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.

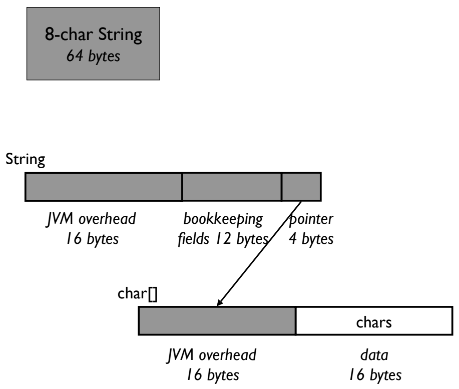


Figure 2.1. An eight character string in Java 6.

- *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.

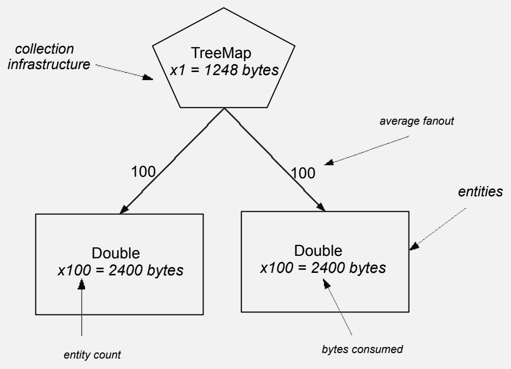
Strings, in particular short strings, are pervasive. It is important to be aware of the overhead of strings when incorporating them into your data design. Given the overhead of Java strings, the choices you make in Java need to be different than in a language like C, where the string overhead is minimal. 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

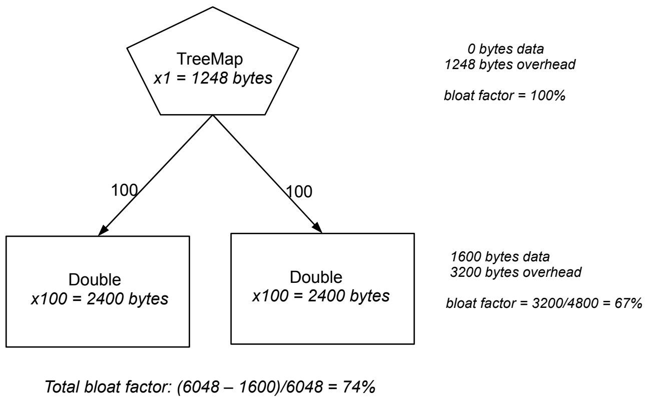


Figure 2.2. EC Diagram for 100 samples stored in a `TreeMap`

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;
```

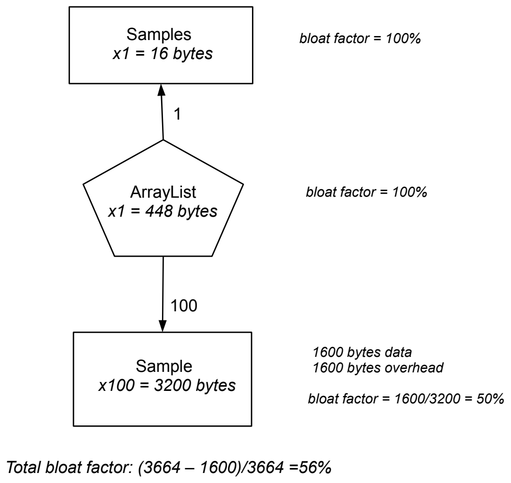


Figure 2.3. EC Diagram for 100 samples stored in an `ArrayList` of `Samples`

```
        }
        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?

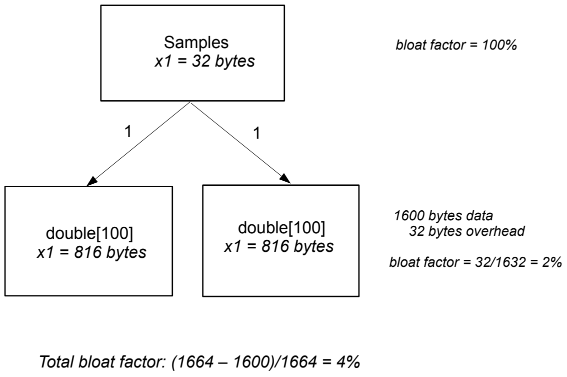


Figure 2.4. EC Diagram for 100 samples stored in two parallel arrays

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.

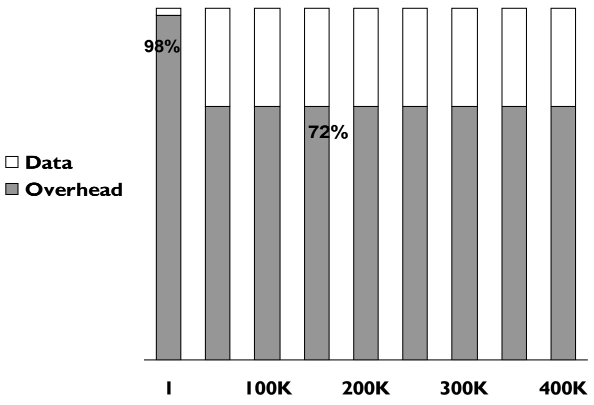


Figure 2.5. Health Measure for the `TreeMap` Design Shows Poor Scalability

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. 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`

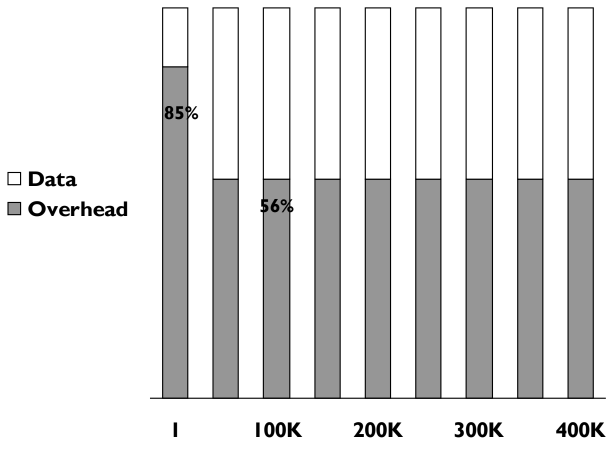


Figure 2.6. Health Measure for the `ArrayList` Design

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.

