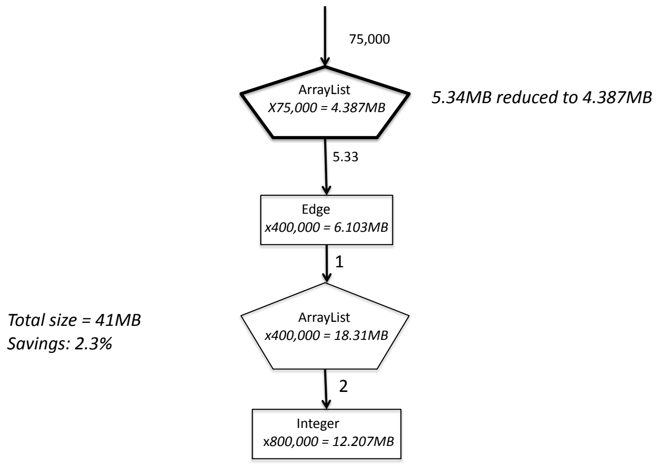


Figure 8.6. The graph example without empty `ArrayLists`.

all nodes to a singleton immutable static `ArrayList`, so that no empty `ArrayLists` are created:

```
public void initGraph() {
      ..
      for (i = 0; i < numNodes; i++) {
         graph.put(nodes.get(i), Collections.emptyList());
      }
  }
```

This initialization avoids the need to check whether an `ArrayList` exists at every use. For example, the size method and iterators will just work. However, you have to be careful not to let any references to static empty collections escape its immediate context. If you give out a reference to a static empty collection, then theres no way to update this escaped empty-collection reference once an actual collection is allocated.

8.5 Fixed Size Collections

Java collections can grow to be arbitrarily big, but this functionality comes at a cost. Collections include wrapper objects, and may be sized with extra growth room. If you know that the size of a collection is fixed, having the ability to expand the collection is unnecessary, and it is better to choose a cheaper alternative. In fact, often you can use a simple Java array, and not use a collection at all.

Let's look more closely at an `Edge` in the graph example in Figure 6.7(a). An `Edge` object is a wrapper containing an `ArrayList`, which is just another wrapper pointing to an `Integer` array. Since an `Edge` will never hold more than two `Integers`, it isn't necessary to store these in an `Arraylist` — a simple array will do. This change, eliminating the `ArrayList` object, removes 24 bytes per `Edge`, as shown in Figure 6.7(b), adding up to 9.16MB. This is an example of choosing an overly-general collection. In this situation, you don't need a collection at all. Using `ArrayList` for storing fixed- or bounded-size arrays is a common practice that can be easily avoided.

There is one final optimization that can be performed on the `Edge` class. Namely, making the two `Integers` fields of `Edge` instead of elements of an array, as shown in Figure Figure 6.7(c). This optimization eliminates the array object, which saves another 24 bytes per edge, which is another 9.16MB.

In effect, these two steps have transformed a dynamic representation of a type, where field values are stored in arrays or `ArrayLists`, into a static representation, namely, a class. Using fixed-size arrays or `ArrayLists` to store fields is a typical pattern in Java. For example, when reading records from a database, it is a common practice to store the data fields of a record in an array or `ArrayList`. If you do not know that the fields are until execution, then this representation makes sense. Even though this representation introduces extra objects, there is often no alternative.

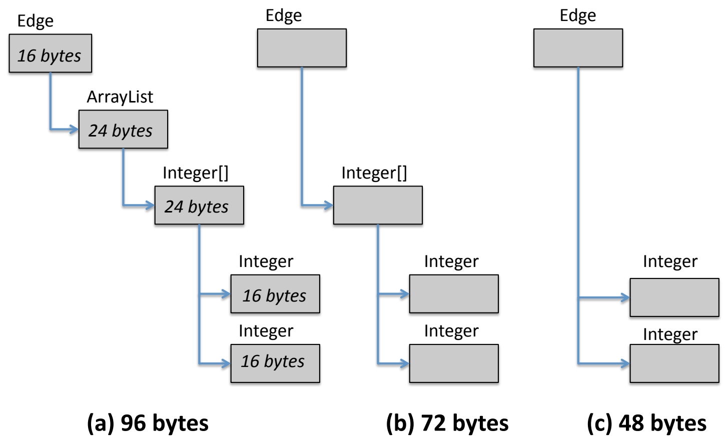


Figure 8.7. (a) An `Edge` has 96 bytes. (b) Replacing the `ArrayList` by an array eliminates 24 bytes. (c) Inlining the array eliminates another 24 bytes.

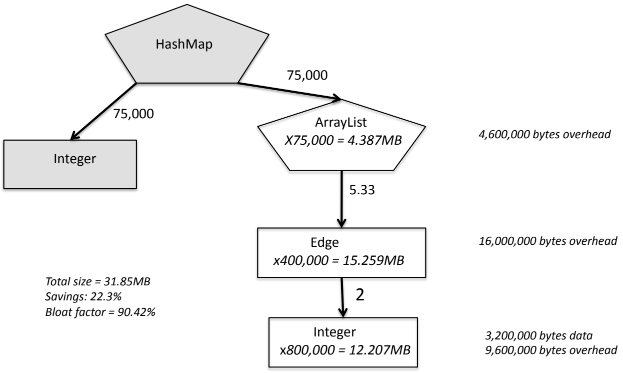


Figure 8.8. Final entity-collection diagram for the graph example.

However, if you do know the structure in advance, it is far better space-wise to inline these fields in a class definition,

The resultin g entity-collection diagram for the graph example is shown in Figure 6.8. The total size, from the first entity-collection diagram in Figure 6.1, has been reduced by 68% , which is quite significant. The bloat factor has gone from 95.7% to 86.6%. You might wonder why the bloat factor is still so high? The answer is that the overhead of nesting an `ArrayList` in a `HashMap` is still very high compared to the data, which is just two `Integers`. There is still one more simple optimization that can be performed, namely, replacing the `Integers` by primitive `int` fields in `Edge`. Calculating the bloat factor after making this transformation is left as an exercise for the reader.

8.6 Summary

This chapter describes a number of ways to mitigate high memory costs when your program has many small collections:

- Choose the most memory-efficient collection for the job at hand. In particular when collections have at most a few elements in them, you don't need expensive functionality like hashing.

- Make sure collections are properly sized. If you know that a collection will not grow any more, then there is no reason to maintain extra room for growth.

- Avoid lots of empty collections. It is common to allocate collections ahead of time, whether or not they will eventually ever be used. If you postpone creating them until they are needed, often you will end up with fewer collections, and no empty collections.

- Use arrays instead of `ArrayLists` when you know the maximum size of the `ArrayLists` in advance.

- Avoid dynamic types, where field values are stored in arrays or `ArrayLists` instead of object fields, if possible. It's better to use a class to represent a type, if the type is really static.

Small collections have a fixed-size overhead, which is there as soon as the collection is allocated. Fixed costs are amortized when collections are big, but when they have only a few elements, the fixed overhead dominates the memory cost. The fixed costs of collections is provided in Section 6.2 to help you make an informed choice. Next chapter looks at per-entry costs of larger collections and nested collections.

Chapter 9

USING COLLECTIONS: TABLES AND INDEXES

Chapter 10

USING COLLECTIONS: ATTRIBUTE MAPS AND DYNAMIC RECORDS

Chapter 11

ADDITIONAL COLLECTION BEHAVIORS

Part II

Managing the Lifetime of Data

Chapter 12

LIFETIME REQUIREMENTS

Your application needs some objects to live forever and it needs the rest to die a timely death. Unfortunately, some of the important details governing memory management are left in your hands. Java promised, with its automatic memory management, that you could create objects without regard for the messy details of storage allocation and reclamation. In Java, you needn't explicitly free objects, which is at once the saviour from, and the source of, many problems with memory consumption. Unless you are careful, your program will suffer from bugs such as memory leaks, race conditions, lock contention, or excessive peak footprint. Furthermore, if your objects don't easily fit into the limits of a single Java process, you will need to manage, explicitly, marshalling them in and out of the Java heap.

Very often, your application uses a data structure in a way that falls into one of a handful of common *lifetime requirements*. The nature of each requirement dictates how much help you will get from the Java runtime in the desired preservation and reclamaion of objects, and where it leaves you to your own devices.

An important step in the design process of any large application is understanding the lifetime requirement for each of your data models. In this chapter, we describe the five common lifetime requirements: objects needed only transiently, objects needed for the duration of the run, objects whose lifetime ends along with a method invocation, objects whose lifetime is tied to some other object, and, most difficult of all, objects that live or die based on need. Table 9.1 summarizes these five important requirements. We step you through each of the requirements, defining them and giving examples of how to know when you have an instance of each.

Once you have mapped out the lifetime requirements of your data models, the next step is to chose the right implementation details in order to correctly implement each requirement. The remaining chapters in this part show how to implement these requirements.

12.1 Object Lifetimes in A Web Application Server

To introduce the common requirements for object lifetime, we walk through several scenarios found in most long-running server applications. These applications provide an interesting case study for lifetime management. Managing lifetime when

Lifetime Requirement	Example
Temporary	new parser for every date
Correlated with Another Object	object annotations
Correlated with a Phase or Request	tables needed only for parsing
Correlated with External Event	session state, cleared on user logout
Permanently Resident	product catalog
Time-space Tradeoff	database connection pool

Table 12.1. Common requirements for the lifetime of objects that your application must implement properly, in order to avoid correctness or performance problems.

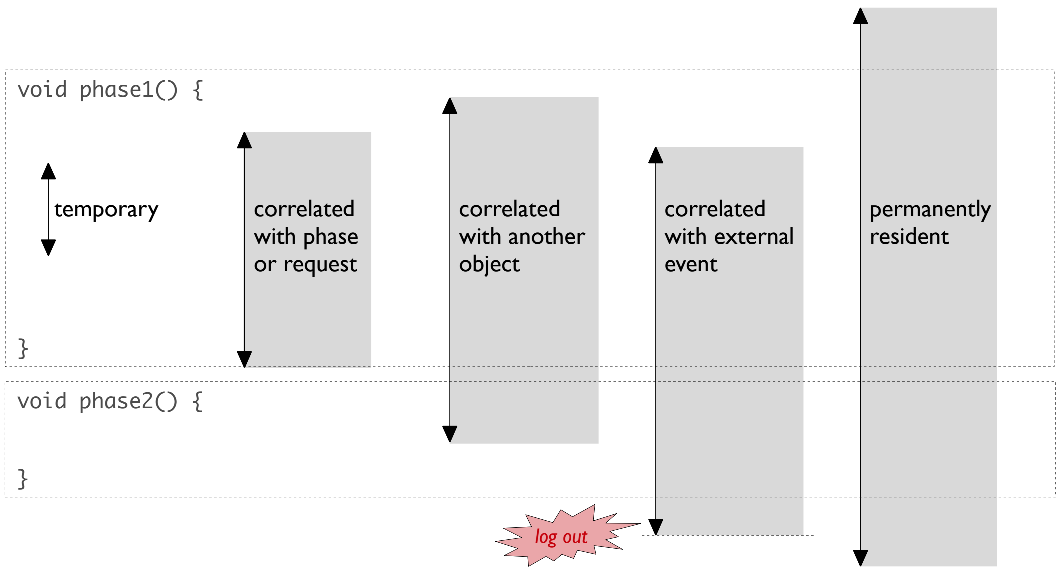

void phase1() {
temporary
correlated with phase or request
correlated with another object
correlated with external event
permanently resident
}
void phase2() {
}
log out

Figure 12.1. Illustration of some common lifetime requirements.

the application runs forever is an especially complex issue. This is true for more than for servers alone. Desktop applications such as the Eclipse integrated development environment shares many of the same challenges. Improperly managing the lifetimes of objects, for short-running applications, often does not result in critical failure. Indeed, the application often finishes its run before one would even notice a problem with memory consumption. Plus, you're probably don't run many instances of a short-running application simultaneously; and so achieving the ultimate in scalability is not a primary concern. In contrast, if an application runs more or less forever, then mistakes pile up over time. In addition, caching plays a large role in these applications, since they often depend on data fetched from remote servers, or from disk, neither of which can support the necessary throughput and response time requirements. The ability for mistakes to pile up, and for misconfigured or poorly implemented caches to impede performance means that special care must be taken when implementing your server application.

The heap consumption of this application will fluctuate over time. A timeline view of expected memory consumption helps to visualize these changes. It visualizes memory during the lulls and peaks of activity, as requests are processed and when sessions time out, and as the server starts up. We will use these over the next few pages, as we walk through the common cases of lifetime requirements.

To help introduce the common lifetime requirements, we walk through an example of a shopping cart server application. The server, on startup, preloads catalog data into memory to allow for quick access to this commonly used data. It also maintains data for users as they interact with the system, browsing and buying products. Finally, it caches the response data that comes from a remote service provider that charges per request. The remainder of this chapter walks you through understanding the lifetime requirements of these data structures.

12.2 Temporaries

The catalog data and session state are both examples of objects that are expected to stick around for a while. In the course of preloading the cache and responding to client requests, the server application will create a number of objects that are only used for a very short period of time. They help to faciliate the main operations of the server. These temporary objects will be reclaimed by the JREs garbage collector in relatively short order. The point at which an object is reclaimed depends on when the garbage collector notices that it is reclaimable. Normally, the garbage collector will wait until the heap is full, and then inspect the heap for the objects that are still possibly in use. In this way, the area under the *temporaries* curve has a see-saw shape. As the temporaries pile up, waiting for the next garbage collection, they contribute more and more to memory footprint. Normally, once the JRE runs a garbage collection, these temporaries no longer in use will no longer contribute to heap consumption.

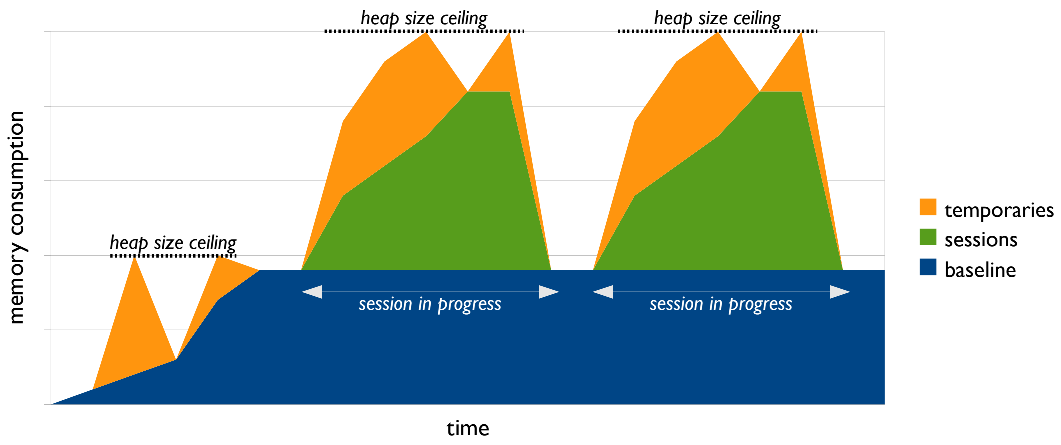


Figure 12.2. Memory consumption, over time, typical of a web application server.

In this way, temporary objects *fill up the headroom* in the heap. If there is a large amount of heap space unused by the longer-lived objects, then the temporaries can be reclaimed less often. This is a good thing, because a garbage collection is an expensive proposition. When configuring your application, you may specify a maximum heap size. It should certainly be larger than the baseline and session data. How much larger than that? This choice directly affects the amount of *headroom*, that is the amount of space available for temporaries to pile up.

Temporaries in Practice If your application is like most Java applications, it creates a large number of these temporary objects. They hold data that will only be used for a very short interval of time. It is often the case that the objects in these transient data structures are only ever reachable by local variables. For example, this is the case when you populate a `StringBuilder`, turn it into a `String`, and then ultimately (and only) print the string to a log file. The point at which these objects, the string builder, string, and character arrays, are no longer used is only shortly after they are constructed:

```
String makeLogString(String message, Throwable exception) {
    StringBuilder sb = new StringBuilder();
    sb.append(message);
    sb.append(exception.getLocalizedMessage());
    return sb.toString();
}
void log(String message, Throwable exception) {
    System.err.println(makeLogString(message, exception));
}
```

A temporary object serves as a transient home for your data, as it makes its way through the frameworks and libraries you depend on. Temporaries are often necessary to bridge separately developed code and enable code reuse. The above example avoids code dupliation and ensures uniformity of the output data by factoring out the logic of formatting messages into the `makeLogString` method.

In many cases, the JRE will do a sufficient job in managing these temporary objects for you. Generational garbage collectors these days do a very good job digesting a large volume of temporary objects. In a generational garbage collector, the JRE places temporary objects in a separate heap, and thus need only process the newly created objects, rather than all objects, during its routine scan.

There are two potential problems that you may encounter with temporary objects. The first is the runtime cost of initializing the state of the temporary objects' fields. Even if allocating an freeing up the memory for an object is free, there remains the work done in the constructor:

```
class Temp {
    private final Date date;

    public Temp(String input) { // constructor
        this.date = DateFormat.getInstance().parse(input);
    }
}
```

Even if an instance of `Temp` lives for only a very short time, its construction has a high cost. It is often the case that this expense is hidden behind a wall of APIs. If so, then what you think of as trivial temporary (since you, after all, are in control of when the instance of `Temp` lives and dies), would in actuality be far from trivial in runtime expense. Expenses can pile up even further if temporary object constructions are nested.