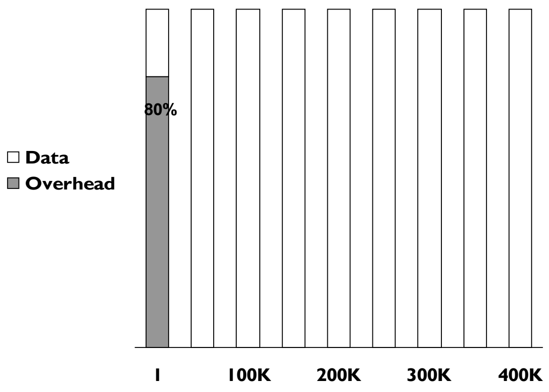


Figure 2.7. Health Measure for the Array-Based Design Shows Perfect Scalability

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 (Service Release 4), 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 `char`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
}
```

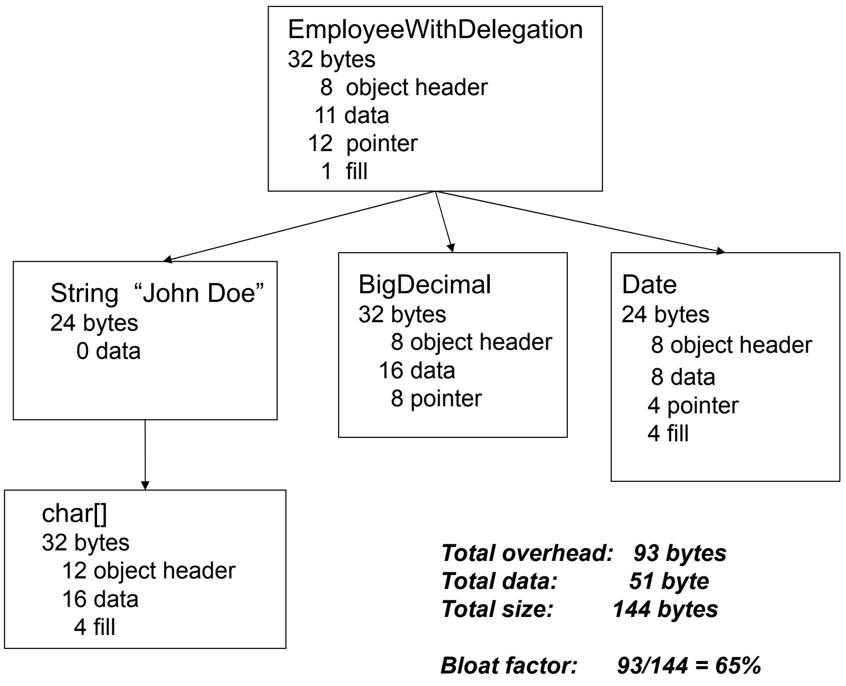


Figure 3.1. The memory layout for an employee "John Doe".

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;
```

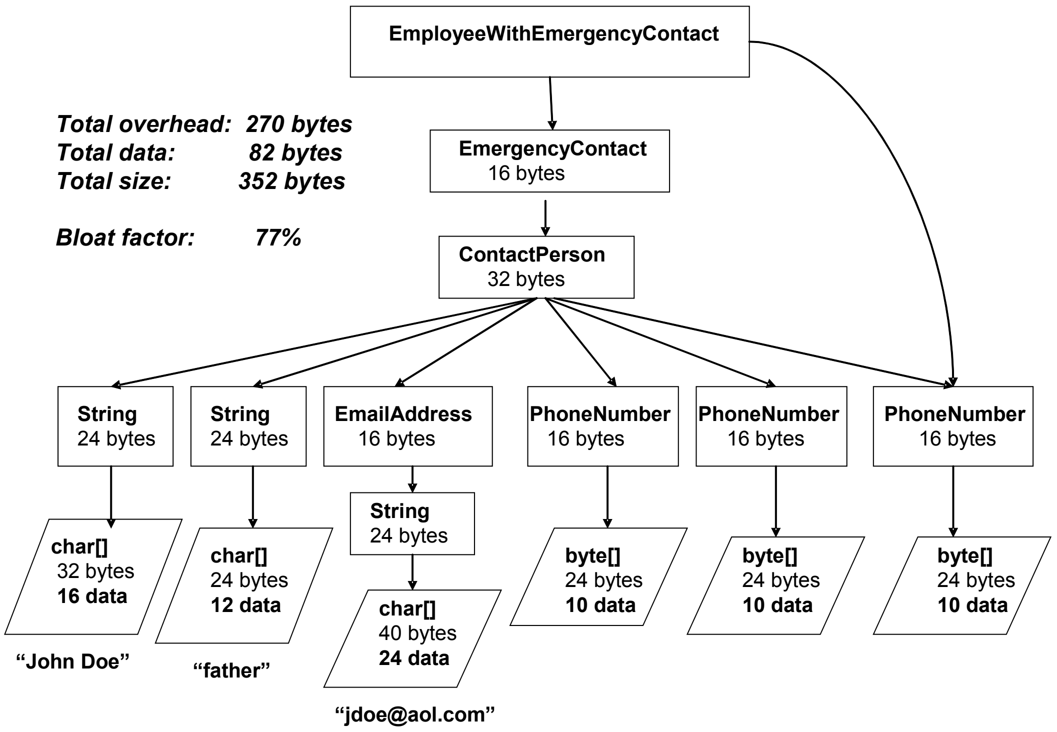


Figure 3.2. The memory layout for an employee with an emergency contact.

```
        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

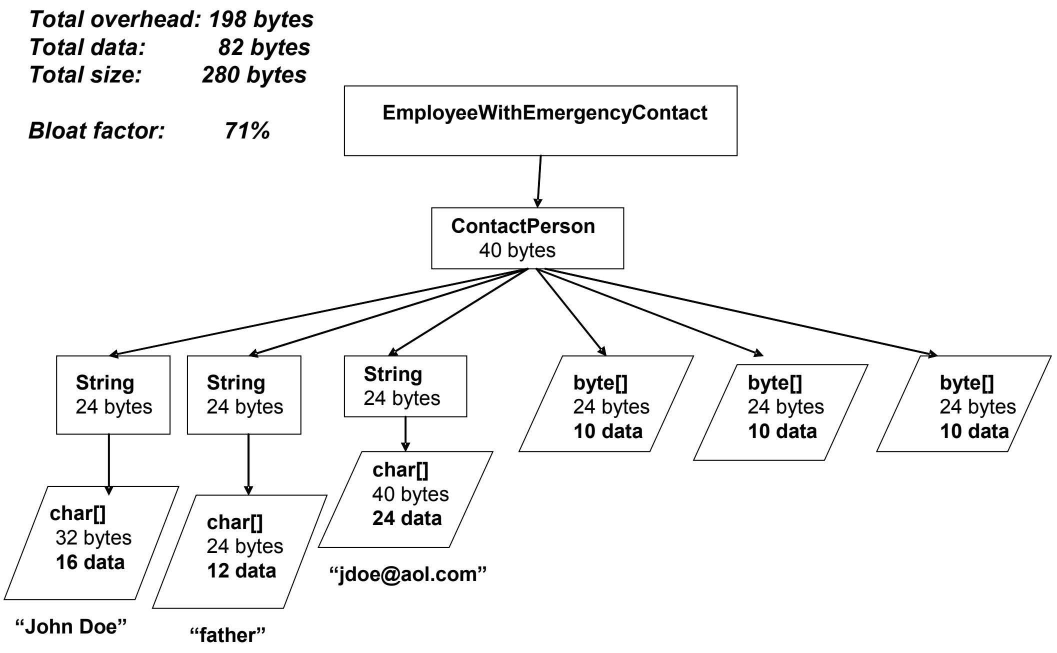


Figure 3.3. Memory layout for refactored emergency contact.

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.

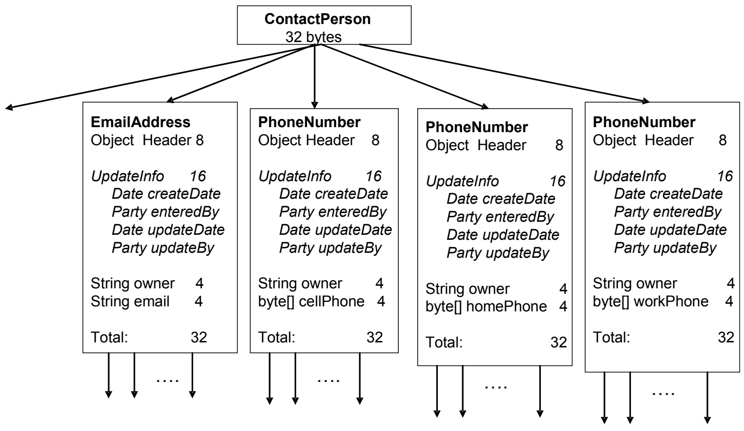


Figure 3.4. The cost of associating `UndateInfo` with every `ContactMethod`.

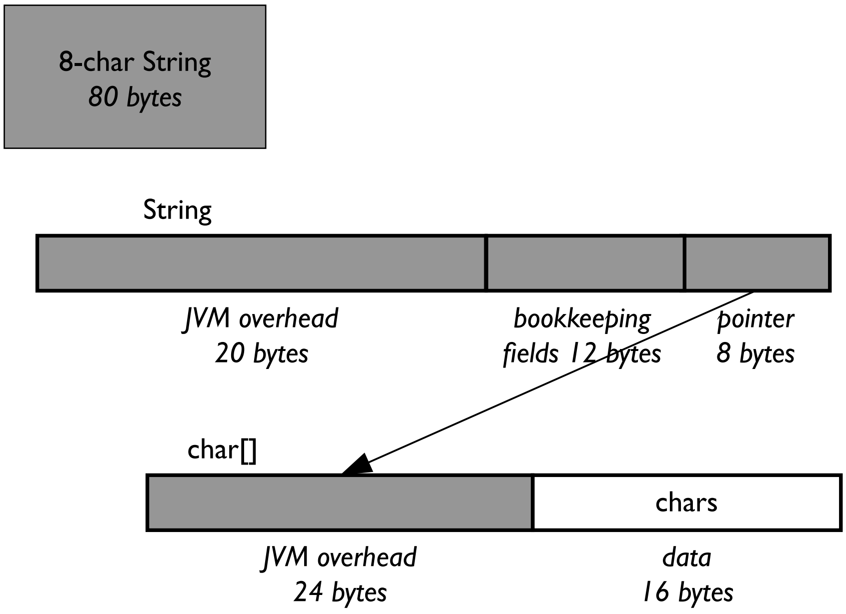


Figure 3.5. The memory layout for an 8 character string by a 64-bit JRE.

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.[1] It is available in IBM Java 6 (Service Release 4) with the option -Xcompressedrefs.

[1]On Java 7, 64-bit Sun JREs have this enabled by default, as long as your heap size is smaller than 32 gigabytes.

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;
    ..
```