There is a second potential problem with temporary objects. By creating temporary objects at a very high rate, it is possible to overwhelm either the garbage collector, or the physical limitations of your hardware. For example, at some point, the memory bandwidth necessary to initialize the temporary objects will exceed that provided by the hardware. Say your application fills up the temporary heap ever second. In this case, based on the common speeds of garbage collectors, your application could easily spend over 20% of its time collecting garbage. Is it difficult to fill up the temporary heap once per second? Typical temporary heap sizes run around 128 megabytes. Say your application is a serves a peak of 1000 requests per second, and creates objects of around 50 bytes each. If it creates around 2500 temporaries per request, then this application will spend 20% of its time collecting garbage.

Example: How Easy it is to Create Lots of Temporary Objects A common example of temporaries is parsing and manipulating data coming from the outside world. Identify the temporary objects in the following code.

```
void main(String xy) {
   doWork(xy.substring(0,10), xy.substring(10));
}
void doWork(String x, String y) {
   doRemoteProcedureCall(parse(x));
   doRemoteProcedureCall(parse(y));
}
Date parse(String string) {
   return DataFormat.getInstance().parse(string, new
      ParsePosition(0));
}
void doRemoteProcedureCall(Date date) {
   long timestamp = date.getTime();
   ...
}
```

This code starts in the `main` method by splitting the input string into two substrings. So far, the code has created four objects (one `String` and one character array per substring). Creating these substrings makes it easy to use the `doWork` method, which takes two Strings as input. However, observe that these four objects are not a necessary part of the computation. Indeed, these substrings are eventually used only as input to the `DateFormat parse` method, which has been nicely designed to allow you to avoid this very problem. By passing a `ParsePosition`, one can parse substrings of a string without having to create temporary strings (at the expense of creating temporary `ParsePosition` objects).

12.3 Time-space Tradeoffs

Sometimes it is beneficial to extend, or shorten, the lifetime of an object. For example, if every request creates an object of the same type, with the same, or very similar, fields, then you should consider caching or pooling a single instance of this object. There are four important cases of time-space tradeoffs. The first covers the situation where recomputing attributes, rather than storing them, is a better choice. The next three cover situations where spending memory to extend the lifetime of certain objects saves sufficient time to be worthwhile: caches, sharing pools, and resource pools. Chapter 11 discusses implementation strategies for these situations.

The example server caches data from an expensive third-party data source. Caching external data in the Java heap complicates your programming and management tasks. The cache must be configured properly so that its contents live long enough to be amortize their costs of fetching, while not occupying too much of the heap. If caches are sized too large, this would leave little space for the temporaries that your application creates. Figure 9.3 shows an example where the cache has

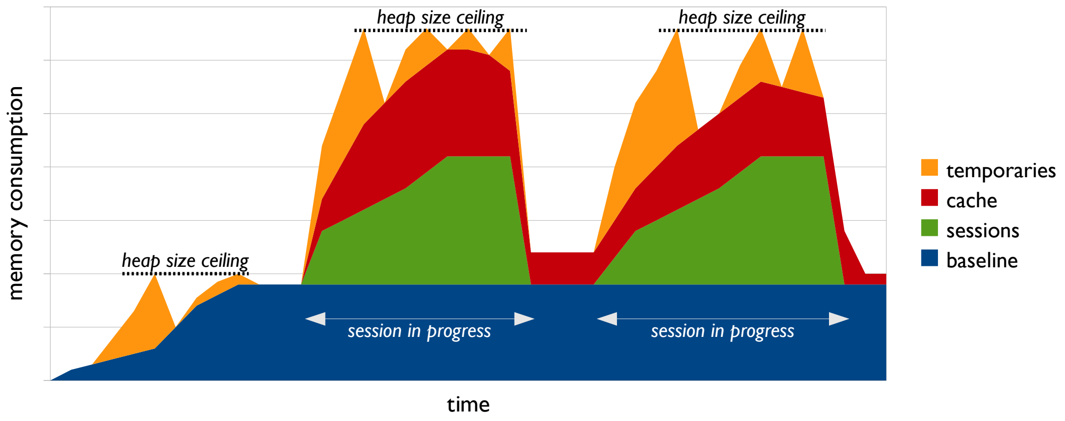


Figure 12.3. When a cache is in use, this leaves less headroom for temporary object allocation, often resulting in more frequent garbage collections.

probably been configured to occupy too much heap space. Observe how, compared to the other timeline figures, there is little headroom for temporary objects. The result is more frequent garbage collections. If the cache were sized to occupy an even greater amount of heap space, it is possible that there would no longer be room to fit session data. The result in this case would be failures in client requests.

Sizing caches is important, but tricky to get right. If the data to be cached is stored on a local disk, then another strategy to caching is to use *memory mapping*. Section 12.5.1 describes how to utilize built-in Java functionality that lets you take advantage of the underlying operating system's demand paging functionality to take care of caching for you.

12.4 Correlated Lifetimes

The catalog data should last forever, while the session data lives for some bounded period of time. It is possible that session state will live beyond the end of your session, but nonetheless it has a lifetime that is bounded. If, due to an bug, part of this session state is not reclaimed, the application will leak memory. Though it is supposed to have a bounded lifetime, it accidentally lives forever. In this case, over time, the amount of heap required for the application to run will increase without bound. Figure 9.4 illustrates this situation, in the extreme case when all of session state leaks. Over time, the area under the curve steps higher and higher.

Correlated Lifetime in Practice Many objects are needed for a bounded interval of time. In some cases, this interval is bounded by the lifetime of another object. In a second important scenario, the lifetime of an object is bounded by the duration of a method call. Once that other object is not needed, or once that method returns, then these *correlated* objects are also no longer needed. These are the two important cases of objects with correlated lifetime.

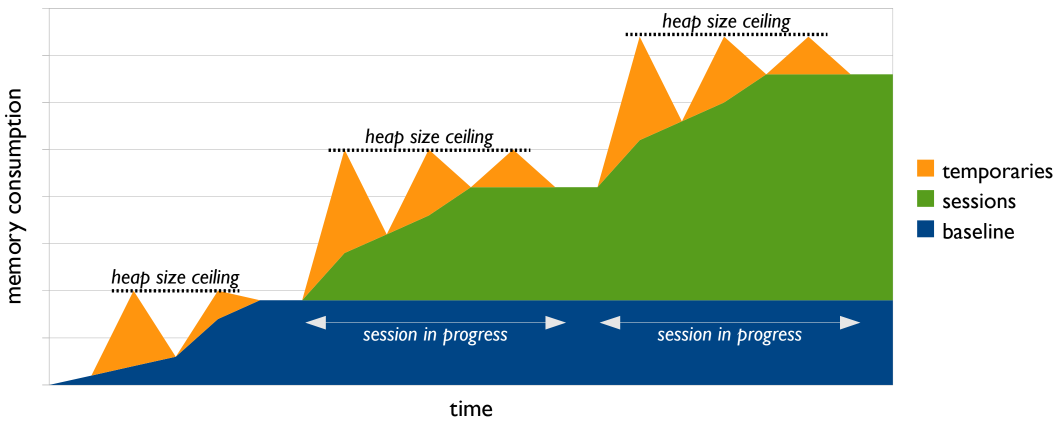


Figure 12.4. If session state is not cleaned up properly, a memory leak is the result. This means more frequent garbage collections, and ever increasing heap size.

Objects that Live and Die Together If you needed to augment the state stored in instances of a class that you are responsible for, you would modify the source code of that class. For example, to add a secondary mailing address to a `Person` model, you add a field to that class and update the initialization and marshalling logic accordingly. This works fine for classes that you own, and when most `Person` instances have a secondary mailing address. Sometimes, you will find it necessary to associate information with an object that is, for one reason or the other, locked down, or where the attributes are only sparsely associated with the related objects.

Example: Annotations In order to debug a performance problem, you need to associate a timestamp with another object. Unfortunately, you don't have access to the source code for that object's class. Where do you keep the new information, and how can you link the associated storage to the main objects without introducing memory leaks?

If you can't modify the class definition for that object, then you will have to store the extra information elsewhere. These *side annotations* will be objects themselves, and you need to make sure that their lifetimes are correlated with the main objects. When one dies, the other should, too.

Objects that Live and Die with Program Phases Similar to the way the lifetime of an object can be correlated with another object, lifetimes are often correlated with method invocations. When a method returns, objects correlated with it should go away. For temporary objects, this is usually easy to ensure, since they are usually only reachable from stack locations. For the medium-to-long running methods that implement the core functionalities of the program, this correlation is harder to get right.

For example, if your application loads a log file from disk, parses it, and then displays the results to the user, it has roughly three phases for this activity. Most of the objects allocated in one phase are scoped to that phase; they are needed to implement the logic of that phase, but not subsequent phases. The phase that loads the log file is likely to maintain maps that help to cross reference different parts of the log file. These are necessary to facilitate parsing, but, once the log file has been loaded, these maps can be discarded. In this way, these maps live and die with the first phase of this example program. If they don't, because the machinery you have set up to govern their lifetimes has bugs, then your application has a memory leak.

This lifetime scenario is also common if your application is an server that handles web requests.

Example: Memory Leaks in an Application Server A web application server handles servlet requests. How is it possible that objects allocated in one request would unintentionally survive beyond the end of the request?

In server applications, most objects created within the scope of a request should not survive the request. Most of these *request-scoped* objects are not used by the application after the request has completed. In the absence of application or framework bugs, they will be collected as soon as is convenient for the runtime. In this example, the lifetime of objects during a request are *correlated* with a method invocation: when the servlet `doGet` or `doPut` (etc.) invocations return, those correlated objects had better be garbage collectible.

There are many program bugs and configuration missteps that can lead to problems. The general problem is that a reference to an object stays around indefinitely, but becomes *forgotten*, and hence rendered unfindable by the normal application logic. If this request-scoped data structure were only reachable from stack locations, you would be fine. Therefore, a request-scoped object will leak only when there exist references from some data structure that lives forever. Here are some common ways that this happens.

- Registrars, where objects are registered as listeners to some service, but not deregistered at the end of a request.

- Doubly-indexed registrars. Here the outer map provides a key to index into the inner map. A leak occurs when the outer key is mistakenly overwritten mid-request. This can happen if the namespace of keys isn't canonical and two development groups use keys that collide. It can also happen if there is a mistaken notion, between two development groups, of who owns respnsibility of populating this registrar.

- Misimplemented hashcode or equals, which foils the retrieval of an object from a hash-based collection. If developers checked the return value of the `remove` method, which for the standard collections would indicate a failure to remove, then this bug could be easily detected early; but developers tend not to do this.

The next chapter goes into greater detail on how to avoid these kinds of errors. Appendix A describes tooling that can help you detect and fix the bugs that make it into your finished application.

12.5 Correlated with External Event

After the server is warmed up, it begins to process client requests. Imagine interacting with a commerce site with a web browser. First you browse around, looking for items that you like, and add them to your shopping cart. Eventually, you may authenticate and complete a purchase. As you browse and buy, the server maintains some state, to remember aspects of what you have done so far. For example, the server stores the incremental state of multi-step transactions, those that span multiple page views. This session state, at least the part of it stored in the Java heap, will go away soon after your browsing session is complete. In the timeline figure, this portion of memory is labeled *sessions*. It ramps up while a session is in progress, and then, in the example illustrated here, soon all of that session memory should be reclaimed.

These session state objects need to be kept around for operations that span several independent operations, and are possibly used across multiple threads. They are used beyond the scope of a phase, are not correlated with another object, but don't live forever.

12.6 Permanently Resident

Figure 9.2 shows the timeline of memory consumption of our example server during and shortly after its initial startup. During the startup interval, the server preloads catalog data into the Java heap. Then, the server is warmed with with two test requests. The total height of the area under the curves represents the memory consumption at that point in time. The preloaded catalog data will be used for the entire duration of the server process. Therefore, the Java objects that represent this catalog are objects that are needed forever. In the timeline picture, this data is respresented by the lowest area, labeled *baseline*. Notice how it ramps up quickly, and then, after the server has reached a "warmed up" state, memory consumption of this baseline data evens out on a plateau for the remainder of the run.

Permanently Resident Objects in Practice In the above example, a `DateFormat` object was created in every loop iteration and used only once. We can improve this situation by creating and using a single formatter for the duration of the run. The Java API documentation, in writing at least, encourages this behavior, but leaves the burden of doing so on you. You must be careful to remember that it is not safe to do so in multiple threads. The next chapter will discuss remedies to this problem. The updated code for the `parse` method would be:

```
static final DateFormat fmt = new DateFormat.getInstance();

Date parse(String string) {
   return fmt.parse(string, new ParsePosition(0));
}
```

There are other cases where your application routinely accesses data structures for the duration of the program's run. For example, if your application loads in trace information from a file and visualizes it, then the data models for the trace data cannot be optimized away entirely. Sometimes it is possible, but not practical from a performance perspective, to reload this data. You could architect your program so that subsets of the trace data are re-parsed as they are needed. Unfortunately, the resulting performance, or the complexity of the code, may suffer drastically. If so, these data structures must, for practical purposes, reside permanently in the heap.

When Objects Don't Fit Sometimes, despite your best efforts at tuning these long-lived data structures, the structures still don't fit within your given Java heap constraints. Chapter 12 discusses strategies for coping with this situation.

Chapter 13

MEMORY FUNDAMENTALS

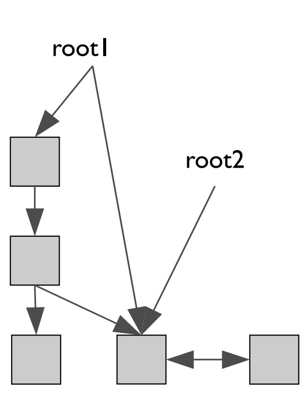
The Java language has a *managed runtime*. In part, this means that, as a Java program runs, the Java runtime takes on the burden of key aspects of memory allocation and reclamation. It is important to understand what Java does for you, in forming its decision of when an object should be reclaimed, when devising your lifetime management strategy. This level of management includes as automatic garbage collection of both instances and Java classes. Therefore, Java has feature that govern, on your behalf, important aspects of the lifetime of objects. Unfortunately, these features often appear in the form of low-level JVM hooks, or implicit behavior that you have to implicitly control, and so require careful coding to make correct use of them. You need to appreciate what the runtime is doing for you, before considering how to reshape object lifetimes to better suit your needs.

This chapter introduces the basics of the garbage collector, and how the Java managed runtime governs object lifetimes. Then, it walks you through the lifecycle of typical objects from allocation to eventual garbage collection. If you are comfortable with the basics of memory management, you may discover that you can skip to the next chapter.

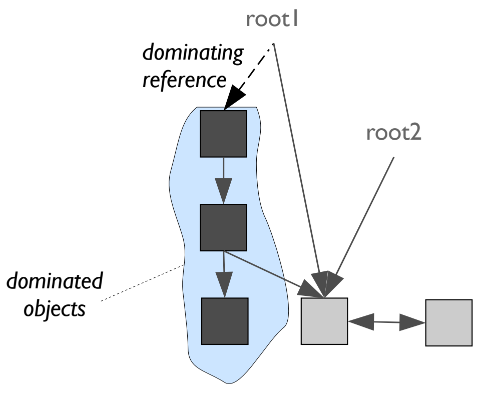
13.1 The Garbage Collector

The garbage collector is the mechanism for determining when an object should be reclaimed. It is governed by a number of configuration choices that you can make. These configuration choices guide the schedule that the collector follows. This includes, for example, the frequency of collection and the lengths of pauses your application experiences. In any case, the collector will obey some basic principles, dictated by how your data structures are interconnected, when determining *what* to collect.

What a Collection Collects: Reachability and Unique Ownership Each time a garbage collection occurs, the collector inspects the heap for possibly *live* objects. The collector treats the heap as a graph of objects. The nodes are the objects themselves, and the edges are field or array slots that result in a pointer from one object to another. Recursively, an object is live either if it is a referenced either by a live object, or, in the base case, by a *root*. The roots of garbage collection are those that



(a) A live data structure.



(b) Then, the dominating reference is clipped.

Figure 13.1. The garbage collector uses the references between objects to determine what objects are collectable. The data structure on the left is filled entirely with live objects. The one on the right, after a link is clipped, now contains some collectable objects; all objects reachable only by traversing a *dominating reference*, i.e. those *dominated objects*, will be collectable, as well.

come from outside the Java space, and include: objects serving as monitors, objects on the stack of a method invocation in progress, and references from native code via the Java Native Interface (JNI). Every other object not live, in this sense, are ready for collection; they are garbage.

This recursive aspect can also be expressed in terms of *reachability*. The live objects are those objects reachable, by following a chain of references, from some root. Figure 10.1 illustrates a simple data structure, and shows which part becomes collectable when a reference is set to null, or "clipped". When the indicated reference is clipped, there is no chain of references from a root to the shaded region of objects.

Reachability is the graph property that determines what objects are still live. This is all the garbage collector cares about, finding the objects that need to be kept around. It is also helpful for programmers to know which objects become dead as the result of a pointer being clipped. The objects within the shaded region of Figure 10.1(b) have the property that each is reachable *only* from the clipped reference. That clipped reference is the unique owner of the shaded objects. The graph property that describes unique ownership is called *dominance*; the clipped reference is said to dominate those objects that it uniquely owns.

The Collection Schedule and Safe Points The garbage collector does not reclaim memory immediately after an object's last use. Instead, to amortize the costs involved in reclamation, the garbage collector often lets reclaimable objects pile up for a while, and reclaims memory in bulk. This bulk operation or reclaiming unused memory is usually implemented as a number of threads. These worker threads, on some schedule, wake up and traverse the heap for live objects. As objects are allocated, memory consumption can be observed to increase, up until some maximum allowed amount. At this point, the collector reclaims unused memory, and the process starts again. In this way, memory consumption over time often assumes a sawtooth edge, such as those shown in Figure 9.3.