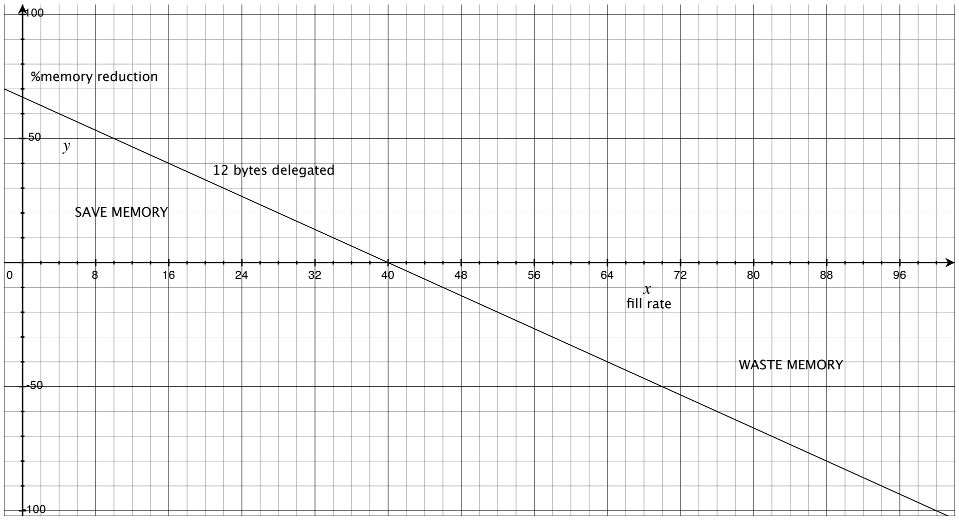


Figure 4.1. This plot shows how much memory is saved or wasted by delegating 12 bytes of memory to a side object. The x-axis is the fill rate, and the y-axis is the percent of memory saved.

```
    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

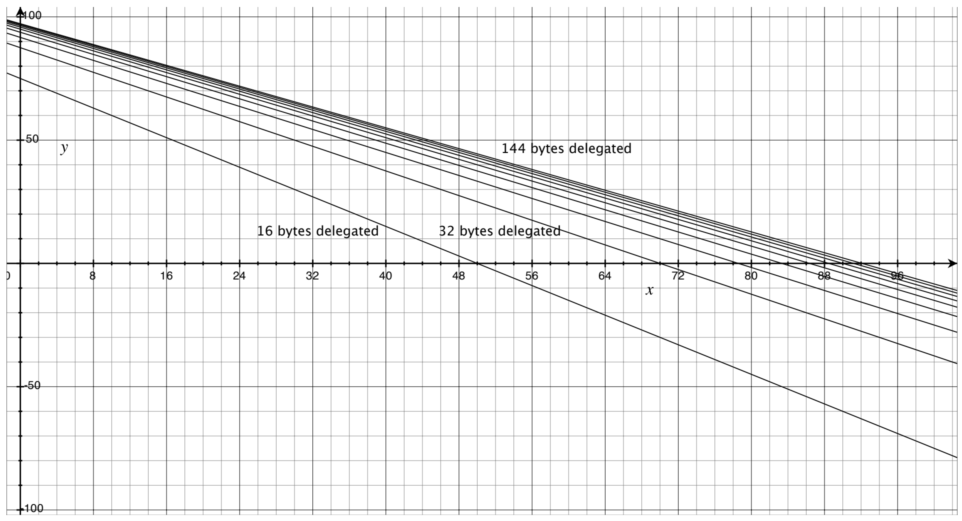


Figure 4.2. This plot shows how much memory is saved or wasted for different delegated field sizes. The x-axis is the fill rate, and the y-axis is the percent of memory saved. Each line represents a different delegated size, starting from 16 bytes, going up to 144 bytes by increments of 16 bytes.

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 `int`s, but more easily printed as `String`s. 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

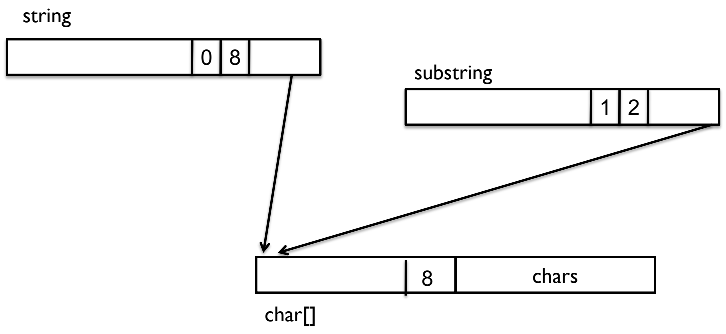


Figure 4.3. A string and a substring share the same character array. The length and offset fields are needed for the substring, but are redundant in the original string object.

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 FIELD VALUES

5.1 Character Strings

Strings vs. Compiled Forms

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.

5.2 Representing Bit Flags

5.3 Dates

5.4 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 6.2 introduces the notion of a sharing pool for sharing dynamic data. Section 6.3 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*,

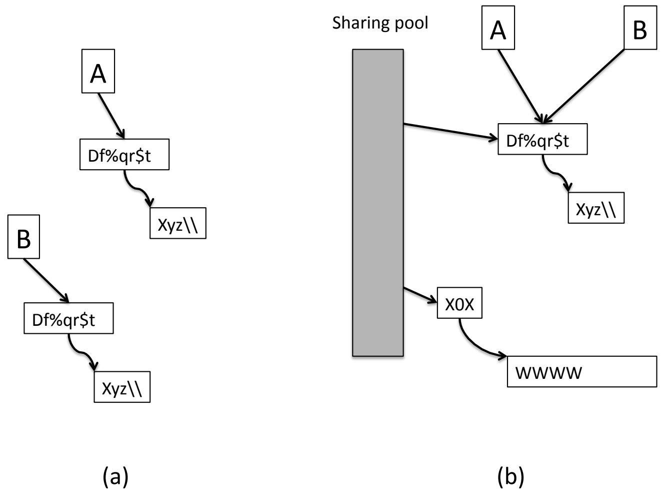


Figure 6.1. (a) Objects A and B point to duplicate data. (b) Objects A and B share the same data, stored in a sharing pool.

as shown in Figure 6.1. In Figure 6.1(a), objects A and B point to identical data structures. Figure 6.1(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 6.1(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 6.1, `A==B` is false in Figure 6.1(a) and true in Figure 6.1(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 6.1(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.

TODO: shorten this to a quick forward reference. 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 18.

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 6.2 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 `Annotations` to themselves:

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

6.6 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

COLLECTIONS: AN INTRODUCTION

Collections are the glue that bind your data together. Whether providing random or sequential access, or enabling look up by value, collections are an essential part of any design. In Java, collections are easy to use, and, just as easily, to misuse when it comes to space. Like much else in Java, they don't come with a price tag showing how much memory they need. In fact, collections often use much more memory than you might expect. In most Java applications they are the second largest consumer of memory, after Strings. It's not unusual for collection overhead to take up between x and y% of the Java heap. The way collections are employed can make or break a system's ability to scale up.

This chapter and the next three chapters are about using collections in a space-efficient way. Collections may serve a number of very different purposes in your application. You may use them to implement relationships, to organize data into tables, to enable quick lookup via indexes, or to store annotations alongside more formally modeled data. Each use has its own best practices as well as traps.

The current chapter is a short introduction to common issues that are important to understand in any use of collections. It concludes with a summary of resources that are available in the standard and some open source alternative frameworks. Each of the following three chapters then goes into depth about a specific way that collections can be used. Each of these chapters takes you through typical patterns of usage, shows what collections cost for those patterns, and gives you techniques for analyzing how local implementation decisions will play out at a larger scale. We will also look at the internal design of a few collection classes.

Chapter 8. Relationships An important use of collections is to implement relationships that let you quickly navigate from an object to related objects via references. This chapter covers the patterns and pitfalls of implementing relationships. At runtime, each relationship becomes a large number of collection instances, with many containing just a few elements. The main issues to watch for are: keeping the cost of small and empty collections to a minimum, sizing collections properly, and paying only for features you really need.

Chapter 9. Indexes and Other Large Collection Structures A collection can serve as the jumping off point for accessing a large number of objects. For example, your application might maintain a list of all the objects of one type, or have an index for looking up objects by unique key. This chapter shows how to analyze the memory costs of these structures. The main issue is understanding which costs will be amortized as the structure grows. The chapter also covers more complex cases, such as a multikey map, where there is a choice between a single collection and a multilevel design.

Chapter 10. Attribute Maps and Dynamic Records Many applications need to represent data whose shape is not known at compile time. For example, your application may read property-value pairs from a configuration file, or retrieve records from a database using a dynamic query. Since Java does not let you define new classes on the fly, collections are a natural, though inefficient way to represent these dynamic records. This chapter looks at the common cases where dynamic records are needed, and shows how to identify properties of your data that could lead to more space-efficient solutions.

7.1 The Cost of Collections

Like other building blocks in Java, the memory costs of the standard collections are high overall. The very smallest collection, an *empty* `ArrayList`, takes up 40 bytes, and that's only when it's been carefully initialized. By default it takes 80 bytes. That may not sound like a lot by itself, but when deeply nested in a design, that could easily be multiplied by a few hundred thousand instances.

Fortunately, there are some easy choices you can make that can save a lot of space. The cost of different collection classes varies greatly, even among ones that could work equally well in the same situation. For example, a 5-element `ArrayList` with room for growth takes 80 bytes. An equivalent `HashSet` costs 276 bytes, or 3.5 times as much. Like other kinds of infrastructure, collections serve a necessary function, and paying for overhead can be worthwhile. That is, as long as you are not paying for features you don't need. In the above example, a 70% space savings can be achieved if the application can do without the uniqueness checking provided by `HashSet`. In Section 2.3 we saw a similar example, achieving a large improvement when real-time maintenance of sort order wasn't needed. In the next chapters we'll see more examples of how a careful look at your system's requirements can help you reduce space.

There is also much that is not under your control in the cost of collections. Because the collection libraries are written in Java, they suffer from the same kinds of bloat we've seen in other datatypes. They have internal layers of delegation, and extra fields for features that your program may not use. The standard collections do have a few options that can help, such as specifying the excess capacity for growth. On the whole, though, they do not provide many levers for tuning to different

Collection	Fixed cost	Variable cost
Object[]	16	4
ArrayList	40	4
LinkedList	48	24
HashSet		
HashMap		
TreeMap	40	40
WeakHashMap	88	52
ConcurrentHashMap		

Table 7.1. Cost in bytes of commonly used collection classes. Minimum costs, not including extra capacity for growth. Shows arrays for comparison.

situations. They were mostly designed for speed rather than space. They seem to have been designed for applications with a few large and growing collections. Yet many systems have large numbers of small collections that never grow once initialized. Given all of this, it is important to be aware of what collections cost, so you can make informed choices as early as possible.

It is not enough to know the cost of a collection in the abstract, but to understand *how it will work in your design.* This is because some collections were only designed to be used at a certain scale. For example, a certain map class may work fine as an index over a large table, but can be prohibitively expensive when you have many instances of it nested inside a multilevel index. Collections have fixed and variable costs, as discussed in Section 2.4. The fixed cost is the minimum space needed with or without any elements; the variable cost is the additional space needed to store each element. The way these costs add up depends on the context — whether there are many small collections, a few large ones, or some combination. High fixed costs only matter when there are many collection instances. High variable costs add up when there are a lot of elements, regardless of whether the elements are spread across a lot of small collections or concentrated in a few large ones. Table 7.1 shows the costs for some commonly used collections. Collections with high fixed costs should only be used for large collections, so that the fixed cost will be amortized over a large number of elements. As a rule, `ArrayList` and other array-based collections have much lower variable costs than other collection classes.