GC Safe Points In most production JREs, garbage collection is, in normal execution, not run at arbitrary points in the code. For example, in the above method `Foo.bar`, even though the object referred to by `localVariableReference1` becomes unreachable before the end of the invocation, most JREs will not notice this until a period of time after the assignment to `null` at line 6. This delay comes about because the garbage collector typically only runs when threads reach certain *safe points* in the code. Safe points commonly include the beginning or end of method invocations, the end of each loop iteration, and points surrounding native method invocations. Therefore, the earliest time at which the object referred to by `localVariableReference1` could be reclaimed is after the first iteration of the loop; it could even possibly be the end of the invocation, if the loop iterates zero times.

This is not to say that the garbage collector runs at every safe point, or that it waits for all threads to reach a safe point before proceeding. Any thread that tries, but fails, to allocate a new object will of course result in a garbage collection at whatever line of code that allocation is found. At that point in time, the other threads will continue executing up until their next safe point, or their own failure to allocate memory, at which point garbage collection can proceed.

In most production JREs, garbage collection is, in normal execution, not run at arbitrary points in the code. For example, in the above method `Foo.bar`, even though the object referred to by `localVariableReference1` becomes unreachable before the end of the invocation, most JREs will not notice this until a period of time after the assignment to `null` at line 6. This delay comes about because the garbage collector typically only runs when threads reach certain *safe points* in the code. Safe points commonly include the beginning or end of method invocations, the end of each loop iteration, and points surrounding native method invocations. Therefore, the earliest time at which the object referred to by `localVariableReference1` could be reclaimed is after the first iteration of the loop; it could even possibly be the end of the invocation, if the loop iterates zero times.

This is not to say that the garbage collector runs at every safe point, or that it waits for all threads to reach a safe point before proceeding. Any thread that tries, but fails, to allocate a new object will of course result in a garbage collection at whatever line of code that allocation is found. At that point in time, the other threads will continue executing up until their next safe point, or their own failure to allocate memory, at which point garbage collection can proceed.

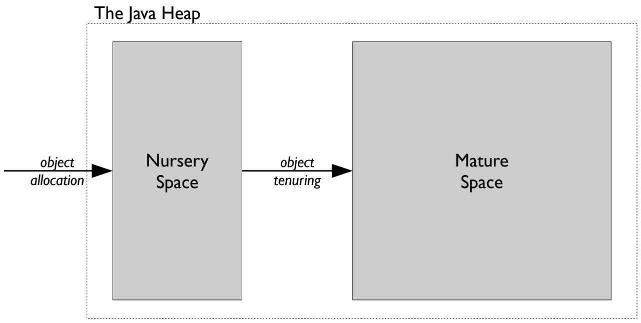


Figure 13.2. Oftentimes, the Java heap is split into two sub-heaps. The nursery space stores newly-allocated objects, and the mature space stores objects that have been tenured. The garbage collector tenures an object after it survives a sufficient number of nursery garbage collections.

Configuration Settings You can guide the frequency of collection, which will change the slope of this sawtooth curve to be either more or less jagged. In one common case, the garbage collector will wait until all available memory is consumed before reclaiming storage. In Java, you can configure this ceiling by supplying a sizing to the `-Xms` (initial ceiling) and `-Xmx` (maximum ceiling) command line options. The JRE will begin with a ceiling at the former level. If collections are occuring too frequently, the JRE may decide to increase the *current* ceiling to a higher level. As the need for memory fluctuates, so the JRE will raise or lower the current ceiling level. The current ceiling will always be some value lower than the maximum, `-Xmx`, setting. One such scenario, of increasing ceiling level, is illustrated in Figure 9.4.

The Nursery and Mature Heaps To optimize for applications that create a large number of temporary objects, some garbage collection strategies attempt to separate the short-lived and long-lived objects into two separate heaps. These two heaps are typically called the *nursery* and *mature* spaces, as illustrated in Figure 10.2. The usual behavior is for objects to be allocated in the nursery and, if they survive a sufficient number of garbage collection cycles, to be *tenured* to the mature space. If a large majority of the objects in the nursery are no longer live at the time the JRE runs a garbage collection on the nursery heap, then a traversal of the nursery help will only touch a small number of memory pages. In this way, ignoring the costs of initialization, reclaiming objects that are short-lived can be very cheap. Some collectors allow you to specify a separate maxmum size for each, and some let you specify only the maximum nursery size and the total maximum heap consumption (via `-Xmx`.)

The Permspace Heap In addition to separating new and old objects, some JREs create a third heap in which to store data that is very unlikely to ever become garbage. This includes the JREs metadata for your Java classes, along with the executable code for your methods. In addition, any strings that you have interned will be stored in this heap, along with any objects that the source code compiler has decided to store in the *constant pool* for a class; these objects include any static strings, such as the one in this code snippet:

```
System.out.println("aStaticString");
```

The maximum size of Permspace, like the other heaps in Java, can often be specified on the command line. In some cases, you may find that your application requires a suspciously large maximum size for Permspace.

Example: Class Duplication and Excessive Permspace A Java Enterprise Edition (JEE) server application is deployed as 100 separate applications, each in its own Web Application Archive (termed `war`) file. Each `war` file contains duplicate class files for logic common to some, or even all applications. The development team didn't think to worry about this, figuring that the JRE would notice and remove the duplication. They were wrong, due to requirements of the JEE specification, and suffered from excessive Permspace consumption.

In JEE, each `war` file represents a distinct application, probably separately developed. Having been coded separately, the JEE model assumes the worst, that the applications will collide in their use of the static fields of classes. Therefore, each `war` is loaded into a separate classloader, with the result that the class duplication is not removed. The server application required 500 megabytes for its Permspace heap, despite having under 100 megabytes of distinct class data.

13.2 How to Free an Object in Java

If you wish an object's storage to be reclaimed, you must take some care. To do so in a language like C is simple, if bug prone. When you call `free` on any pointer to a dynamically allocated memory region, memory is immediately available for subsequent use.[1] A C style of memory management comes with well known risks, and commonly leads to memory errors. For example, memory might be deallocated more than once, or variables that do not point to the start of an allocated region might be passed to `free`. Still, using `free` is straightforward to use, and immediate in effect.

In Java, you are immune from these problems, but there is no way to return memory for immediately use in future allocations. Instead, you indicate, either implicitly or explicitly, that objects are no longer needed. The explicit mechanism is to set references to null. To do so implicitly, there are several devices at your disposal. This section describes the details of both.

It is important to remember that in contrast to C, all means of indicating an object is no longer needed have a certain degree of delay. There are delays of two

[1]It is possible to run a C program with a special `malloc` library that introduces a simplified form of garbage collection.

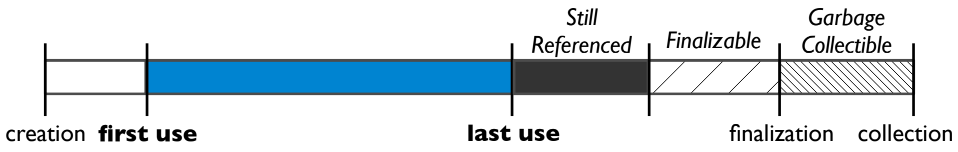


Figure 13.3. Timeline of the life of a typical object.

sorts. First, if you rely on the implicit mechanisms for indicating an object is no longer needing, there is often a delay from its last use to this point. Second, there is a delay from the point at which you indicate an object is no longer needed until its storage is reclaimed.

The Lifecycle of an Object These two delays are but a part of the larger lifecycle that every Java object goes through, from creation to reclamation. In a well-behaved application, an object's lifetime spans its allocation, use, and the short period during which the JRE takes control and reclaims the space. For some subset of an object's actual lifetime, that is the time from creation to reclamation, your application will make use of the data stored in its fields. Figure 10.3 illustrates the lifecycle of a typical object in a well behaved application.

Example: Parsing a Date Consider a loop that shows an easy way to parse a list of dates. What objects are created, and what are their lifetimes?

```
for (String string : inputList) {
    ParsePosition pos = new ParsePosition(0);
    SimpleDateFormat parser = new SimpleDateFormat();
    System.out.println(parser.parse(string, pos));
}
```

For each iteration of this loop, this code takes a date that is represented as a string and produces a standard Java `Date` object. In doing so, a number of objects are created. Two of these are easy to see, in the two `new` calls that create the parse position and date parser objects. The programmer who wrote this created two objects, but many more are created by the standard libraries to finish the task. These include a calendar object, number of arrays, and the `Date` itself. None of these objects are used beyond the iteration of the loop in which they were created. Within one iteration, they are created, almost immediately used, and then enter a state of drag.

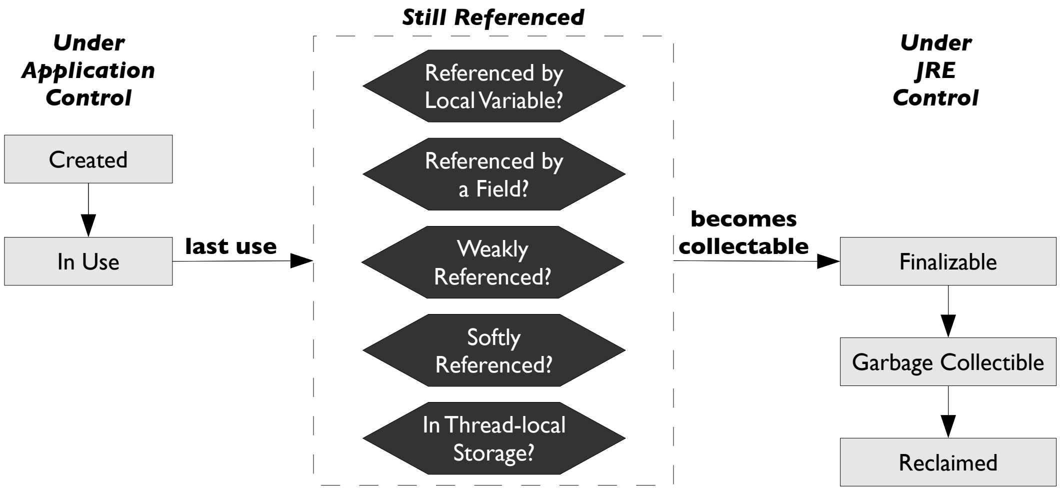


Figure 13.4. After its last use, an object enters a kind of limbo: the application is done with it, but it is not yet a candidate for garbage collection. When an object exits this limbo depends on the way it is referenced by your application.

Memory Drag

At some point, an object will never be used again, but the JRE doesn't yet know that this is the case. The object hangs around, taking up space in the Java heap until the point when some action is taken, either by the JRE or by the application itself, to make the object a candidate for reclamation. The interval of time between its last use and ultimate reclamation is refered to as *drag*.

The `pos` object represents to the parser the position within the input string to begin parsing. The implementation of the `parse` method uses it early on in the process of parsing. Despite being unused for the remainder of the parsing, the JRE does not know this until the current iteration of the loop has finished. For this duration of time the object is in a kind of limbo, where it is referenced but never be used again. This limbo time also includes the entirety of the call to `System.out.println`, an operation entirely unrelated to the creation or use of the parse position object. Once the current loop iteration finishes, these two objects will become candidates for garbage collection.

Depending on how the object is referenced by your application, it will transition from in-use to garbage-collectable in a different manner. There are eight ways an object may be referenced. It can be referenced by a:

1. local variable of a method

2. static field of a class

3. instance field of a `java.util.ref.WeakReference` object
4. instance field of a `java.util.ref.SoftReference` object
5. instance field of a `java.util.ref.PhantomReference` object
6. thread-local storage
7. instance field of any other object
8. some combination of the above

The first seven are cases of unique ownership of an object. There exists only one path, via that particular reference, to reach the object. The eight case is one of *shared* ownership. After your program creates an object, it references the object in one or more of these eight ways. The ownership of an object may vary over time — as your program runs, these references will come and go. After some time, program execution may reach a point where the object is no longer under application control. At this point, the JRE is now in charge of its lifetime. For example, if the only reference to an object is from a field of some other object, and your code reassigns that reference to `null`, this becomes a point of transition, from application control to JRE control. Each of the eight ways of referencing an object comes with its own guidelines as to when this transition occurs. Figure 10.4 and Table 10.1 summarize how an object transitions to JRE control. The following code gives examples of these eight patterns of ownership:

```
class Foo {
   void bar(Object argument) {
      Object localVariableReference1 = new ...;
      threadLocal1.set(argument);
      threadLocal2.set(new ...);
      localVariableReference1 = null;
      for (...) {
         Object localVariableReference2 = new ...;
         ...
      }
   }

   static Object staticField = new ...;
   Object instanceField = new ...;
   Reference weak = new WeakReference(instanceField);
   Reference soft = new SoftReference(instanceField);
   Reference phantom = new PhantomReference(instanceField);
   ThreadLocal threadLocal1 = new ThreadLocal();
   ThreadLocal threadLocal2 = new ThreadLocal();
}
```

uniquely owned by	when object becomes candidate for reclamation
nothing	immediately
local variable	after scope exits, e.g. method returns
instance field of an object	when that object becomes reclamation candidate
static field of a class	when that class is unloaded
field of `WeakReference`	immediately
...with `ReferenceQueue`	immediately placed on queue, then after queue is polled
field of `SoftReference`	approximately LRU
...with `ReferenceQueue`	after LRU, placed on queue, then after queue is polled
entry in thread-local storage	when that thread dies

Table 13.1. When an object becomes a candidate for reclamation. The dominating reference can be explicitly overwritten, e.g. by your code expliclty assigning the reference to `null`. Otherwise, an object only automatically becomes a candidate under the restricted circumstances shown here.

The Lifetime of Local Variables Variables that are declared within a method body often have a lifetime that is bound to, at most, the duration of an invocation of that method. Common examples of this are local variables, loop variables, and variables declared within some inner scope such as within the body of a loop or `if` statement. For these variables, when a loop continues to the next iteration as in the case of `localVariableReference2`, when the body of a clause of an if/then/else statement finishes, or when the method invocation returns as in the case of `localVariableReference1`, there is a good chance that the object referenced by that variable will be reclaimable.

There are situations where an object may *escape* the local scope in which it was declared. In these cases, the object has *shared ownership* until this local scope exits. See below for further discussion of shared ownership. This is an example of an object escaping a local variable scope, so that it is, for the duration of that local scope, also owned by a static field of a class object:

```
class Foo {
   static Object static_obj;

   void foo() {
      Object obj = new Object();
      static_obj = obj;
      ... // uses of obj
   } // obj scope exits, but static_obj ownership persists
}
```

The next section discusses the lifetime of static fields.

The minimum that the Java language specification requires is that non-escaping objects that are declared within some scope inside of a method will be reclaimable by the time that scope exits. Many modern JREs try to optimize by inferring, when they can, the line of code at which an object will certainly never be used again. For this reason, some (but not all!) sources of memory drag that would have been a problem with older JREs, are no longer an issue. For example, you can run this test program and, by inspecting the output, determine whether your JRE is performing this kind of optimization:

```
/* Test for whether the optimizer detects that obj is
    reclaimable before the end of this method */
static public void main(String[] args) {
   Object obj = new Object() {
      protected void finalize() {
         System.out.println("Finalized");
      }
   };
   System.out.println("StartOfLoop");
   for (int i = 0; i < 1000000; i++) {
       new HashSet().add(i);
   }
   System.out.println("EndOfLoop");
}
```

If you see the `Finalized` message before the end of loop message, this is a sure sign that the JRE is being clever. Be careful to remember that seeing the finalized message is definitive evidence, but not seeing that message might only mean that your loop doesn't iterate enough times to cause a garbage collection to occur. You may need to experiment with the number of loop iterations, before coming to a conclusion.

The Lifetime of Statics, and Class Unloading The JRE allocates memory for every class, to store its static fields, such as the one on line 13 in the above example. This memory, to store all static fields plus some bookkeeping information, is often referred to as the *class object* for the class. It is possible for the same class to be loaded into multiple class loaders; in this way, using more than one class loader lets you avoid the problem of colliding use of static fields in separately developed parts of the code. A static field therefore only exits scope when the class object is reclaimed, which occurs when the respective class is unloaded, by the JRE, from its class loader.

If a class is never unloaded, which is likely to the be case for your application, then that class object will remain permanently resident. The *default* class loader, which is the one that will be used unless you specify otherwise, never unloads application classes.

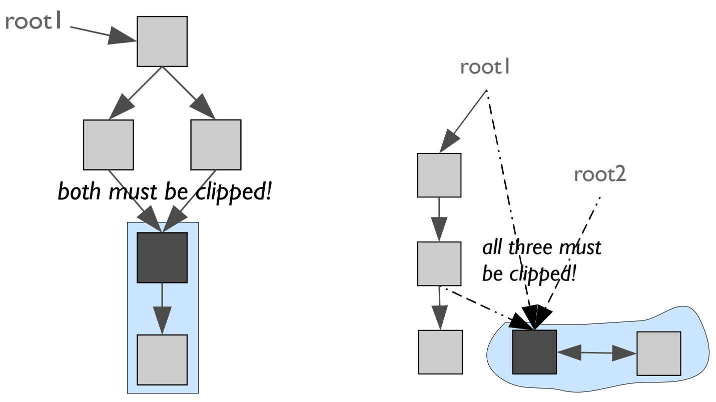
If you need classes to be unloaded, then you must manually specify a class loader to use. Unloading a class is then accomplished by ensuring that all references to both the class object and your custom class loader, are set to `null`. This will render the class unloadable, and will also render objects referenced by static fields all classes loaded into that class loader as garbage collectable. There exist module management systems, such as OSGi [?], that facilitate this task.