It is helpful to look at collection costs together with the data they are storing. If a data structure uses expensive collections to store small amounts of data, than it will have a high bloat factor, and ultimately the application's ability to support a large amount of data will be limited. The next chapters show how to analyze collection costs in the context of your design. This analysis will help you choose the right collection for your design, or restructure your data into a more efficient design if necessary.

Resource	Description	Discussed in
`Collections` statics	Memory-efficient implementations of singleton and empty collections.	Section ??
	Unmodifiable, checked, and synchronized behaviors are added via collection wrappers. Can be costly if used at too fine a granularity.	Section ??
`Arrays` statics	Provides static methods if you need to create simple collection functionality from arrays	Section ?? shows one example
`IdentityHashMap`	Lower-cost map when using an object reference as key	Section ??
`EnumMap`	Compact map when keys are `Enums`	Section ??
`EnumSet`	Compact representation of a set of flags	Section ??
`WeakHashMap`	Supports one common scenario for managing object lifetime using weak references.	Section ??
Java 1 collections	Some classes in the earlier, Java 1 libraries, like Vector and Hashtable, are a lower-cost choice in some contexts where synchronized collections are needed	Section ??
`ConcurrentHashMap`	Hash map when contention is a concern. Avoid use at too fine a granularity.	Section ??

Table 7.2. Some useful resources in the Java standard library

7.2 Collections Resources

In this book we focus mostly on the standard collections. We also include some information on classes from alternative, open source frameworks that can be helpful in keeping memory costs down.

There are some lesser-known resources in the standard Java Collections framework that provide specialized functionality. Some can help you save memory if you require only those features. Others provide useful features, but can have a significant memory cost if not used carefully. Table 7.2 is a guide to the resources we discuss in this book.

In addition to the standard Java classes, there are a number of open source collections frameworks available. Some are designed specifically to improve space and time efficiency, while others are aimed at making it easier to program, adding commonly needed features not found in the standard libraries. The alternative collections frameworks can be helpful in two ways when it comes to saving memory. First, some frameworks provide space-optimized collection implementations. Second, some frameworks provide classes that make it easier to manage object lifetime. Building your own object lifetime management mechanisms, for example a concurrent cache, can be error-prone (see, for example, Section **??**). Well-tested implementations will save you a lot of effort, and can lead to better overall use of space. Keep in mind that not all alternative collection classes have been optimized for space. Some of them actually take up more space than a similar design using the standard collections. As always, there is no substitute for analyzing space costs empirically.

In this book we'll look at four of the most recent and relevant open source collections frameworks. In the next few chapters we'll look at how you can use some of these classes to solve specific problems. Table 7.3 gives a summary of the capabilities that we discuss (sometimes only briefly). This is only a sampling of what's out there. We encourage you to further explore these and other frameworks on your own.

The Guava framework, which grew out of the Google Collections, is designed primarily for programmer productivity, providing many useful features missing from the standard Java collections. Although space usage has not been the main focus, it does include some specialized classes, for example the immutable collections, that are more space-efficient than their general-purpose equivalents. Guava's `MapMaker` class provides very general support for building caches and other lifetime management mechansisms, with optional support for concurrency. While most open source frameworks provide some level of compatibility with the standard collections APIs, Guava has made compatibility a priority.

The Apache Commons Collections framework has similar objectives to the Guava framework, focusing mostly on programmer productivity rather than on efficiency per se. It does however provide some capabilities for saving memory, such as maps and linked lists with customizable storage, and specialized maps that contain just a few elements. It also provides lifetime management support through its `ReferenceMap` class, in a less general manner than the Guava equivalent. As of this writing, the Commons API has not been updated to take advantage of generics.

The GNU Trove framework has time and space efficiency as its main goal. From a memory standpoint its highlights are: collections of primitives that avoid boxing and unboxing; map and set implementations that are lighter weight than the standard ones; and linked lists that can be customized to use less memory.

The fastutil framework has similar goals to Trove, primarily time and space efficiency. Like Trove, fastutil provides primitive collections, along with lighter-weight implementations of maps and sets. Some other memory-related features include array-based implementations for small maps and sets, and support for very large arrays and collections when working in a 64-bit address space.

Feature	Supported by	Discussed in
Primitive collections	fastutil, Trove	Section ??
Lighter-weight maps and sets of objects	fastutil, Trove	Section ??
Immutable collections	fastutil, Guava	Section ??
Small collections	Commons (maps only), fastutil	Section ??
Customized linked list storage	Trove	Section ??
Maps with weak/soft references	Commons, Guava	Chapters ?? and ??
Caches	Guava	Section ??

Table 7.3. A sampling of memory-related resources available in open source frameworks

Important note: there are many kinds of open source licenses. Each has different restrictions on usage. Make sure to check with your organization's open source software policies to see if you may use a specific framework in your product, service or internal system.

7.3 Summary

Using collections carefully can make the difference between a design that scales well and one that doesn't. Some items to be aware of when working with collections:

- The standard Java collections were designed more for speed than for space. They are not optimized for some common cases, such as designs with many small collections. In general the Java collections use a lot of memory.
- Collections vary widely in their memory usage. Awareness of costs is an essential first step in choosing well. Sometimes there is a less expensive choice of collection class available, either from the standard library or from open source alternatives. Initialization options can also make a difference.
- The same collection class will scale differently depending on its context. Ensuring scalability means analyzing how a collection's fixed and variable costs play out in a given situation. Watch out for: collections with high fixed costs when you have a lot of small collections, and collections with high variable costs when you have a lot of elements.

- Analyzing your application's requirements, specifically which features of a collection you really need, can suggest less expensive choices.

Chapter 8

RELATIONSHIPS

Relationships in an entity-relationship model are typically implemented in Java using the standard library collection classes: each object points to collections of other objects related to it. Since relationships are at the core of any data model, it is not uncommon for a Java application to create hundreds of thousands, even millions, of collections. Therefore, simple decisions, like which collection class to use, when to create it, and how to initialize it, can make a surprisingly big difference on memory cost. This chapter shows how to lower memory costs when implementing relationships with collections.

8.1 Choosing The Right Collection

The standard Java collection classes vary widely in terms of how much memory they use. 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. This section looks at what to consider when choosing a collection to represent a relationship target.

Using collections for relationships often results in many small or empty collections, because there are either lots of objects in the relationship or many objects are related to a only few other objects, or both. When there are lots of collections with only a few entries, you need to ask whether the functionality of the collection you choose is worth the memory cost of that functionality.

To make this discussion more concrete, let's return to the product and supplier example from section 4.1, and change it a little bit. Instead of only one alternate supplier, a product now may have multiple alternate suppliers, and each product stores a reference to a collection of alternate suppliers. An obvious choice is to store the alternate suppliers in a `HashSet`:

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

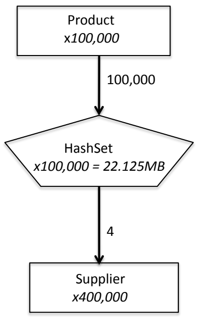


Figure 8.1. A relationship between products and alternate suppliers, stored as a `HashSets` of alternate `Suppliers` related to `Products`.