Due to these complexities, your design should generally anticipate that the memory for these static fields is permanently resident. This means that any static fields referencing an instance, rather than containing primitive data, will render that instance also permanently resident. Unless, that is, you take action to explicitly clip the static field reference, by assigning the field to null. Otherwise, that instance will be forever reachable along a path from some garbage collection root through the static field reference. In this way, storing a reference in a `static` field of an class is one way to implement a permanently resident lifetime policy.

The Lifetime of Instance Field References As long as an object `B` is dominated by an instance field of another object `A`, then these two objects will live and die together. The only device at your disposal to make `B` garbage collectable before `A` is to break all paths of references from `A` to `B`. In simple cases, where `B` is directly referenced by a field of `A`, then reassigning that reference to `null` will do the trick. After doing so, the `B` object will now be a candiate for garbage collection.

Shared Ownership When you invoke a library method, there is no way in Java to know what the called method does with your object. It could very well squirrel away a reference to any object reachable from arguments you pass to the invocation. Despite your best efforts at keeping track of which references exist to an object, it can easily become an uncontrolled mess once you pass these objects to third-party libraries. In the above example, if you call the `parse` method of a `SimpleDateFormat` object, the method contract says nothing about how it treats the given string or `ParsePosition` passed as parameters. Consider the case where you need the string to become garbage collectable soon after having parsed it, but the formatter maintains a reference in order to avoid reparsing the same string in back to back calls. This calls to mind the worst of the days of explicitly managing memory in a language like C.

In the case where there is more than one reference to the object, the story gets more complicated. In contrast to C, where a `free` of *any* pointer suffices for deallocation, in Java *all* references to an object must be assigned to `null`. This is tricky in many cases, because it may not be easy to know where all those references emanate from. Figure 10.5 illustrates a situation where three references must be clipped before an object, the darkly shaded one, becomes a candidate for garbage collection. There are two other important things to note in this example. First, just



(a) Diamond sharing. (b) Root sharing.

Figure 13.5. When an object is shared, such as the shaded ones shown here, care must be taken to clip all edges from emanating from outside of the region you wish to reclaim.

as in Figure 10.1(b), after clipping the three indicated references, an entire data structure, not just that darkly shaded object, becomes a candidate for reclamation. This structure consists of the two objects contained within the lightly shaded region. The second important thing to note is that you needn't clip the backwards edge, or any edge contained entirely within the data structure you no longer need.

13.3 Advanced Java Lifetime Management Features

There are several important lifetime management policies that cannot be expressed using normal mechanisms of local variables and static and instance field references. For example, to implement the correlated lifetime policy described in **??** using only the mechanisms presented so far is difficult. In Figure 10.1(b), there are three objects contained within the shaded region; these three objects will all be garbage collectible if the indicated dominating reference is set to `null`. In this way, the lifetime of the lower two can be correlated with the lifetime of the object directly pointed to by the dominating reference. This can work, in certain limited circumstances, but it requires that you encode the correlation in the class definition itself. To correlate an instance of the `B` class with an instance of the `A` class, you must add a field to `A`:

```
class A {
  B b;
}
```

In addition to requiring the pollution of class definitions, it also can result in wasted memory. If only some `A` instances have a correlated `B`, then you will be wasting a pointer slot for every instance of `A`. It turns out that this is how the Java standard `HashMap` was implemented, in its mechanism for maintainin a correlation between the various views (the key set, value set, and entry set), and the map itself:

```
class HashMap {
   Set keySet, valueSet, entrySet;
}
```

This choice had a big implication on the memory bloat factor of small maps. As we have learned in previous chapters, every instance of a standard Java hash map must pay the expense for potential use of features. These features aren't always employed, and certain rarely together, for a single map instance.

The Java language provides mechanisms that allow you more flexibility in implementing lifetime management policies. These advanced features are exposed via the `java.lang.ref.Reference` family of classes and thread-local storage. Programmers very often confuse these mechanisms, and there is quite a bit of disinformation on the Internet; there is especially confusion over the use of soft and weak references. Therefore, it is important for you to take some care in understanding the low-level features.

To contrast the normal referencing mechanisms with these advanced features, the normal ways of referencing objects are typically referred to as *strong* references. This term came about to contrast with the terms used for the more advanced features: weak, soft, and phantom references.

Weak References (referencing an object without inhibiting its reclaimability) The constructor for a `WeakReference` takes an object as a parameter. The resulting instance of `WeakReference` is said to *weakly reference* that other object. A weakly referenced object will be garbage collectable, or not, independently of being referenced from the `WeakReference` object; the weak reference provides a non-owning handle to another object. This is a sometimes helpful combination, to be able to refer to the object, without inhibiting it being reclaimed as it normally would. To tunnel through the level of indirection introduced by the weak reference, you call its `get` method. If this method returns `null`, then you know that the referenced object has been reclaimed. **??** discusses how to leverage this functionality for an alternative implementation of the correlated lifetime pattern.

There are three issues that you must handle with care. First, even though the weak reference itself does not prolong the lifetime of the referenced object, once you call `get`, you now have a regular reference to that object. If this reference is stored in a local variable, or in a field of another object, or in a collection, then you have now altered its lifetime. Once normally referenced, the underlying object now obeys the lifetime rules discussed earlier. Second, you must be careful to avoid race conditions, such as:

```
class A {
   WeakReference<B> b;

   B getB() {
      if (b.get() != null) {
         return b.get();
      } else {
         // perform any necessary cleanup
         // notify caller of b's reclamation
      }
   }
}
```

In that code, between the first call to `b.get()` and the second, the underlying object may have been reclaimed. You must modify that code to call `b.get()` only once, and stash the result in a local variable until `getB` returns. Third, if you need to be informed that the underlying object has been reclaimed, you must use the *reference queue* mechanism, which will be discussed shortly.

Soft References The constructor for a `SoftReference` also takes an object as a parameter. The object lives as it normally would until the point in time when there are no other strong references to the object. When an object is only softly or weakly referenced, then it enters a special transitionary lifetime state. The JRE will keep this object around, for as long as there is enough free Java heap. Once heap grows tight, the JRE will begin treating the soft references as if they were weak references — the soft references that the JRE choses to discard will no longer inhibit the reclaimability of the softly referenced objects. This discarding of soft references is sometimes referred to as "clearing" soft references.

The Java language specification makes no specific requirements as to how JRE implementations should chose which soft references to discard. The only requirement imposed by the language specification is that the JRE must have cleared *all* soft references before it throws an `OutOfMemoryException`. That is, it must have reclaimed all objects that are uniquely owned by soft (or both soft and weak) references before it gives up, and fails due to heap exhaustion. Early JREs tended to make poor decisions, when choosing how to clear soft references. One JRE would wait until the heap was exhausted, at which point it would clear all soft references. Most 1.5 and 1.6 JREs use a more sophisticated least recently used (LRU) heuristic. They keep track of the last time `get` was called, on a per-`SoftReference` basis, and begin to clear soft references if their last use was long ago. Sometimes, they measure this distance relative to the rate of object allocation; this modified heuristic will not clear soft references if your application isn't allocating objects at a high rate.

For those JREs that use some sort of LRU heuristic, soft references can form the basis for implementing caches. You must be careful not to depend on this heuristic blindly. You should first run an experiment against the JRE to which you intend to deploy: implement a simple cache with soft references (see **??**); enable verbose garbage collection, and observe the messages that indicate soft references being cleared. Making this observation is easier on an IBM than a Sun JVM. To do so on an IBM JVM, enable verbose garbage collection statistics (by adding `-verbose:gc` to the command line), and track lines of output of this form:

```
<refs soft="27801" weak="3" phantom="0"
   dynamicSoftReferenceThreshold="19"
   maxSoftReferenceThreshold="32" />
```

It is the number of soft references you need to track. With an LRU-based soft reference clearing heuristic, you should observe that clearing occurs at a constant rate. If you observe lulls and spikes in clearing, then you must not depend on soft references for implementing your caches!

The Reference Condundrum If you choose to leverage weak and soft references, you are in for a treat of complex programming. There is a complex programming hassle that stems from `WeakReference` and `SoftReference` being normal Java objects. If a weak reference does not prevent an object from being reclaimable, and the weak reference itself is represented by a Java object, then what is to keep the `WeakReference` object itself from being reclaimed? The same thing holds for soft references.

The only way to prevent a weak reference object from being immediately reclaimed is to reference them, somehow, with strong references. If your goal is to connect one object with another, via a weak or soft reference, then your job can be straightforward. The above code, with `A.getB()`, works pretty well, as long as it is properly modified to avoid the race condition. The only room for improvement is unnecessary carrying around the `WeakReference` object itself for the lifetime of the `A` instance, even after the weakly referenced `B` has been reclaimed.

This baggage, of the reference wrappers, runs the risk of being a major contributor to the overhead of using the weak or soft referencing mechanism. The cost of a `SoftReference` wrapper is typically 12 bytes for the object header, 4 bytes for the pointer to the referent, and 16 bytes for the clocks necessary to implement an LRU eviction; it turns out that, to facilitate the interaction with the JVM, every reference requires an extra 3 pointers, or 12 bytes, on top of these costs. In total, then, every soft reference you use in your program costs at least 48 bytes. If the `A` object consumes only 24 bytes on its own, then softly referencing a `B` instance triples the unit ost of `A`. A weak reference wrapper saves those 16 bytes of clocks, and so costs a still high 32 bytes per wrapper.

The overhead grows even higher when you wish to associate a `B` with an `A`, but cannot modify the class definition of `A`. In this case, you must introduce a map in which to store the relation between the two. Worse, however, is that this construct now leaks memory. When the `B` objects are reclaimed, the map will still hold a strong reference to the reference object that was serving as a wrapper around that reclaimed `B`.

The Basics of Reference Queue Management Java provides *reference queues* as a way to avoid this extra baggage, and to avoid memory leaks in the use of weak and soft references. Using reference queues adds an extra layer of complexity to an already difficult programming task, but they are necessary for most uses of weak and soft references. You can construct a reference wrapper with an associated reference queue. If you do so, then the associated object, when the JRE decides to clip the weak or soft reference, will not become immediately reclaimable. Instead, two special things happen. First, the wrapper will be placed on the associated reference queue. Second, once the wrapper is on the queue, the wrapper will change its behavior to no longer reference the referent object. The latter effect complicates the clean up process: the wrapper is placed on the queue, but calls to `get` no longer return the referent object.

The reference queue becomes a way for the JRE to notify you that it is ready to clip the reference. By associating a reference queue with weak and soft references, you are given a cleanup hook. With this hook, you can free up resources that are tied to the association that the weak or soft references has established. For example, you have have native resources tied to the association, such as open file descriptors. You can also clean up the memory consumed by the reference objects themselves, such as by assigning the `WeakReference<B> b` field to null in the example above. Since any calls to `get`, from this hook, will return `null`, either your hooks must not require access to the referent object, or you need to secure some other means of accessing it.

One important task is cleaning up the reference queue itself. If you don't finish the job properly, then the reference queue will become a source of memory leaks. In order to detect that an object has been placed on the queue, your only recourse is to call the `poll` method of the associated reference queue. This will return the reference wrapper that is ready to be clipped, after removing it from the queue. To avoid a memory leak in the reference queue, you must call `poll` at least at the same rate at which you create reference wrappers. That is, just before you create a reference wrapper, you must also poll the reference queue, preferably in a loop, to see if any previously created wrappers are ready to be clipped. This sounds complicated to get right, and it is. Be careful!

If you are directly associating an `A` with a `B`, it is necessary to store the reference queue in a static field; storing it in an instance field of `A` wouldn't make any sense. This means that the calls to poll the reference queue must be protected with critical sections, in order to avoid race conditions. However, this introduces lock contention problems:

```
class A {
   static ReferenceQueue<B> refQueue;
   SoftReference<B> b;

   static void cleanupQueue() {
      synchronized(refQueue) {
         Reference<? extends B> bb;
         while ((bb = queue.poll()) != null) {
            // perform clean up bb
         }
      }
   }
   void makeB(B b) {
      cleanupQueue();
      this.b = new SoftReference<B>(b, queue);
   }
}
```

Section 11.5 discusses solutions to this concurrency problem.

The WeakHashMap The standard library includes an important construct, the `WeakHashMap`, that is not only quite useful, but also hides most of the complexity of managing reference queues. A weak hashmap is a map that only weakly references its keys. When a key is reclaimed, the map evicts the corresponding entry. Behind the scenes, it uses weak references and reference queues, but you needn't worry about any of that. Your own code is not polluted by mention of `WeakReference` and `ReferenceQueue`, nor of the polling complexities necessary to keep the reference queue from overflowing with reclaimed keys.

The Danger of Diamonds Using a construct such as `WeakHashMap` is not a guarantee of success. The danger lies in there being another reference to the key that is not a weak reference. If this strong reference comes from the key data structure itself, then there is no problem. It is expected that, for the expected lifetime of an entry, there will be a strong reference that comes from another data structure — this is the reference that you expect to keep the entry alive. The main danger lies in a second strong reference emanating from the value data structure of a key's entry in the map. If you insert values that strongly reference the key, as illustrated in Figure 10.6 then the key will very likely never be uniquely owned by the weak reference, even after the expected strong reference is reclaimed.

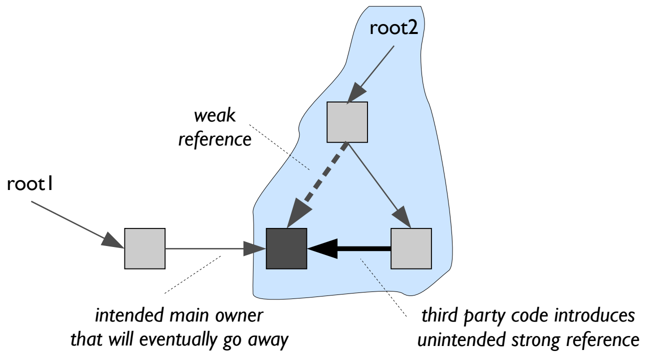

root2
weak reference
root1
intended main owner that will eventually go away
third party code introduces unintended strong reference

Figure 13.6. Despite your best efforts to use weak references correctly, if you introduce a second strong reference in a way that forms a *diamond* shape (the shaded region), it will likely never be reclaimed.

As is shown in the figure, this problematic reference structure has a diamond shape. The top of the diamond is the `Entry` object of the map, from which emanate two paths to the key, the weak reference from the `Entry` to the key, and the strong reference path that flows through the value. In Figure 10.6, the darkly shaded object has a good chance of never being reclaimed.

In the following code, if `loopback` is true, then the `Finalized` message will never appear.

```
static public Object setupCycle(boolean loopback) {
   class Cycle {
      Object loopback;
   }
   class Entry {
      Object key;
      Cycle value;
   }
   Object key = new Object() {
         protected void finalize() {
            System.out.println("Finalized");
         }
      };
   Entry e = new Entry();
   e.key = new WeakReference(key);
   e.value = new Cycle();
   if (loopback) e.value.loopback = key;
   return e;
}
```

Finalization and Phantom References Java provides two other closely related cleanup hooks, in the form of finalization and phantom references. These hooks are called just after the garbage collector has discovered that the object is collectible, but before its memory is reclaimed. If you implement a `finalize()` method in a class, then every instance of that class will go through a finalization process. After being discovered to be garbage, these instances will be enqueued on a special queue, usually termed the *finalizer queue*. Most JREs spawn a single thread, termed the finalizer thread, that periodically scans the finalizer queue, invoking the `finalize` method on the enqueued objects.

Phantom references offer a somewhat more refined version of this pre-reclamation hook. First, with phantom references, you can associate a cleanup hook on a per-object basis, rather than, as is the case with finalization, on a per-class basis. Second, phantom references give you the option of having more than one cleanup queue and thread, in contrast to finalization where there is a single finalizer queue and (usually) a single finalizer thread.

You can use the hooks offered by finalization or phantom references to free up resources that are implicitly associated with an object. Any Java objects uniquely owned by this object will be reclaimed in the normal course of garbage collection. It is those resources that are *implicitly* tied to a Java object, such as file descriptors, socket connections, and database resources such as compiled queries, that require special attention.

You must be very careful in relying on finalization or phantom references. The Java language specification provides no assurances of how often, or even whether, finalization will be run on an object. In the normal course of program execution, eventually the finalizer will run. This is because the finalizer queue consumes Java heap, and hence the finalizer thread will always do whatever finalization is possible before the JRE gives up due to heap exhaustion. However, if your Java objects serve as proxies for some native, or remote, storage, and the space consumed by the Java proxies is small compared to the external state, then you may have problems. The JRE knows nothing about this external state, and so will not schedule the finalizer thread if an external resource is exhausted.

Furthermore, the specification is very lax about whether finalizers will be run before program termination. You can ask the JRE to attempt to finalize objects before the program terminates, by calling `System.runFinalizersOnExit(true)`, a deprecated part of the API. However, most JREs these days run only a partial finalization, if you ask. It is hard for the JRE to do the right thing in a deadlock free manner. Should it only schedule the finalizer thread, which would run any pending finalization? This would be safe, and is what the JRE will do if you ask. But this misses all the currently-live objects that would have have been finalized, had the program reached a point where they were reclaimable. The JRE can't unwind, on exit, all the Java references thta keep those objects alive. Also, there is no analogous request to have the JRE run a garbage collection on exit. Hence, even for those objects which are actually ready to be finalized, the JRE won't do so on exit. It is for these reasons that the API has been deprecated.

Given these downsides, it is best for you to implement a more robust lifetime management strategy. If you can establish a correlated lifetime pattern, such as that the external storage should be reclaimed when an event occurs, or when a method returns, then you should do so.

Thread-local Storage The last advanced memory management feature offered by Java is the ability to associate memory with a thread. Thread-local storage provides a way to avoid synchronization costs, often at the expense of some degree of wasted memory. To avoid synchronization, you often need to replicate some data structures. This feature is, of course, only helpful if your program runs with multiple threads.