```
        ..
        HashSet<Supplier> alternateSuppliers;
    }

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

Suppose there are 100,000 products that each have four alternate suppliers on average. Figure 8.1 shows an entity-collection diagram for the relationship between products and alternate suppliers.

Using a `HashSet` for alternate suppliers turns out to be a very costly decision. The alternate suppliers are represented by 100,000 very small `HashSets`, each consuming 232 bytes, for a total cost of 22.125MB. This cost is all overhead. It's hard 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 is very, very common. 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 little extra checking code,

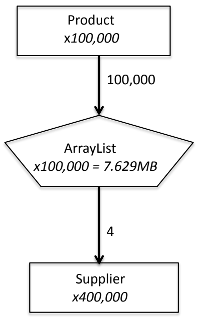


Figure 8.2. A relationship between 100,000 products and alternate suppliers, where the alternate `Suppliers` associated with each `Product` are stored in an `ArrayLists`.

and usually without significant performance loss when sets are small. Figure 8.2 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.

8.2 The Cost Of Collections

Let's look inside a `HashSet` to see why it is so much bigger than an `ArrayList`. Some of its extra cost is because of additional functionality, such as entry uniqueness and constant-time access. Other extra cost because of unavoidable Java overhead, and because `HashSet` is optimized for performance: the `HashSet` implementation assumes `HashSets` will be very large, and sacrifices memory space in favor of performance. Additionally, `HashSet` reuses the `HashMap` implementation, which is expensive memory-wise. `HashSet` is implemented as a degenerate `HashMap`, that is, a `HashMap` with no values.

The internal structure of a `HashSet`, shown in Figure 8.3, consists of wrapper objects, an entry array, and linked lists of entries. Most collections have similar kinds of internal components.

Wrapper Objects. The `HashSet` object itself is just a wrapper, delegating all of its work to a `HashMap`. Therefore, there is an additional wrapper, a `HashMap` object.

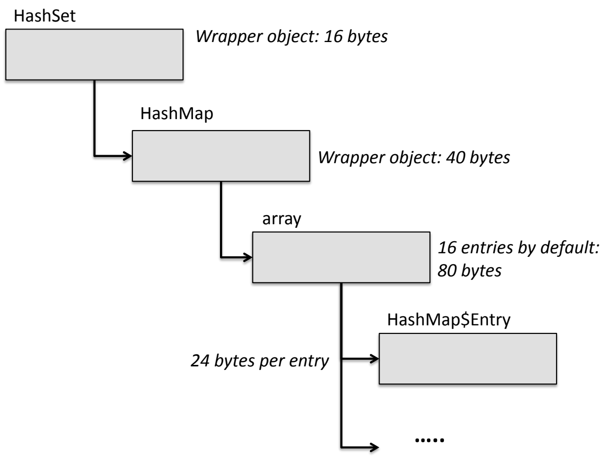


Figure 8.3. The internal structure of a `HashSet`.

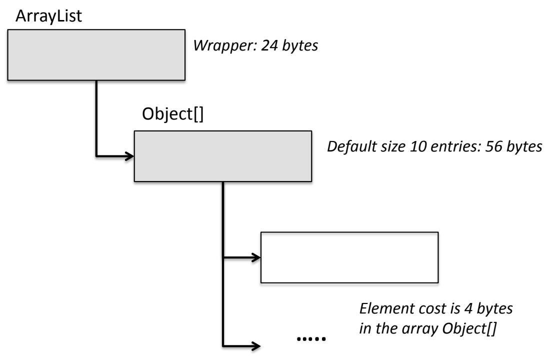

ArrayList
Wrapper: 24 bytes
Object[]
Default size 10 entries: 56 bytes
Element cost is 4 bytes in the array Object[]
.....

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

All collections have wrapper objects, but a `HashSet` has two of them.

Entry Array. Since this is a hashing structure, there is an array to store the hashed entries. The default initial size of the array is 16.

Entries. `HashMap` stores all entries in clash lists, so there is a `HashMap$Entry` object for each `HashSet` entry. The size of a `HashMap$Entry` is 24 bytes.

In contrast, `ArrayList` has a smaller fixed cost and a smaller variable cost than `HashSet`. It is really just an expandable array, consisting of a wrapper object and an array of entries, as shown in Figure 8.4. 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` inflates its fixed cost. Lower variable cost means that `ArrayList` scales much better than `HashSet`. The variable cost of an `ArrayList` entry is an entry array pointer, which is 4 bytes. The variable 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 8.1 shows the memory costs of four basic collections, `ArrayList`, `LinkedList`, `HashMap`, and `HashSet`. This table shows the default size when the collection is just allocated without entries, and any additional entry cost. These costs have been calculated based on the Sun JVM, using the techniques described in Chapter 3.

Collection	Fixed Default Cost with No Entries	Variable Cost
ArrayList	80 bytes (capacity for 10 entries)	4 bytes
LinkedList	48 bytes	24 bytes
HashMap	120 bytes (capacity for 16 entries)	28 bytes
HashSet	136 bytes (capacity for 16 entries)	28 bytes

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 variable cost, the memory needed for an entry, indicates scalability.

The various other Java standard library implementations in circulation have costs similar to these. You can calculate them using the same methodology.

Our guess is that the collection class developers would be surprised by the relationship usage pattern that results in hundreds of thousands of small `HashSets`. 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 overhead cost of a `HashSet` is remarkably high. Creating many small collections multiplies this basic infrastructure cost, which is all overhead, filling the heap.

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, collections used to represent relationships may 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.