Consider an example of using the `SecureRandom` class. Instances of this class provide a stream of pseudo-random numbers, in a way that is cryptographically strong. If you have a singleton instance of this class, you may experience scalability problems due to lock contention; the contention is hidden within the `SecureRandom` implementation. You can use thread-local storage to avoid this contention, at the (in this case, small) expense of having one instance per worker thread:

```
class MyRandomNumberGenerator {
  static ThreadLocal<SecureRandom> rng = new ThreadLocal<
     SecureRandom>() {
    protected SecureRandom initialValue() {
      SecureRandom random = new SecureRandom();
      random.setSeed(/*some good seed*/);
      return random;
    }
  }

  public int next() {
    return rng.get().next(32); // need 32 bits of data
  }
}
```

Java 7 adds a `ThreadLocalRandom` implementation to the standard library.

Data stored in thread-local storage will live as long as the thread. If you need the memory to be reclaimed before the thread terminates, you must explicitly set the storage to `null` via a call to `rng.set(null)`. If you are using the `java.util.concurrent` thread pool framework, then you can use its hooks that are called after a task, or after a thread, terminates. You can do so by extending the `ThreadPoolExecutor` and overriding the `afterExecute` and `terminated` methods, respectively.

Thread-local storage, like the `WeakHashMap`, is an example of the JRE hiding some of the complexity of managing weak references. Under the covers, the thread-local storage implementation uses weak references so that, if a `ThreadLocal` object is reclaimed, then the storage associated with it, for all threads, will be reclaimed, too. In some implementations, this will not happen immediately, because these implementations do not use reference queues. They use an alternative approach that at least keeps the amount of memory spent on stale thread-local storage bounded.

Chapter 14

AVOIDING LEAKS: CORRELATED LIFETIME PATTERNS

Typically, objects die soon after the point in time of their last use, once all dominating references are removed or naturally go out of scope. For a great many objects, the normal flow of method invocations results in local variables going out of scope, which renders these objects reclaimable without any special effort on your part. In the absence of memory leaks, and without any optimizations, objects live and die according to this *natural lifetime*, as discussed in depth in Section 10.2.

However, the built-in lifetime mechanisms, by relying on objects going out of scope, are insufficient to implement the more complicated patterns. Implementations of the correlated lifetime pattern, introduced in Section 9.4, are very prone to memory leaks. You may need an object to survive for a period of time that is not bound to any one method invocation, but rather to the lifetime of another object. The lifetime of some objects are indeed correlated with an invocation, as in the case of objects correlated with a phase or request, but even here there are difficulties. Oftentimes, the invocation that marks the beginning of a request is in a part of the code outside of your control, or is distant from the allocation site of the objects that must go away when the request finishes. Implementations of the time-space tradeoff pattern, introduced in Section 9.3, can be ineffective if they aren't sized properly. They, too, can result in memory exhaustion, e.g. if a cache's key misimplement equality, or if it is sized too large.

It is important to code according to practices that will assure that an object dies when it should. The correlated lifetime and time-space tradeoff patterns are the most difficult cases to get right, and so those most in need of rigorous coding practices.

14.1 The Sharing Pool Pattern

It is very common for applications to store many copies of the same data in multiple data structures. This is especially a problem with strings. Heaps can often store the same string dozens or even hundreds of times. The sharing pool pattern is a way to amortize the memory cost of data over many uses of it that overlap in time. If they don't overlap in time, then you might still find it beneficial to amortize the construction time, but this is a different lifetime pattern. It is a time-space tradeoff, rather than a space-reduction optimization; see the resource pool pattern below.

The general shape of the solution lies in a *canonicalizing map*, one that maps a new data structure to a previously constructed data structure with the same shape and data values. Once you have made the decision to extend the lifetime of an object, of those canonical instances, you have introduced a lifetime management problem.

Example: Duplicate Strings You application loads data from a file. This data contains a large number of name-value maps that will be used frequently throughout program execution. These maps represent configuration information. The names come from a small set of 16 distinct names. The set of string values is unknown at development time, but is known to be small; you know there aren't going to be many distinct values, but you are unwilling or unable to nail them down at compile time. How can these maps be stored in a memory efficient way?

Without any special effort, each instance of this kind of configuration map would store the some subset of same 16 key strings. Furthermore, each map would store duplicates of the values. The following code snippet has those two aspects of duplication:

```
Map<String,String> map = new HashMap<String,String>();
void handleNextEntry() {
   String key = getNextString();
   String value = getNextString();
   map.put(key, value);
}
```

Java provides a built-in mechanism for sharing the contents of strings across many string instances. By *interning* a Java `String`, you ensure that the returned `String` will only have distinct storage if it is a string value that hasn't been interned yet. You can modify the first try as follows:

```
void handleNextEntry() {
   String key = getNextString().intern();
   String value = getNextString().intern();
   map.put(key, value);
}
```

It is possible to do even better, if you have the luxury of modifying both ends of the communication channel, i.e. both the serialization and this deserialization code. There are only 16 distinct names used in all instances of this configuration map. This seems like a perfect case for an enumerated type. An enumerated type can be used to represent strings as numbers at runtime. The only place the strings are stored is in the string constant pool. Each class, when compiled, keeps a pool of the strings that are used by code in that class. In this way, an enumerated type is an even more highly optimized sharing pool than that provided by the interning mechanism.

Enumerated types offer an additional opportunity for decreased memory bloat. Since, in this example, the keys can be represented as an enumerated type, you can use the `EnumMap` to store the mapping. It is over 3 times as space efficient as a `HashMap`, consuming only 28 bytes per collection and 8 bytes per entry compared to 120 bytes per collection and 28 bytes per entry for a `HashMap`.

```
enum PropertyName = {...};
Map<PropertyName,String> map = new EnumMap<PropertyName,
    String>(PropertyName.class);
void handleNextEntry() {
   PropertyName key = getNextPropertyName();
   Object value = getNextString().intern();
   map.put(key, value);
}
```

There is an important variant of a sharing pool called the Bulk Sharing Pool. Like a normal sharing pool, the goal of a bulk sharing pool is to amortize the memor costs of storing data. However, rather than mitigate the costs of data duplication, a bulk sharing pool aims to amortize the costs of Java object headers across the elements in a pool. This is a topic that stretches notions of how to store data beyond the normal Java box, and so will be discussed, along with many similar matters, in Chapter 12.

14.2 The Single Strong Owner Pattern

Lifetime management for temporaries (the lifetime pattern discussed in Section 9.2), and for objects that are part of only one data structure at a time is usually pretty straightforward. The lifetime of a temporary object, such as one created within a method and not used beyond the end of the method, will be nicely governed by local variable scoping rules.

Many lifetime management bugs arise from shared ownership of non-temporary objects. When an object that is simultaneously part of more than one data structure, as introduced in Section 10.2, it becomes hard to keep track of what actions need to be taken in order to make that object reclaimable by the JRE. Even if you make your best effort to avoid this problem, such as by using weak references, you can still have problems. The diamond structures described in Section 10.3 are a good example of a case where, even with weak references, objects may stick around too long. What programming patterns can help to avoid these problems, so that you aren't left hunting down hard to diagnose memory leaks late in the development lifecycle?

Ideally, every object would be part of only one data structure at a time. This would simplify lifetime management issues, because there would be no hidden links for you to track down and eliminate. This is of course not possible in any practical setting. It is necessary for multiple, probably unrelated, parts of the code to need access to a common set of objects. The listener pattern, covered in more detail below, is a common case of this. For example, in user interface code, both the callback handler for user events and the redraw loop will operate on the underlying data model that the view exposes.

A more practical spin on this single-owner ideal is that every object should have a *home base*.

The Home Base as Single Strong Owner

If an object simultaneously is part of multiple data structures, then identify one of these as its *home base*. The home base data structure should be the *single strong owner* of this shared object. Every other data structure must only weakly or softly reference the object.

For example, consider an object that is part of a cache. While it is not in use, the cache is the sole owner of the objects. If it weren't for the cache holding a non-weak reference to the object, it would be reclaimed. When a cached object is being used by the program, it will likely also be part of other data structures; these structures may possibly span multiple threads. The cache is a natural home base, and any other transitory owners of the objects msut be designed so that their ownership is indeed transient.

A first piece of implementing this single strong owner pattern is the home base itself:

```
class HomeBase {
   Set owned = new HashSet();

   public void own(Object o) {
      owned.add(o);
   }
}
```

On its own, the `HomeBase` class provides a repository for strong references, but doesn't help much in assuring that it is the *only* strong reference to the objects.To add this extra level of assurance requires four pieces of logic. First, you need to make sure that every other collection in which these objects are placed does not have a strong reference to the object. Second, it would be a big headache to have to call `HomeBase.own()` on every object that you create. This would heavily pollute your code and be a nightmare to maintain. You can combine the first two, if there are facades for the collections that take care of the registration process for you. Third, there are several important use cases for which the collections are intended to hold data for multiple tasks. Therefore, you can't simply associate one `HomeBase` repository with a collection; e.g. you may have a single map that contains data for multiple tasks, each of which needs its own repository. The final issue is how to ensure that the repository itself becomes reclaimable soon after you are done with it. A rigorous coding practice is necessary to avoid holding on to the repository itself for longer than necessary.

You can use a factory design pattern to help. The factory should have this basic structure:

```
class HomeBaseFactory {
   public HomeBase newOwner() {
      return new HomeBase();
   }

   public Map newMap(HomeBase home) {
      return new WeakHashMap() {
         public Object put(Object key, Object value) {
            home.own(key); return super.put(key, value);
         }
      }
   }
}
```

In this base implementation, the `newOwner` method doesn't do anything fancy. But it does provide factory methods for creating a map facade that takes care of associating ownership with a given repository, while keeping the map itself free of eternally persistent references to the map's contents. Once the repository is reclaimed, then the weakly referenced key will be reclaimed, at which point, or shortly thereafter, the `WeakHashMap` will take care of removing the entire entry (see Section 10.3). From this base implementation, it should be easy for you to implement similar factory methods for the other kinds of collections, such as sets and lists.

It is often the case that the repository for ownership can reside within a thread. If so, you can leverage the thread-local storage mechanism to implement a factory that provides unique ownership respositories

```
class ThreadLocal_HomeBaseFactory extends HomeBaseFactory {
   ThreadLocal<HomeBase> threadLocals = new ThreadLocal<
       HomeBase>();
```

```
    protected void own(Object o) {
      // the thread's HomeBase assumes ownership
      return threadLocals.get().own();
    }

    public HomeBase newOwner() {
      HomeBase home = new HomeBase();
      threadLocals.set(home);
      return home;
      // the caller will now have the only strong reference
          to the HomeBase repository, we maintain only a
          weak reference to it
    }
  }
```

But this implementation suffers from memory drag. After the thread's task completes, the thread-local storage maintains a reference to the `HomeBase` and all the owned objects. It will only be overwritten either when the same thread is scheduled to process a new task, or when the thread terminates. You could overcome this by adding a `clear` method, and inserting a call to it at the right place in your code:

```
  public void clear() {
    threadLocals.set(null);
  }
```

However, this is a messy and error prone solution. An alternative solution is to rely on local variable scoping to automatically clean things up for you. When a task begins, you can grab a strong reference to the `HomeBase` repository, and have the thread-local storage maintain only a weak reference to it:

```
  class ThreadLocal_HomeBaseFactory extends HomeBaseFactory {
    ThreadLocal<WeakReference<HomeBase>> threadLocals = new
        ThreadLocal<WeakReference<HomeBase>>();

    protected void own(Object o) {
      // the thread's HomeBase assumes ownership
      return threadLocals.get().get().own();
    }

    public HomeBase newOwner() {
      HomeBase home = new HomeBase();
      threadLocals.set(new WeakReference(home));
      return home;
```

```
        // the caller has the only strong reference to the
           HomeBase repository, we maintain only a weak
           reference to it
     }
  }
```

Now, all objects owned by the repository will be automatically reclaimable when the return value of `newOwner` goes out of scope.

14.3 The Correlated Lifetime Patterns

Implementing a correlated lifetime pattern in a way that does not result in memory drag or memory leaks is difficult. There are four important cases: annotations, annotation pools, listeners, and phase/request-scoped objects.

14.3.1 Annotations

If you don't have the luxury to change a class definition, but need to associate some information with it, then your only choice is to use a map. To ensure that the lifetime of the annotation is correlated with the lifetime of the annotated object, you can use a `WeakHashMap`. Section 10.3 introduced this utility class, that is part of the standard library. For example, to associate a class `A` with a class `T`:

```
class AnnotationMap<T,A> extends WeakHashMap<T, A> {
   public void annotate(T t, A a) {
      super.put(t, a);
   }
}
```

This implementation works well, at least for single-threaded programs. Soon after an annotated T instance is reclaimed, the `WeakHashMap` will automatically take care of removing the annotation entry from the map. If your application has multiple threads concurrently accessing and creating these annotations, then you will suffer from lock contention. Section 11.5 discusses solutions to this problem.

There is a more immediate potential problem, however. If your annotations are more than simple objects like dates, then you have to be very careful to avoid the strong-weak diamond problem described in Section 10.3. This problematic situation can arise if an annotation, either directly or via some chain of fields, strongly references the annotated object. This is a common and innocent mistake, when you code in a way that avoids the messiness of creating a reverse lookup map, from annotations to annotated objects:

```
class TimestampAnnotation<T> {
   T t;
```

```
      Date date;

      public TimestampAnnotation(T t) {
         this.t = t;
         this.date = new Date();
      }
   }
```

But this example will result in the annotation map leaking memory. You must have the annotaitons weakly reference the annotated object, i.e. the `T t` field must be replaced with a `WeakReference<T> t`.

14.3.2 The Annotation Pool Pattern

[edith has something for this already]

14.3.3 Listeners

Another common instance of the correlated lifetime pattern that is easy to mess up is the listener pattern. A common implementation strategy is to have the list of listeners be a list of strong references to the callback functions. The Java Swing implementation of `JComponent` stores an `EventListenerList` instance, which has an array of strong references to the callback handlers. This implementation strategy has the benefit of being uniform: independent of whether the listener list, or some other collection, is the home base for the callbacks, you follow the same approach. Unfortunately, this approach requires that you maintain and debug code that explicitly deregisters the callback hook from the listener queue.

To avoid this source of bugs, you must follow the single strong owner principle. You must choose which of the listener list or some other collection is the home base for the callbacks. For example, if you already have a place to store the callbacks, then the listener list can be created by a call to that home base factory: ListenerList list = factory.newList().

14.3.4 Phase/Request-Scoped Objects

It is a huge challenge to ensure that an object created within a phase or request dies soon after the phase or request completes. If the object is created at the top level of the request method, and is never stashed into any static fields or fields of objects which are bound to enclosing method scopes, then the normal local variable scoping rules (see Section 10.2) would apply, and life would be pretty easy:

```
void doLogin() {
   Object obj = new Object();
   restOfWork(obj); // if obj does not escape ...
} // ... then lifetime of obj automatically ends here
```

If, during the execution of the `restOfWork` method, `obj` does not escape into some other scope, then its lifetime ends when the `doLogin` method returns; and, possibly, somewhat before that, as discussed in Section 10.2. The lifetime of the object `obj` will be correlated with the `doLogin` request, by the natural local variable scoping rules. However, it is very easy to write code that alters the lifetime of `obj`. This is especially true if you have a distributed team that are collaborating to implement the functionality of `restOfWork`. Since the requirement, that the lifetime of `obj` be correlated with the `doLogin` request is not specified in the code itself, and likely not even in comments or documentation, the team does not know to maintain this lifetime property. For example, a developer may choose to use `obj` as the key into a longer-lived map. This is a common scenario, such as when `obj` is a session identifier that is unique to the user's session or to the specific request being processed. For example, if you are producing a page composed of many parts, each part generated by independently written pieces of code, you can glue them together via a `requestState` map:

```
static Map requestState = new HashMap();
void restOfWork(Object requestKey) {
  requestState.put(requestKey, ...);
}
```

Now, these instances of `obj` will survive for an indefinite period of time. There is no way to be sure of how long these keys will last, because it ends on whether any of the `obj` intances are equal. If two calls to `restOfWork` are passed equal objects, then the first one will be reclaimable shortly after its entry in the map is replaced with the new one.

It is also pretty easy to introduce a memory leak. If you stash the object, in this case as a key, into a map, you must plan out a way to remove it when the `doLogin` request is done. One solution is to associate a cleanup hook with every data structure that should be correlated with a request, and invoke these at the end of a request. You could use the Listener lifetime pattern to do this. Each data structure that possibly contains request-scoped objects must register as a listener. Then, assuming that the request is processed by a single thread, you could combine the Listener lifetime pattern with a use of thread-local storage:

```
/* the cleanup API */
interface CleanupHook { ... };

/* every thread keeps a registry of cleanup hooks */
static ThreadLocal<ListenerRegistry<CleanupHook>>
   requestLocals = new ThreadLocal<ListenerRegistry<
   CleanupHook>>() {
  public ListenerRegistry<CleanupHook> initialValue() {
    return new ListenerRegistry<CleanupHook>();
```

```
      }
   }
   void doLogin() {
      Object obj = new Object();
      restOfWork(obj); // obj might escape!
      requestLocals.get().notifyAll();
   }
```

In some cases, this strategy can be made to work. Mostly, though, and even in the current example, it is not a good approach. The `requestState` map is global, used across all requests. How can you implement a `CleanupHook` that knows which map entries to remove? In general, every data structure in which some request-scoped objects are stored may have this problem. Each may have a different requirement for extracting the correct objects, those for the request that just completed, from the tangle that comes from many other concurrent requests.

How can you design a foolproof strategy that is minimally invasive? It would be nice to piggyback on the automated reclamation that either local variable scoping, or weak references offer. A single strong owner factory pattern, where either a local variable of the top-most method in the request, or thread-local storage, holds the single strong reference to any request-scoped data:

```
   HomeBaseFactory requestLocals = new
       ThreadLocal_HomeBaseFactory();
   void doLogin() {
      HomeBase myLocals = requestLocals.newOwner();
      Object obj = new Object();
      restOfWork(obj);
      // when myLocals goes out of scope, all request scoped
          objects will automatically become reclaimable
   }
   static Map requestState = requestLocals.newMap();
   void restOfWork(Object requestKey) {
      requestState.put(requestKey, ...);
   }
```

This implementation avoids the need for you to code any cleanup logic in the maps and sets that store request-scoped data. You only need to alter the *constructor* of these maps to use the single strong owner pattern; as long as you make sure to call `requestLocals.set(null)`, then any request-scoped objects will be reclaimable immediately after the request completes — all using the normal, built-in scoping rules. It would be even better if you could arrange it so that the `HomeBaseFactory` is a local variable of the top-level request method; then, you only need to change the constructors of the maps and sets that store request-scoped data. There are many minor variations of this base implementation. You can tailor them to your specific needs.

14.4 The Time-space Tradeoff Patterns

There are four important cases of time-space tradeoffs. The first covers the situation where recomputing attributes, rather than storing them, is a better choice. The next three cover situations where spending memory to extend the lifetime of certain objects saves sufficient time to be worthwhile: caches, sharing pools, and resource pools.

Caches If the data stored in a data structure is frequently and expensively recomputed or refetched, and the data values are the same every time, then it is worthwhile to cache the computation or data fetch. The expense of re-fetching data from external data sources and recomputing the in-memory structure can often be amortized, at the expense of stretching the lifetime of these data structures. A good cache defers the time that an object will be reclaimed, as long as there is sufficient space to handle the flux of temporary objects your application creates.

Soft Reference Rule

Soft references must always be over values, not keys. Otherwise, testing equality of keys will trigger a use of the reference. This will extend the lifetme of the value, even though the only use of the entry was in checking to see if its key matches another.

Resource Pools A cache can amortize the cost, in time, of fetching or otherwise initializing the data stored in an object. A sharing pool can amortize the cost, in space, of storing the same data in many separate objects. In both cases, the data is the important part of what is stored.

There is a third case, where one needs to amortize the cost of the allocations, rather than the cost of initializing or fetching the data that is stored in this object. A resource pool stores the result of the allocation, not the data. Therefore, the elements of a resource pool are interchangeable, because it is the storage, not the values that matter. It is important to note that, though the data values are not the important part, the elements of the pool are objects, and are thus intended to store data! A resource pool handles the interesting case where the data is temporary, but you need, for performance reasons, the objects to live across many uses. The protocol for using a resource pool then involves reservation, a period of private use of the fields of the reserved object, followed by a return of that object to the pool.

Resource pooling only makes sense if the allocations themselves are expensive. There are several reasons why a Java object can be expensive to allocate. Creating and zeroing a large array in each iteration of a loop can bog down performance. Creating a new key object to determine whether an value exists in a map can sometimes contribute a great deal to the load of temporary objects.

A more important example of the need for amortizing the time cost of allocation comes when this Java object is a proxy for resources outside of Java. If your application accesses a relational database through the JDBC interface, you will experience the need for resource pooling. There are two kinds of objects that serve as proxies for resources involving database access. First are the connections to the database. In most operating systems, establishing a network connection is an expensive proposition. It also involves reservation of resoures in the database process. Second are the precompiled SQL statements that your application uses. As with the connections, these involve setup cost, of the compilation itself, as well as the reservation of memory resources, that the database uses to cache certain information about the query.

14.5 Concurrency Issues

If your program operates with many concurrent threads, you have to program differently, because straightforward implementations of the above strategies will result in concurrency issues. One of the primary problems will be lock contention, as threads concurrently poll a reference queue. An important example of this problem shows up in the implementation of a cache that can support many concurrent users.

A cache is a map, usually of bounded size, with an eviction strategy for maintaining that bound. The Java standard library provides a concurrent map implementation, in the form of the `ConcurrentHashMap` class, but this is not a cache, because it has no eviction hooks with which one can bound its size.

Following the soft reference rule, and using the basic guidelines for managing reference queues from Section 10.3, leads to a first attempt at a `ConcurrentCache` implementation. You can extend the basic concurrent hashmap, wrapping the map's values with soft references:

```
class ConcurrentCache<K,V> extends ConcurrentHashMap<K,
    SoftReference<V>> {
   private final ReferenceQueue<V> refQueue = new
       ReferenceQueue<V>();

   protected void cleanupQueue() {
      SoftReference<V> v;
      while( (v = refQueue.poll()) != null) {
         remove(???); // oops!
      }
   }
}
```

Oops! This implementation provides no way to remove the key from the map, when cleaning up the evicted entries. To fix this, you'll need to stash a pointer to the key in the soft reference wrapper. It would be nice if the `ConcurrentHashMap` implementation let you extend its implementation so that its `$Entry` class extended soft reference; the `$Entry` would serve this role perfectly. Instead, you have to replicate this pointer structure, at a silly but unavoidable cost of memory bloat. You can do so in a `CacheSoftReference` wrapper:

```
class CacheSoftReference<K, V> extends SoftReference<V> {
  private final K k;

  public CacheSoftReference(K k, V v, ReferenceQueue<V>
     refQueue) {
    super(v, refQueue);
    this.k = k;
  }
}

class ConcurrentCache<K,V> extends ConcurrentHashMap<K,
   CacheSoftReference<K,V>> {
  private final ReferenceQueue<V> refQueue = new
     ReferenceQueue<V>();

  protected void cleanupQueue() {
    SoftReference<V> v;
    // poll causes lock contention!
    while( (v = refQueue.poll()) != null) {
      remove(v.k);
    }
  }

  public V put(K key, V value) {
    cleanupQueue();
    return super.put(key, new CacheSoftReference<K,V>(key,
       value,refQueue));
  }

  public V get(K key) {
    cleanupQueue();
    return super.get(key);
  }
}
```

This implementation still suffers from several critical problems. First, every call to `put` must check the reference queue for pending evictions in order to avoid unbounded growth of the eviction queue — in the steady state, `put` calls are likely to cause evictions. Even though get calls won't cause evictions, in order to avoid pending evictions piling up as cached elements are discovered to be unused, every call to get must also poll for evictions. This can result in foiling the concurrency aspect of the ConcurrentHashMap. Second, if the cache as a whole goes unused for a long period of time, the pending evictions will pile up.

The only way to fix the lock contention problem, at least as of Java 6, is to spawn a thread that periodically polls the reference queue for evicted entries. This will also fix the second problem. This spawned thread's run method will look just like the cleanupQueue method above, except that it should loop indefinitely, and call refQueue.remove() rather than poll(); the former blocks until an eviction occurs (though you must still be careful to check the return value for null, despite what the Javadocs claim).

At first sight, it would seem that you should be able to remove the calls to cleanupQueue from the put and get methods. This, after all, was the whole point of introducing the cleanup thread. However, this modified implementation, while an improvement, suffers from a new problem. Now, if you remove the cleanupQueue calls, when there is a large spike of put calls in a short period of time, you are at risk of running out of Java heap due to a large pileup of pending evictions.

You must have a safety valve in place to prevent this situation. One possibility is to keep an approximate count of the number of put calls, and call cleanupQueue only periodically. In order to avoid lock contention in maintaining this count, you can do so in an unsynchronized way. There is still a pathologic possibility that every racey increment of the put counter won't actually increment the counter. If this worries you, you can use an AtomicInteger, at increased expense. Instead of calling cleanupQueue directly, the put method now calls a new helpCleaner method:

```
static private final int SAFETY_VALVE = 1000;
private void helpCleaner() {
   if (putCount.incrementAndGet() >= SAFETY_VALVE) {
      putCount.set(0);
      cleanupQueue();
   }
}
```

There is no reason for get calls to call this method. The only point of this safety valve is to avoid a sudden large influx of put calls. Indeed, in this final implementation, the ConcurrentCache class needn't override the get method of ConcurrentHashMap.

Chapter 15

TRADING SPACE FOR TIME: CACHES, RESOURCE POOLS AND THREAD-LOCAL STORES

Chapter 16

CUSTOMIZATION AND TUNING: ADVANCED LIFETIME MANAGEMENT TECHNIQUES

Part III

Scalability

Chapter 17

ASSESSING SCALABILITY

Chapter 18

EXTREME SCALABILITY OF LONG-LIVED DATA

All of the tuning advice presented so far in this book can only get you so far. Sometimes, despite your best efforst of tuning entities and collections, your application's objects still do not fit into the memory constraints of the target platform. Managing a great number of long-lived objects therefore comes with its own set of challenges, even though objects these objects are free from the bugs that riddle those with correlated lifetime. To manage objects that, after a reasonable amount of tuning, still don't fit into the heap, you have three solutions at your disposal: throw hardware at the problem, implement a kind of demand paging, or code your data models in a non-object oriented way.

Buy More Memory The first solution you may consider is buying more memory for the target platform. If your budget allows for this, then by all means you should strongly consider doing so. There are some downsides to keep in mind. The first is heap size limits. At some point, you will need to switch from a 32-bit JVM to a 64-bit JVM. This switch will result in a large increase in overhead. As discussed earlier, the amount of blowup resulting from a switch to a 64-bit JVM depends on the degree of delegation in your data models. A 64-bit JVM only adds overhead to the extent that your data models have headers and pointers. The alternative, running with compressed references, limits you to around 32GB of memory, and imposes a few percent slowdown. In addition, running with a large heap can result in increases in the length and fluctuations in pause times.

Shuttle Objects in and Out of the Java Heap Rather than attempting to fit all of your objects into a single heap, you can store them outside of the Java heap, and swap them in (and out), on demand. If the logic of your application allows for recomputing the data stored in these objects, you need to be careful to compare the recreation cost with the costs of marshalling objects. You can choose to marshall objects to and from a local disk, or you can use one of several frameworks that provide a distributed key-value map.

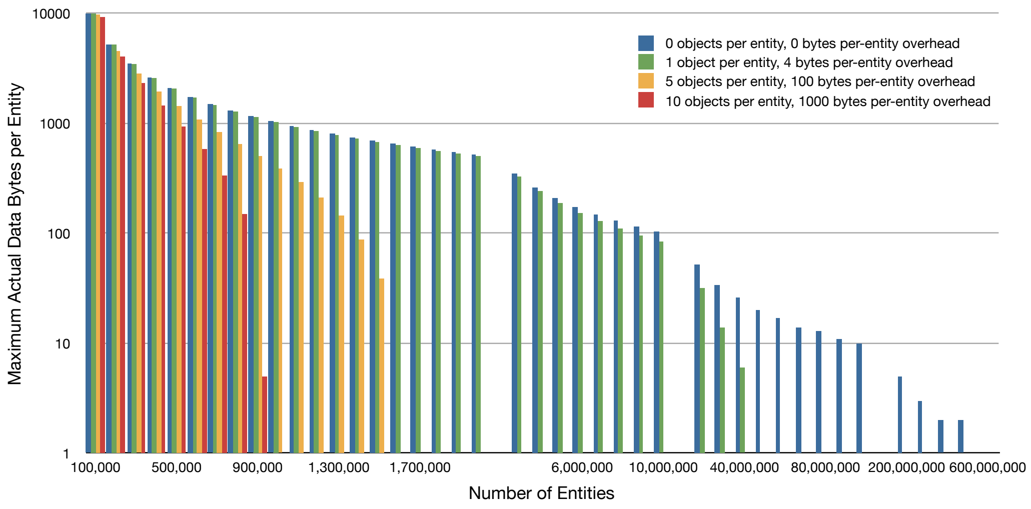


Figure 18.1. The amount of actual data you can store in your entities depends on the degree of delegation in your entities, and the per-entry overhead of the collection in which these entities reside. This chart assumes a limit of 1 gigabyte of Java heap.

Break the Java Mold Despite being an object-oriented language, there is nothing in the Java language that prevents you from storing objects in a non-object oriented way — nothing, that is, except programming time and maintenance expense. It is possible to store data only in arrays, as one would in a language such as Fortran, and retain a great deal of object orientation in your data models and programming interfaces.

This chapter presents a methodology for predicting the scalability of your data models, and presents several solutions to demand-marshalling objects, and for storing objects in a "Fortran style".

18.1 Scalability: Quantitative Methodology

TODO: write this section, explaining Figure 12.1, or delete it.

18.2 Bulk Sharing of Immutable Data

Primitive arrays are often the repository for most of your application's actual data. If your application has a few large primitive arrays, then the overhead of the arrays will be dwarfed by the actual data contained therein. A common problem with Java applications is that they have many small primitive arrays. If the large majority of your actual data is in these primitive arrays, and the average length of each array is 10 characters, then your application will have a memory bloat factor of 61% — almost half of your heap will be wasted on the object headers of the primitive arrays.

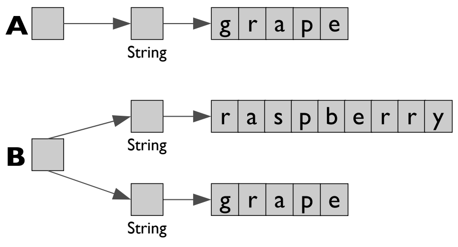
The problem grows worse if these primitive arrays are wrapped inside of objects such as `String` or `ByteArrayOutputStream`. If wrapped inside of `String` objects, then your application is doomed to a memory bloat factor of 83%.

If these primitive arrays store only a small number of distinct sequences, this problem of high overhead can, at least indirectly, be addressed by interning. Earlier, Section 5.3 discussed string interning. If your primitive data is string data, then you can use the JREs built-in mechansm for pooling this data, so that only one copy of each string is kept in memory. For non-string data, you will have to roll your own solution. In any case, however, interning only tackles a part of the problem at hand. By keeping only a single canonical copy of the primitive data, you eliminate the primitive array headers for the duplicates. Figure 12.2a and Figure 12.2b illustrate a simple case of interning. Of three strings, there are two duplicate “grape” sequences. The residual high overhead is due to the remaining primitive array object headers, and all of the `String` objects; none of the `String` objects are eliminated by interning. Interning removes only the duplicate primitive array overheads.

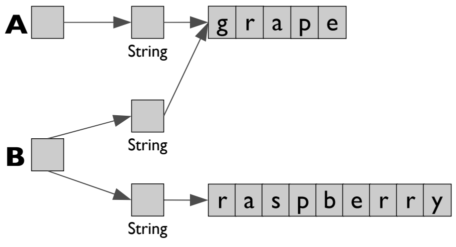
By eliminating many duplicates via interning, you stand to save a large amount of memory, but your memory bloat factor will still be quite high. Instead of a bloat factor of 83%, with interning your heap will have a bloat factor of at least 88%. This value is a lower bound, because there is another curious problem that arises when handling duplicate data. If there is a bounded number of distinct sequences, but an increasingly large number of total sequences, your memory bloat factor can approach 100%. In this case, the per-sequence memory overhead grows with the number of sequences, but the amount of actual data is bounded by the number of distinct sequences.

There are two further optimizations open to you, but these require deeper changes. Both optimizations store all of the sequences in a single, large array. This storage style is an example of optimizing for a bulk of uniformly typed data that is accessed in a uniform, and stylized, fashion. Why pay the expense of many small primitive arrays, if your code does not need each sequence to be a Java object? If you will never synchronize or reflect on sequences, as objects, then the primitive array header is a needless expense. Figure 12.2c illustrates this bulk storage of the sequences in a single large array. You pay the primitive array overhead just once, across all pooled sequences, rather than for every sequence.

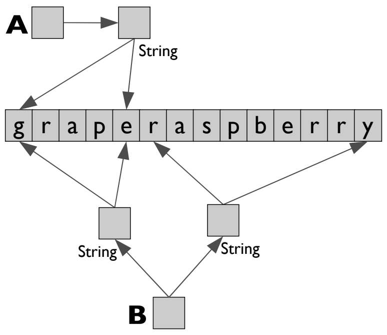
To achieve the ultimate in memory efficiency, you must also eliminate the `String` wrapper objects. This last step requires the most work on your part. If you have the luxury of modifying the class definitions for the objects that contain the string wrappers, then you can replace every pointer to a string with two numbers. These numbers store indices into the single large array of bulk data, and demark the sequence that the string wrapper would have contained. You are essentially inlining the offset and length fields that every Java `String` object has, and doing away with the hashcode field and the extra header and pointers. This eliminates almost all sources of overhead.



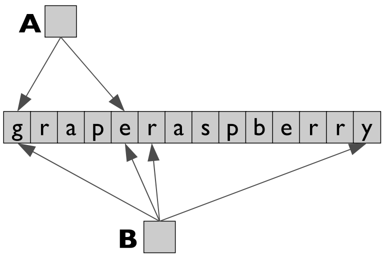
(a) Three normal `Strings`.