Fortunately, it is possible to right-size `ArrayLists`. If you know that an `ArrayList` has a maximum size x, which is less than the default size, then it's worth passing x as a parameter to the constructor to set 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,

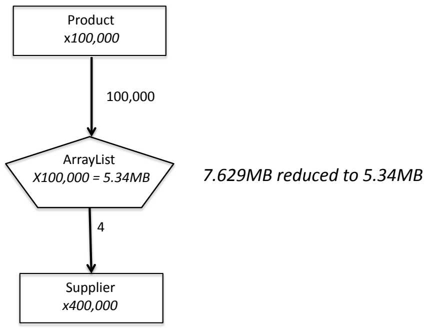


Figure 8.5. The relationship between `Products` and `Suppliers` after all of the `ArrayLists` have been trimmed by calling the `trimToSize` method.

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, in which case `ArrayLists` can be trimmed between these two phases, so that the cost of reallocation and copying is paid only once.

Returning to the example of the relationship between products and alternate suppliers, the `ArrayLists` in Figure 8.2 have default capacity. If we assume that the the relationship is built in one phase, and used in another phase, then it is possible to trim the `ArrayLists` after the first phase. This should save quite a bit of space, since there are 100,000 `ArrayLists` with four entries on average. In fact, trimming these `ArrayLists` saves 2.29MB, as shown in Figure 8.5.

`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.

8.4 Avoiding Empty Collections

Too many empty collections is another common problem that leads to memory bloat. A quick look inside an empty collection shows that it is not all that empty.

bytes, assuming a default initial size. Even if the default initial size is overridden and set to 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 relationship in Figure **??** is initialized by the code:

```
ArrayList<Product> products;
int numProducts;
   ..
   public void initAlternateSupplierRelationship() {
       ..
       for (int i = 0; i < numProducts; i++) {
          Product product = products.get(i);
          product.alternateSuppliers =
                            new ArrayList<Supplier>();
       }
   }
```

Initially, each product is mapped to an empty `ArrayList` for alternate suppliers, so there are 100,000 empty `ArrayLists` before any `Suppliers` are inserted. As the alternate suppliers are populated, many of these `ArrayLists` will become non-empty, but it is likely that a good number of products have no alternate suppliers. If 25% of the products in the final graph still have no alternate suppliers, there will be 25,000 empty `ArrayLists`, which consume .95MB even after calling `trimToSize`. Figure 8.6 shows the entity-collection diagram after removing 25,000 empty alternate supplier `ArrayLists`. Note that there are now only 75,000 alternate supplier `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`.

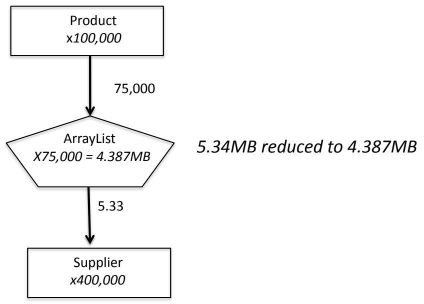


Figure 8.6. The relationship between `Products` and `Suppliers` where there are no empty `ArrayLists`.

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

```
public void initAlternateSupplierRelationship() {
  ..
  for (int i = 0; i < numProducts; i++) {
     Product product = products.get(i);
     product.alternateSuppliers =
                          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 their immediate context. If you give out a reference to a static empty collection, then there is 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.

In our example, alternate suppliers are stored in an `ArrayList`, which inside is a wrapper pointing to an array of `Suppliers`. If we assume that every product has at most four alternate suppliers, then it isn't necessary to store these in an `ArrayList` — a simple array will do. Eliminating the `ArrayList` object removes 24 bytes per product, but we have to add 4 bytes to each `Product` to store the number of alternate suppliers. The total savings is 1.43MB for 75,000 products:

```
class Product {
   String sku;
   String name;
   ..
   int numAlternateSuppliers;
   Supplier[4] alternateSuppliers;
}
```

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 `Product` class. Namely, making the four alternate suppliers into fields of `Product` instead of elements of an array. This optimization a 32 byte array object for 75,000 products, while adding three additional fields to the `Product` for 100,000 objects, saving another 1.1MB in total:

```
class Product {
    String sku;
    String name;
    ..
    Supplier alternateSupplier1;
    Supplier alternateSupplier2;
    Supplier alternateSupplier3;
    Supplier alternateSupplier4;
}
```

This representation is very similiar to the original `Product` class in section 4.1 where there is one alternate suppler field, and here there are four. Taking all of the optimizations together, we have gone from the initial `HashSet` representation of 22.125MB in Section 8.1 to the in-lined field representation of 1.86MB.

8.6 Hybrid Representation

An interesting case is when the sizes of the collections used in a relationship is not uniform. That is, some collections are small and some collections are very big. It's reasonable to use an expensive collection like `HashSet` for the big collections, but then the small collections pay the price. One way to handle this problem is to use a hybrid representation. For example, you can use arrays for smaller collections, and `HashSets` for larger collections.

One catch is that usually you will not know in advance which collections in the relationship will end up being small and which will grow to be large. Therefore a conversion operation will be necessary at some point if a collection grows large enough. Each collection starts as an array, and then once it grows past a threshhold, it is converted to a `HashSet`.

In our example, suppose that the vast majority of products have four alternate suppliers on average, but a few products have over 100 alternate suppliers. We can choose a threshhold, say six entries, which triggers a conversion to a `HashSet` when the threshhold is exceeded. Here is the class `Product`:

```
public class Product {
```

```
static final int threshhold = 6;
String sku;
String name;
..
int numAlternateSuppliers;
Supplier[] alternateSuppliers;
HashSet<Supplier> bigAlternateSuppliers;

void addAlternateSupplier(Supplier supplier) {

    /* Check for duplication */
    if (hasAlternateSupplier(supplier)) {
         return;
    }

    /* Try to add the supplier to the array
        alternateSuppliers */
    if (numAlternateSuppliers == 0) {
        alternateSuppliers = new Supplier[threshhold
            ];
    }
    if (numAlternateSuppliers < threshhold) {
        alternateSuppliers[numAlternateSuppliers++] =
             supplier;
        return;
    }

    /* If threshhold is exceeded, need to use
        bigAlternateSuppliers */
    if (numAlternateSuppliers == threshhold) {
        bigAlternateSuppliers = new HashSet<Supplier
            >();
        for (int i = 0; i < threshhold; i++) {
            bigAlternateSuppliers.add(
                alternateSuppliers[i]);
        }
        alternateSuppliers = null;
    }
    bigAlternateSuppliers.add(supplier);
    numAlternateSuppliers++;
    return;
}
```