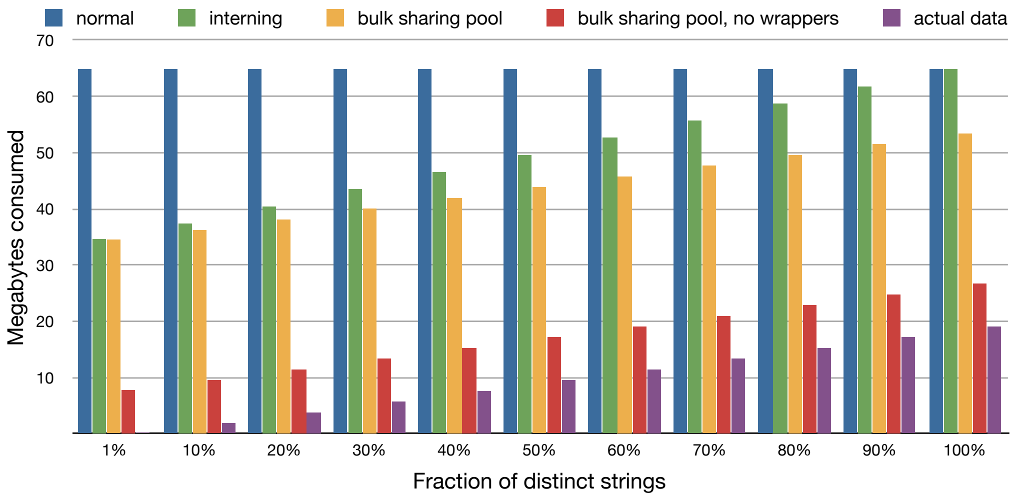
(b) Interning.



(c) Bulk sharing pool.



(d) Bulk sharing pool, no wrappers.



(e) The memory consumed by one million strings, each of length 10 bytes, for varying degrees degrees of distinctness; e.g. 10% means that there are only 100,000 distinct strings.

Figure 18.2. If your application has many long-lived, but small, arrays of primitive data, it could suffer from high overhead. Interning avoids duplication. On top of interning, a Bulk Sharing Pool offers the additional potential for you to eliminate the primitive array wrappers, and so can be beneficial if there are still a large number of unique sequences.

Figure 12.2d charts the memory consumption of the four implementations: using normal Java `Strings` without any attempt to remove duplicates; using Java's built-in string interning mechiansm; using a bulk sharing pool; and using a bulk sharing pool without any `String` wrappers. The chart also includes a series comparison with the amount of memory consumed by actual data. The chart shows memory consumption of each implementation for varying degrees of distinctness of the strings, for an case with one million strings of length 10 characters each. For example, if 500 thousand of the million strings are distinct (which corresponds to the 50% point in the chart), the normal implementation consumes 65 megabytes, the interning implementation consumes 50 megabytes, the bulk sharing pool implementation consumes 44 megabytes, and the bulk implementation without string wrappers consumes 17 megabytes. There are 500 thousand distinct characters, so the actual data consumes about 10 megabytes.

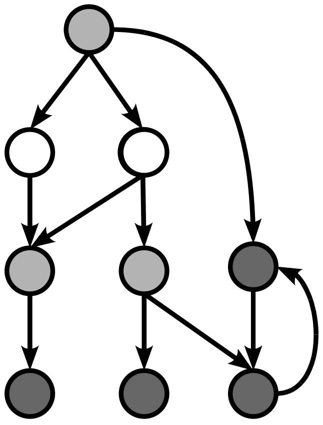
The chart makes it pretty clear how much a few headers and pointers can affect the scalability of your application. With only a bit of work, you can have an implementation that scales very well, with only minimal overhead on top of the actual data.

18.3 Representing Relationships

When representing relationships between entities, using standard object oriented practices, you are severely limited in the number of entities and relationships that you can store. Doing so in a straightforward manner, one that obeys object oriented practices, is very similar to the task of representing a graph of nodes and edges. For example, when caching data from a relational database in the Java heap, the entities (rows in a database table) and relations (columns that contain indices into tables) become the nodes and edges in a graph.

If a graph has a great many nodes and edges, you must design its storage carefully. Consider the small example illustrated in Figure 12.3. There are several ways to implement the abstract data types, of nodes and edges, shown in that figure. Each strategy has its positives and negatives, depending on whether ease of maintenance or memory consumption are of primary importance.

Figure 12.3 shows the interfaces for a `Graph`, `Node`, and `Edge` that one must implement. A common implementation strategy, applied to the `Node` and `Edge` data types, is a straightforward mapping of interfaces to concrete classes. Figure 12.4 shows such an implementation for the `Node` data type. Following this strategy, the `Node` class has three fields, to store its `Color` and the relations to children and parents; these relations are implemented with a standard collection, likely a `HashSet`. The `Edge` class has two fields, to store pointers to the source and target nodes. This implementation is easy to implement. It is also easy to maintain: changes to the interfaces can be directly mapped to changes to the implementation, because the two are parallel versions of each other. The nodes and edges, and relations



(a) Example graph.

```
interface Graph {
   Set<Node> getNodes();
   Set<Edge> getEdges();
}
interface Node {
   Color getColor();
   Set<Edge> getChildren();
   Set<Edge> getParents();
}
interface Edge {
   Node getFrom();
   Node getTo();
}
enum Color {
   White, LightGray, DarkGray
}
```

(b) Abstract data types for nodes and edges.

Figure 18.3. An example generic graph of three colors of nodes.

```
class Node {
  Color color;
  Set<Edge> parent;
  Set<Edge> child1;
  Set<Edge> child2;
}
```

(a) Concrete classes

```
public Node(Color color) {
   color = color;
   parents = new HashSet<Edge>();
   children = new HashSet<Edge>();
}
```

(b) Node constructor that uses the default `HashSet` constructor.

Figure 18.4. A straightforward implementation of the Node interface.

between them, are objects that can be manipulated using normal object oriented practices; e.g. you can write `node.getChildren().get(5).getTo().getColor()`, which reads as a fairly natural, albeit verbose, expression of what you intend.

A Straightforward Implementation Without any thought for memory concerns, the constructor for a node would be as shown in 12.3. In this implementation, the memory cost per node is three pointers plus two collections of default size. As Table 6.1 shows, the the default constructor of a `HashSet` allocates an array to hold 16 elements; the per-collection cost is 136 bytes and the per-entry cost is 28 bytes. If, on average, a node has one parent and 2 children, then each node will consume one object header plus $3 * 4 + 2 * 136 + 2 * 3 * 28$, or 464 bytes; using `ArrayList` instead of a hash map would cost $3*4+2*(80+10*4)$, or 264 bytes. Both choices result in 24% of memory wasted on null pointers. The 136 bytes (29%) spent on the parent collection is unnecessary, for those nodes that have exactly one parent. Even an optimally structured set which includes just one pointer for a set with one entry would still impose two pointer costs to reference a single parent node: one pointer to reference the set, one for the set to reference the parent node.

In addition to the cost of the nodes are the cost of the edges objects. By objectifying each edge, this implementation pays a cost of one object header plus two pointers, or 24 bytes (20 bytes, before rounding up to an 8-byte alignment boundary). For the example with an average of one parent and two children per node, the effective cost per node is $264 + 3 * 24$, or 336 bytes.

Ideally, a node with one parent and two children should consume four pointers, or 16 bytes: one to reference a `Color`, one to reference the parent node, and two pointers to reference the children. The disparity between this optimal value and the cost of the standard implementation is 320 bytes. A inspection of Figure 12.1 shows that this implementation, with 16 bytes of actual data and 320 bytes of overhead, can support at most 2.7 million nodes per gigabyte of heap. An ideal implementation, with no storage costs beyond the necessary 16 bytes of pointers, would be able to support at most 65 million nodes per gigabyte of heap.

Optimizing the Implementation By specializing your code to handle only a limited degree of functionality, you can achieve a fair degree of compactness without much effort. If every node has exactly the same number of incoming and outgoing edges, an easy alternative implementation presents itself. You can specialize the implementation for this structural special case, and thereby eliminate the waste that comes from allowing a flexible number of incident edges. Figure 12.5a shows an implementation of the `Node` interface that does away with collections. On top of the ideal implementation, the cost per node is one object header plus two pointers for each of the three `Edge` objects. On top of the 16 byte ideal cost, this implementation costs an additional $3*(2*12+2*4)$ bytes, for a total of 112 bytes. This cost is one third that of the initial implementation, though 7 times the cost of the ideal implementation. This implementation supports just over 9 million nodes per gigabyte of

```
class Node {
   Color color;
   Edge parent;
   Edge child1;
   Edge child2;
}
```

(a) No collections

```
class Node {
   Color color;
   Node parent;
   Node child1;
   Node child2;
}
```

(b) No objectified Edges

Figure 18.5. Two implementations that have been specialized for the case where no object has more than one parent and no more than two children.

heap.

Even if many nodes have no parents, or fewer than two children, this specialized implementation remains preferable to the original one that uses collections. This is because an empty collection consumes no less than then the one pointer that this collection-free implementation costs; this is the case if you point to the singleton `Collections.emptySet()`. In the case when a node has one child, then a set must be allocated, whose expense, even if the set has only a single entry, will always be higher than a single pointer.

Figure 12.5b shows a yet more highly optimized `Node` implementation that stores pointers to `Node`s, rather than `Edge`s. One somewhat extreme variant of this implementation does not store `Edge` objects at all. This `Edge`-free implementation approaches the ideal implementation, in its capacity for storing large graphs. You must still pay one object header per node, for the `Node` objects themselves. Thus, this impementation has a per-node memory cost of 16 bytes for the pointers plus 12 bytes for the header. At this unit cost, a one gigabyte heap would support at most 38 million nodes.

Though quite scalable, this implementation presents several complications. First is the obvious limitation to at most one parent and at most two children per node. Second, an `Edge`-free storage strategy dictates that the `Node` API also be updated so that `Node.getChildren()` and `getParents()` return a set of nodes, rather than a set of edges. You cannot avoid storing `Edge` objects and yet support an external interface of `Edge`s The same issue holds for an implementation that stores only a single parent pointer: how can one efficiently support an interface that expects a set of edges, if the storage contains only a single pointer? If you don't make this API change, and instead choose to return facades that route the edge operations properly, users of the API will be in for some surprises. The following implementation does not work:

```
public Set<Edge> getParent() {
   Set<Edge> set = new HashSet<Edge>();
```

```
        set.add(parent);
        return set;
    }
```

This implementation will not reflect any updates that the caller makes to the returned set. It also violates the implicit reference-equality contract of interfaces in Java: two calls to the `getParents()` interface must have reference, i.e. `==`, equality. If possible, you can update the written specification for the interface to indicate that a read-only *copy* of the parent set is returned. Unforunately, neither of these policies can be statically enforced, and are hence prone to misuse. Worse, if the interface design is out of the scope of changes you can make, then you are out of luck and must cache the returned set somewhere, so as to preserve reference equality.

Third, in addition to being quite expensive, in creating a set for every call to `getParent`, this implementation lacks in expressive power compared to the other implementations presented so far. For example you will find it more difficult to extend the graph interface to support edge labels. Adding edge labels with low overhead is not impossible, but requires some careful planning, and thinking outside the Java box.

Difficulties of Supporting Edge Properties in Optimized Implementations If the API method `Node.getChildren()` returns a set of nodes, rather than a set of edges, then the code to access an edge label of the second child of a node can no longer be `node.getChildren().get(1).getLabel()`. This way of traversing the graph leads to a node, not an edge! Instead, edge labels must be fetched from some larger entity, such as the graph model itself: e.g. `graph.getEdge(from, to).getLabel()`, or `graph.getEdgeLabel(from, to)`. Without some care, every edge label query will require a hash table lookup. Your design necessitates storing the edge labels in a side data structure, one whose elements are not directly connected, via pointers, to the nodes.

Storing edge labels in a parallel map-like structure makes sense only if a small fraction of the edges have labels. Otherwise, the costs of the map infrastructure may very well overwhelm the cost of the labels. Let's say that each label consumes 4 bytes of actual data. A quick look at Table 6.1 shows that the per-entry cost of a `HashMap` is 28 bytes. If, without edge labels, your implementation supports 38 million nodes per gigabyte of heap (recall that this implementation supports at most one parent and two children per node) then adding edge labels, in an ideal fashion that adds no additional overhead, will decrease your capacity to 26.8 million nodes per gigabyte — a reduction in capacity of 30%. If you store them in a `HashMap`, your capacity will decrease to at most 10.3 million nodes per gigabyte: the map costs 28 bytes per entry, plus you need to create some sort of `Pair` object to house the source and target nodes that form the map's key, plus you need to create a wrapper object to house the edge label itself. In contrast to the 30% reduction of an ideal implementation of edge labels, this straightforward implementation results in a 73% reduction in capacity. In this case the penalty of using a map is especially high because the starting implementation, i.e. one supporting 38 million nodes per gigabyte, was fairly highly tuned. The more highly tuned your memory design, the larger the negative repercussions of subsequent bad choices. For example, if your starting implementation supported 22 million nodes per gigabyte, then the penalty of a map-based implementation of edge labels would be a somewhat lower 61%.

A more efficient approach, if you have the luxury of modifying the `Node` implementation directly, or subclassing it and overriding factory methods, is to inline the edge labels directly into the `Node` class:

```
class Node {
   Color color;
   Node parent;
   Node child1;
   Node child2;
   Label parentLabel;
   Label child1Label;
   Label child2Label;
}
```

To avoid any need to maps, though, would require you to update the API for fetching edge labels. Calling `graph.getEdge(from, to).getLabel()` with this implementation would require an intervening map structure. In contrast, an API that had the form `node.getParentLabel(0)` and `node.getChildLabel(1)` would allow for direct access to the labels, without a map. Inlining the storage for labels requires a concomitant inlining of the APIs. Therefore, while this implementation approaches the optimal capacity, it is not especially desirable, from an engineering perspective.

Insufficiencies of Pure Object Orientation This activity of tuning the original graph implementation has lead to two difficult ends. First, it is difficul to support nodes with widely varying numbers of parents and children. If all nodes had only a very small number of incident edges, or a very large number, then specialized implementations are possible. Second, optimizing storage has come at the expense of easy extensibility; one can remove the use of `Edge` objects, but, to support edge labels requires either expensive maps to parallel data structures, or pollution of data types not directly connected to the planned extension.

Using a fully object-oriented Java design, it is impossible to achieve both compactness of storage and the generality to handle a variety of graph structures. To do so requires coding in a style that is not conventionally object oriented.

18.4 Breaking the Mold of Object Orientation

The examples of the previous section worked through various optimizations of representing relationships, and illustrated the interplay of storage optimization and API design. Very often, achieving a high level of memory efficiency requires breaking what would generally considered to be the object orientation of the API or the storage. The main goals are to support a combination of flexibility and scalability that is not possible when following the precepts of object orientation.

18.4.1 Supporting Singleton Sets

One common case, when using collections, is that many of them contain zero or only a single element. It seems silly to pay the expense of a full-fledged collection for these special cases. The Java standard library offers has a partial solution to the case of many empty collections, in the form of the family of factor methods that include `Collections.emptySet()` and `Collections.emptyList()`. These are partial solutions, because they only handle the case of unchanging collections. There is no provision, in the standard libraries, for optimized storage of single-entry collections. Consider our graph from the previous section:

```
class Node {
   Color color;
   Set<Edge> parents;
   Set<Edge> children;
}
```

The previous section proposed an optimization for the special case of nodes with at most one parent. If this were always the case, for every single node ever created by your application, then it is valid, and indeed a good idea, to change the `Node` class definition to inline the pointer to that potential part. If it is only sometimes the case, then this trick does not work.

You can regain a degree of flexibility, at the expense of some added coding complexity. First, you must update the `Edge` interface to indicate that an edge is also capable of acting as a singleton set of edges:

```
interface Edge extends Set<Edge> {
}
```

Second, you must update your edge implementations to implement the boilerplate of the `Set<Edge>` interface; this part is straightforward. For example, you must implement all of the mutating methods, such as insertion and removal, to thow an unsupported operation exception. This ugliness wouldn't be necessary if the standard library had an interface that represented an unmodifiable set; instead, an unmodifiable set is only a hidden implementation constructed via the `Collections.unmodifiable` family of factory methods.

Now you have an edge that is also a singleton set of edges. It then becomes possible to have a single node implementation that supports arbitrary numbers of parents, but is also optimized the special case of nodes with only a single parent. This is possible, without your having to change the fields of the `Node` class in any way:

a field `Set<Edge> parents` can, transparently, be either an optimized singleton, or a more general set.

So far, the changes necessary to implement this optimization have been local in scope. They have not violated any principles of abstraction, because changes have been isolated to the `Edge` details. The final change you need to make requires some abstraction-violating changes. This implementation of the constructor and `addParent` does not specialize for small collections:

```
class Node {
  public Node() { // constructor
     this.parents = new HashSet<Edge>();
  }
  public void addParent(Node parent) {
     parents.add(new Edge(this. parent));
  }
}
```

In order to take advantage of the edge-as-singleton-set optimization, you must pollute the node's `addParent` implemnetation. Here is an implementation that optimizes both for empty and singleton sets:

```
class Node {
  public Node() { // constructor
     this.parents = Collections.emptySet();
  }
  public void addParent(Node parent) {
     if (parents.size() == 0) parents = new HashSet<Edge>();
     else if (parents.size() == 1) {
        Edge firstParent = parents.iterator().next();
        parents = new HashSet<Edge>();
        parents.add(firstParent);
     }
     parents.add(new Edge(this. parent));
  }
}
```

This data abstraction violation is unfortunate and only avoidable by reintroducing the very wrappers you have sought to eliminate. It would be nice if `Edge.add()`, i.e. the implementation of the `Set` interface, could reroute of all pointers to this edge to point to a proper set. However, in Java, it is not possible to have `Edge.add()` change this edge's reference identity. Doing so would require a level of indirection, where edges are referenced via *handles*, pointers to pointers, rather than direct pointers. An `ArrayList` does this, by wrapping an object around the underlying array. This storage structure offers a degree of transparency, where the array can be reallocated, or conceivably changed to an entirely different storage structure without any global changes. The Smalltalk language provides a `become` primitive that allows for pointer rerouting, without the use of handles, though at a nontrivial runtime expense. In the worst, but common, case, every call to `become` requires a full garbage collection.

18.4.2 Column-oriented Storage

It would be nice if you could avoid the overhead of collections, without having to pollute your code with special cases. It would also be nice if you could add edge or node properties, without having to worry about the interaction of their storage with the storage of the relations themselves. If your main use of the relational information is for traversals that don't modify the graph structure itself, then there is an opportunity to dramatically reduce the overheads of storing this information.

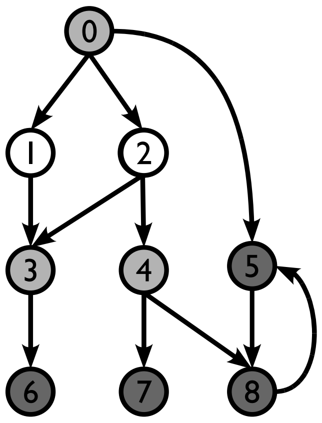
Consider the simple example graph from Figure 12.3(a), which has been updated in Figure 12.6(a) to include a dense index for each node. The indices are natural numbers that range, in this example from 1–9, with no gaps. If each node has a dense index, then each node attributes can be stored in a flattened form. If you store the data compactly, you can avoid much of the pointer and header overhead that plagues most normal object oriented storage representations.

To store data compactly, you have two options: either store data in the style of C, Pascal, or COBOL arrays of records, or in the style of Fortran parallel arrays. An array of C structs, or Pascal records, stores the fields of the structures densely. The fields of each record are stored back to back in the array, without intervening pointers or headers. If the records of node attributes are stored in an array called `nodeAttributes`:

```
// this is C code
typedef struct NodeAttributes {
   Color color;
} NodeAttributes;
NodeAttributes[] nodeAttr;
```

then, to retrieve the color of the node with index 5 becomes `nodeAttr[5].color`.

Unfortunately, Java does not allow for arrays of structures. If the storage size of every field is known, and fixed, you can mimic this storage structure in Java, to a point. However, if your records have fields of varying types, this style of programming can boils down to doing the very same work you would do for marshalling data in and out of the Java heap. If the first and third attributes of your nodes are of `byte` size and the second is of `int` size, then accessing the third field of the node with index 5 is tricky. It requires treating the entire record as a bit bucket, and shifting and masking appropriately.



(a) Example graph, where each node has been assigned a dense integer index.

index	Color
0	LightGray
1	White
2	White
3	LightGray
4	LightGray
5	DarkGray
6	DarkGray
7	DarkGray
8	DarkGray

(b) The node attribute.

from	→	to
0	→	1
0	→	2
0	→	5
1	→	3
2	→	3
2	→	4
3	→	6
4	→	7
4	→	8
5	→	8
8	→	5

(c) Edges.

Figure 18.6. A graph of nodes and edges can be represented as a set of parallel arrays — a column-oriented approach to storage, in the style of Fortran. Edge attributes, along with additional node attributes, would be stored in additional columns. The index column is not a stored attribute, it is only shown in the table, for clarity.

The alternative is to store your data in a Fortran style, where each attribute is stored in its own array. With this storage structure, you avoid the messy details of addressing fields with a non-uniformly typed record. Furthermore, adding new attributes to nodes or edges is a trivial operation. Each is merely an extra array in the set of parallel arrays. In this style of storage, you express collections of objects, rather than individual objects. The individuals are represented by indices. Instead of having a class for a node, you have a class for a group of nodes that participate, together, in a graph or several related graphs:

```
class NodeModel {
    Color[] colors;
    public Color getColor(int nodeIndex) {
        return colors[nodeIndex];
    }
}
```

Figure 12.6 gives an example of a graph with 9 nodes.

If you need the convenience of an actual `Node` object, you can have a bit of both worlds. As long as you use the node objects only for temporary purposes, then you have a good chance of avoiding a memory footprint problem. These transient node objects act as facades to the `NodeModel`, but require a reference to the `NodeModel` and the node's index.

```
class TransientNode {
   NodeModel nodeModel;
   int nodeIndex;
   public Color getColor() {
      return nodeModel.getColor(nodeIndex);
   }
}
```

Though it has one extra field, on top of the earlier node implementations, for the `NodeModel` reference, as long as it is transient, it could be fine. This is something that requires experimentation in your setup. With transient node facades, you are trading off more time spent in garbage collection for convenience and the greater assurances you get from strong typing, compared to passing around integers throughout your code, to represent node indices. If these result in big drags in performance for your use cases, remember that there is no absolute need for these transient node facades. You must also be careful to either disallow, by convention, reference equality checks against transient nodes, or cache any created objects.

The edges of a graph can be represented as two parallel arrays, storing the source and target node indices of each edge, as shown in Figure 12.6c, and prototyped here:

```
class EdgeModel {
   int[] from, to;
   public int getEdgeFrom(int i) {
      return from[i];
   }
   public int getEdgeTo(int i) {
      return to[i];
   }
}
```

However, this edge representation cannot efficiently support graph traversals. There is no efficient way to retrieve the outgoing or incoming edges from a given node. Even a `TransientEdge` class would not help, because there is no connection between a node index and an index into the edge model. Properly representing edges requires a bit more work.

Representing Edges in a Column-oriented Storage Model To allow for efficient traversals of the edges, you must index them, as a database would. First, consider the outgoing edges from a node. If the `EdgeModel` parallel arrays are sorted by the **from** attribute, then all of the children of a node will be stored contiguously. For example, the edges in Figure 12.6c have been sorted in this fashion. At this point, it is easy to establish an index, as an extra attribute of the `NodeModel`. This attribute will store an index into this sorted edge model, which marks where the children of each node begin. Once you have this index, then you no longer need to store the **from** attribute in the edge model; it is implicit, in the combination of the edge-start attribute of the node model. If you do so, however, then you must also store the number of children of each node. Figure 12.7 shows an update to Figure 12.6, where the node model now has, for the outgoing edges, two new attributes: **EdgeStart** and **Num**. The same thing can be done for the incoming edges, if we instead sort the edges of Figure 12.6c by the **to** attribute. Figure 12.7a also shows the two new attributes for the incoming edges. For example, node 2 has two children and one parent. The children start at index 4 in Figure 12.7b, which shows the two children to be nodes 3 and 4.

Limitations of Column-oriented Storage There are two main problems you will run into, with a column-oriented approach. The first has been touched on briefly: the lack of strong typing for nodes and edges. If everything is just an integer, your code will be buggy and hard to maintain. Java does not have a facility for naming types, such as `typedef` in the C language. The Java `enum` construct seems like it could help, but this use case would require a permanent object for every node, and, besides, this construct is limited to around 65,000 entries per enumeration. You can use transient nodes, with some cost to performance, but your code must still obey an implicit contract, one not enforced by the `javac` compiler, that reference equality is never used on these transient facades.

The second problem centers around modifications to the node or edge model. This style of storage works fantasically well, much better than normal Java objects, for certain kinds of modifications. Adding attributes to models is easy. Adding nodes is straigtforward. However, deleting nodes, and adding or removing edges from existing nodes can only be done with some extra work. For deletions, you would have to implement a form of garbage collection yourself. Nodes and edges can be marked as deleted; deleted elements would be ignored by normal access mechanisms. Adding edges to existing nodes is even more difficult. For these reasons, it is highly recommended that you only employ column-oriented storage for data structures that do not change in these ways.

18.5 Serialization

There are often situations where your data structures must be shuttled in and out of the Java heap. Your application might be distributed across multiple boxes that are connected by a network, rather than a distributed shared memory. It is possible that your your data structures do not fit into available physical memory, and so, rather than relying on the operating system's paging functionality, you may find that you must implement a facility for doing so more efficiently. Sometimes, you have enough physical memory, but are constrained by available address space. This

		Children		Parents	
index	Color	EdgeIndex	Num	EdgeIndex	Num
0	LightGray	0	3	–	0
1	White	3	1	0	1
2	White	4	2	1	1
3	LightGray	6	1	2	2
4	LightGray	7	2	4	1
5	DarkGray	9	1	5	2
6	DarkGray	–	0	7	1
7	DarkGray	–	0	8	1
8	DarkGray	10	1	9	2

(a) Node attributes.

index	NodeIndex
0	1
1	2
2	5
3	3
4	3
5	4
6	6
7	7
8	8
9	8
10	5

(b) Children edges.

index	NodeIndex
0	0
1	0
2	1
3	2
4	2
5	0
6	8
7	3
8	4
9	4
10	5

(c) Parent edges.

Figure 18.7. In order to support efficient traversal of the graph edges, you will need to index them. These tables update the example of Figure 12.6 to do so. For example, node 2 has two children and one parent. The children start at index 4 in Figure 12.7b, which shows the two children to be nodes 3 and 4.

is especially likely to happen if you are running on a 32-bit JRE; e.g. if you run with a 1.5 gigabyte Java heap, and have one hundred thousand more more loaded classes, you run the risk of exhausting the 2 gigabyte limit that most 32-bit operating systems have, for each process. On many 32-bit versions of the Windows operating system, and on 31-bit IBM mainframes, the address space limit is even lower than 2 gigabytes. It might also be that you have a need to perform relational queries against your Java data structures, and are considering storing them in a relational store to avoid having to implement the querying logic yourself. Your reality might also be some combination of these.

It is important to remember that, in Java, the terms marshalling and serialization are not synonymous. Serialization is the task of writing out the bits, and the form of those bits. Marshalling, in the Java lexicon, is serialization plus the assurance that all parties in the communication can agree upon the version of the code that generated those bits. Without this agreement, it can become difficult to manage evolving code bases, in the face of legacy installations. This disparity of terminology is somewhat unique to Java, but the distinction is nonetheless important.

This book does not address the issues of compatibility of code bases, other than to observe that, if your use case does not require handling code skew, then you have options that will perform much better than using the built-in Java marshalling mechanisms. In Java, there are several commonly available mechanisms for marshalling your objects into and out of the Java heap. These include the built-in Java object serialization, the Apache `XMLSerializer` and `XMLDeserializer`. There is also a variety of libraries that provide a mapping between Java objects and a relational backing store; an example is RedHat's `Hibernate`. All of these come with a fair amount of expense, because the *operational* form of the data, that is the way the bits are laid out as your Java code operates on them, is different from the serialized form. Therefore, use of these serialization libraries usually entails an expensive translation between two disparate storage formats.

18.5.1 Memory Mapping

One way to bring data in and out of Java without paying a marshalling expense is via memory mapped I/O. When you *memory map* a file into your address space, you can treat the file as if it were an array. Reads and writes to the array are reflected as disk reads or writes, and these operations are usually done at a page granularity. Actual disk I/O may not occur with every array access. This is the case if the operating system decides that it has enough physical memory to keep the written pages in memory, and you haven't specified that array writes should be written through to disk every time. Reads may be serviced from this cache, as well. In this way, memory mapped I/O can have the benefit of well-tested caching that balances that performance needs of all processes running on your machine, without any work on your part. Memory mapping is a common trick used that is used by C programmers seeking the ultimate in performance.

Since the cache is managed by the operating system, cached pages persist across process boundaries. Therefore, if your application runs as multiple steps, each in a separate process, then storing your data structures in memory mapped files can result in a combination of caching and serialization-free persistence. The unwritten buffers may eventually find their way to disk (if the underlying file is not deleted first), but this needn't happen when one process terminates.

Used to its utmost, memory mapping can additionally offer one-copy bulk transfers of memory to and from other processes, disk, or the network. For example, say your application takes data from the network and writes it to disk. If you memory map the network input buffers and the output file, and issue bulk transfers from the input to the output, then it is possible that the operating system will transfer the data directly from the network buffers to the disk buffers, without first copying them out of the kernel, or into the Java heap.

Memory Mapping in Java As of Java 1.4, a memory mapping facility is provided by the `java.nio` library. This Java library manifests data as `ByteBuffer` objects. This interface mimics an array, providing random access and bulk `get` and `put` methods, but no insertion or deletion operations. Using this common interface, you can access four repositories of data: network transmissions, files on disk, memory allocations in the native heap, and allocations in the Java heap. While the latter two can serve only as transient repositories for your data, they avoid the expense of having to create a file on disk — an expensive operation, on most file systems, if done frequently.

You may find it useful to have the option to have some data stored in transient storage, and others in persistent storage, backed by files on disk, and interact with both using the same API. The `java.nio` library lets you do this. An important advantage of using native `ByteBuffer` storage, over Java heap storage, is that your application can run on arbitrarily large inputs without the constraints of a fixed-maximum size Java heap.

No matter where the data is stored, you also have a choice of whether to use standard Java byte ordering, or the byte ordering of the platform on which the application is executing. Of course, this only matters if the data values you are accessing are larger than a byte. On top of a `ByteBuffer`, you can layer other primitive-type views. For example, the instance method `ByteBuffer.asIntBuffer()` returns an `IntBuffer` that takes care of any bit manipulations that are necessary to access the data as Java integers. If you have chosen to use native byte ordering, then accessing the elements of an `IntBuffer` on a 32-bit JRE requires no bit manipulation.

An example use of this library to create an integer array-like view over a file is:

```
IntBuffer mapAsIntegers(File file) {
   ByteBuffer buffer = new RandomAccessFile(file).
      getChannel().map(MapMode.READ_WRITE,0,file.length());
   // use platform, not Java, byte ordering
```

```
    buffer.order(ByteOrder.nativeOrder());
    return buffer.asIntBuffer();
}
```

Memory Mapping Your Column-oriented Storage There is a beneficial combination of memory mapping and a column-oriented approach to storing your data. The interface to all memory mappings is an array, and a column-oriented storage structure stores data as arrays. The update, from the previous `EdgeModel` is easy, but requires a notion of a namespace for each edge model. The namespace is necessary so that the model can be mapped in from disk:

```
class EdgeModel {
    IntBuffer from, to;
    public EdgeModel(File namespace) { // constructor
        from = mapAsIntegers(new File(namespace, "from"));
        to = mapAsIntegers(new File(namespace, "to"));
    }
    public int getEdgeFrom(int i) {
        return from.get(i);
    }
    public int getEdgeTo(int i) {
        return to.get(i);
    }
}
```

Difficulties with Memory Mapping in Java Each memory mapped file consumes only as much *physical* memory as is available and which the operating system decides, in its balancing act, is worthy to allocate to the mapping. While all major operating system manage consumption of physical memory, they do not similarly manage address space consumption. Thus, each mapped `ByteBuffer` consumes a swath of address space equal to the *full* size of the map. For this reason, if you decide to use memory mapping, you will run into several pernicious and interconnected problems. These problems are orthogonal to any problems that stem from a choice to use column-oriented storage.

The primary problem comes from the lack of an unmapping facility in the `java.nio` library. You can not explicitly unmap a mapped area. Instead, and, quite oddly, in direct contradiction of widely published best practices, the library takes care of unmapping in the `finalize` method of the mapped `ByteBuffer`. This leaves the timing of unmapping at the whim of garbage collection and the subsequent scheduling of the finalizer thread. If you application allocates very little in the way of temporary objects, but uses memory mapping extensively, you will have failures due to address space exhaustion. The garbage collector won't run, because plenty of Java heap is available for allocation. However, memory mappings fail, because the process's address space is fully consumed by older mappings, many of which would be unmapped if the garbage collector were only to run.

This implementation decision has a strong negative interaction with the second problem: the JRE does not, upon address space exhaustion, attempt to run a garbage collection and finalization pass in order to clear out mapped byte buffers. Therefore, you may suffer from `OutOfMemoryException` failures, due to running out of address space, despite the fact that many of your mapped regions are actually ready to be unmapped.

The third problem you may encounter is primarily to be found on Windows platforms. The Windows operating system does not allow a file to be removed from the file system if there are existing memory maps over any part of the file. This, combined with the inability to explicitly unmap a mapping, can lead to a situation where files remain on disk when you no longer need them. For example, this can happen if there is a point in your code where you know the file is no longer necessary, but there exist references from live objects or the stack to the `ByteBuffer` object that represents the memory mapping. Since the `ByteBuffer` facade is not garbage collectible, then its `finalize` method will not be called, and hence the unmapping will not occur.

There is a set of policies you can follow to reduce the likelihood of such problems. First, if possible, you should implement a correlated lifetime pattern for the `ByteBuffer` objects. If these objects become garbage collectible as soon as a phase or request completes, or correlated object is collected, then the `ByteBuffer.finalize` method will be called as soon as possible after that correlated event occurs. Next, you must trap all memory mapping failures, force a garbage collection and finalization pass, and then retry the memory mapping; doing this several times in a loop is recommended. Third, on some versions of the JRE, you will find that a memory mapping failure results in the JRE terminating the process. This was one, by the JRE developers, with the thought that a memory mapping failure was a sign of catastrophic failure. You know now that this is not the case, it is rather a symptom of the overall poor design of the `java.nio` library. To work around this problem, you can set up a security policy that disables calls to `System.exit`, though you must be careful that your own code does not rely on this call.

The last good design principal, when using memory mapping in Java, is to rely on file-based memory mappings as little as possible. If you need only temporary mappings, then consider using the anonymous mappings described earlier:

```
IntBuffer allocateAnonymousInts(int numBytes) {
   ByteBuffer buffer = ByteBuffer.allocateDirect(numBytes);
   buffer.order(ByteOrder.nativeOrder());
   return buffer.asIntBuffer();
}
```

When using anonymous maps, you must be aware of the default limits on these native allocations. With Sun JREs, you can configure the maximum allowed number of bytes allocated in this way via the command line argument `-XX:MaxDirectMemorySize`; this option takes the same arguments as the `-Xmx` setting. With IBM JREs, you do not need to specify a maximum value.

Chapter 19

WORKING WITH SECONDARY STORES

19.1 Serialization

19.2 Memory mapping

Appendix A

TOOLS TO HELP WITH MEMORY ANALYSIS

Appendix B

JRE OPTIONS RELATED TO MEMORY

Appendix C

A COMPARISON OF SIZINGS ON JREs