```
    }
```

The method `addAlternateSupplier` adds the supplier to the array if the size is less than the thresshold, otherwise, it adds the supplier to the `HashSet`. It allocates the alternate supplier array and `HashSet` only if and when it is needed, to avoid wasting space with empty collections when there are no or only a few alternate suppliers.

Implementing hybrid representations is more complicated than just using one collection for a relationship. However, it can save significant space in some cases.

8.7 Summary

Collections used to represent relationships often result in many small collection instances, whose cost is dominated by a fixed-size overhead. This chapter describes a number of ways to mitigate high fixed memory costs for relationships implemented with 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 collections in advance. You can also eliminate an `ArrayList` altogether when there are always at most a few entries that can be stored in fields.
- When the usage pattern is not uniform, it is sometimes reasonable to use a hybrid representation.

Knowing which relationships and collections in your application are the most important and need to scale is key to applying these optimizations effectively.

Chapter 9

LARGE COLLECTION STRUCTURES

In a relational database system, data is neatly organized for you into tables. You may suggest fields to index, and the details of indexes are taken care of for you. In object-oriented programming languages you have more freedom. You have a sea of interconnected objects, and you are responsible for designing structures that are the entry points into your data. Collections, especially large collections, are your main tool. In this chapter we look at the memory considerations when designing large collection structures for accessing your data. We look briefly at structures that just gather data in one place, such as a list of all the objects of one type. The bulk of the chapter is about indexes that let you look up data by value. Maps are the main implementation mechanism in these structures [1].

We'll first look at the costs of large collections, and some ways to keep them to a minimum for the task. There are also some open source collection classes that can save you a lot of space, if you are able to use them in your system. We'll next look at a few special cases that enable you to use specialized, and less expensive, solutions. The remainder of the chapter is about analyzing the space costs of more complex, multi-level collection structures. We look at three common cases: an index with a multipart key, a multilevel concurrent index, and an index with multiple values per key. There are more decisions to make when designing these structures, and their costs are not obvious without a careful analysis.

9.1 Large Collections

Just as with small collections, choosing the right collection for the task can make a big difference in the memory overhead of large collections. Suppose you need to maintain a collection of all the orders processed for the day. At the end of the day these orders are posted in bulk to a remote database. The orders contain their own timestamps, so we don't really care about maintaining the sequence in which they were received. Figure **??** compares an implementation using `ArrayList` with one

[1] We use the term map to mean a Java collection class, and the term index to describe the general functionality.

using `HashSet`. As with small collections, if we don't need the uniqueness checking or some of the other features of `HashSet`, then we are clearly much better off with `ArrayList`.

In larger collections, the variable overhead — the cost each element incurs — determines the cost of the collection. That's because as a collection grows in size, its fixed overhead, such as wrapper objects or array headers, becomes insignificant. For example, for an `ArrayList` with even 1000 elements, the fixed cost is just .1% of the total. Whereas with small collections we need to concern ourselves with both fixed and variable costs, for large collections we need only look at variable costs. The difference in size in the above example is pretty dramatic, more than 7:1. That reflects the difference in variable costs of the two collection classes. As a general rule, the variable overhead of array-based collections, like `ArrayList` is much lower than that of entry-based collections, like `HashSet`, where a new entry object is allocated for each element in the collection. Table **??** shows a comparison of variable costs.

Excess Capacity As we saw in the previous chapter, many collection classes allocate excess capacity for performance reasons, mainly to accomodate growth. One difference between small and large collections is the effect of excess capacity on the overall memory overhead.

As a collection grows, the amount of spare capacity allocated is usually based on the number of elements in the collection. For example, `ArrayList` reallocates an array that is 50% larger when it needs extra space, and `HashMap` doubles the number of buckets when the number of elements reaches a user-specified load factor. So for large collections, we can think of the spare capacity as an additional charge on each element of the collection, an addition to the variable cost. Since collections grow in jumps, rather than in single steps, it makes sense to look at this cost on average across many elements, rather than as a cost on the last element we added. We'll use the term *average variable cost* of a collection to mean the variable cost plus its share of the excess capacity. .. examples here .. include example of a properly sized ArrayList? .. average is just an estimate - can be higher or lower in practice, depending upon the exact size of your collection.

For large array-based collections, since their total overhead is so much smaller to begin with, the excess capacity can be significant. So from a relative standpoint, there is more benefit to be found in reducing it. For `HashMap`, `HashSet`, and similar collections, since the bulk of the overhead is from the entry objects, reducing the excess capacity (which is reflected in the size of the array allocated) will have a much smaller effect on the size. In addition, reductions are limited by the need to avoid collisions, so there's not much point in skimping on capacity in these structures.

Solutions for reducing excess capacity for large collections are the same as for small collections. If you can estimate the number of elements in advance, you can try to size the collection carefully when you create it. If your data structure has a distinct load phase, and the collection has a `trimToFit()` call, you can trim the size of the collection after the load phase is complete. In hash-based collections, such as `HashMap` and `HashSet`, excess capacity is needed to reduce the likelihood of collisions, so it's important to leave headroom for this purpose. The default load factor is usually a pretty good guide.

Entry- vs. Array-based Maps and Sets There are two primary ways that hash tables are implemented. Most of the maps and sets in the standard libraries use a technique known as *chaining* (also known as open hashing), Figure **??** shows a typical implementation. There is a separate entry object for element in the collection. Each entry object points to a key and a value, and possibly contains additional information such as a cached hash code. There is a linked list of entry objects for each hash bucket.

The other main technique is known as *open addressing* (for added confusion, also known as closed hashing). In this technique, keys, values, and other information for each element are stored directly in arrays. These are usually parallel arrays, though some implementations use a single array and interleave keys, values, etc. in successive slots. Chains of entries that map to the same bucket are threaded through the array(s). Figure **??** shows a typical implementation. There are many variations in practice.

Generally speaking, for larger collections, open addressing hash tables use less memory — at least in Java, where the cost of an object is so high. The fastutil and Trove open source frameworks provide maps and sets that save space by using open addressing. An added advantage in both of these frameworks is that sets are specialized for that purpose, rather than delegating their work to a map. So unlike in the Java standard libraries, you are paying only for a value, not a key and value, per entry.

Since open addressing hash tables usually use less memory, why do so many hash table libraries use chaining? Generally speaking, hash tables based on chaining are much simpler to implement. More importantly, there are performance differences between the two approaches, though there is no easy rule about one always being faster than the other. For your own system, if performance of your collections is critical it can be worth doing some timings first. Keep in mind, though, that in many systems, the amount of time spent in hash table lookups is a small fraction of the total execution time to begin with.

Another thing to be aware of in open addressing implementations is that the need for excess capacity can reduce the space savings to a greater extent than in open chaining implementations. So it's important to look at the whole picture, at the average variable cost, when making decisions on which framework to use (see Table **??**).

Sharing the Costs of Entries in Entry-based Collections

9.2 Identity Maps

9.3 Maps with Scalar Keys or Values

9.4 Multikey Maps. Example: Evaluating Three Alternative Designs

9.5 Concurrency and Multilevel Indexes

9.6 Multivalue Maps

9.7 Summary

Chapter 10

ATTRIBUTE MAPS AND DYNAMIC RECORDS

Part II

Managing the Lifetime of Data

Chapter 11

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 11.1 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.

11.1 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 11.1. Common requirements for the lifetime of objects that your application must implement properly, in order to avoid correctness or performance problems.

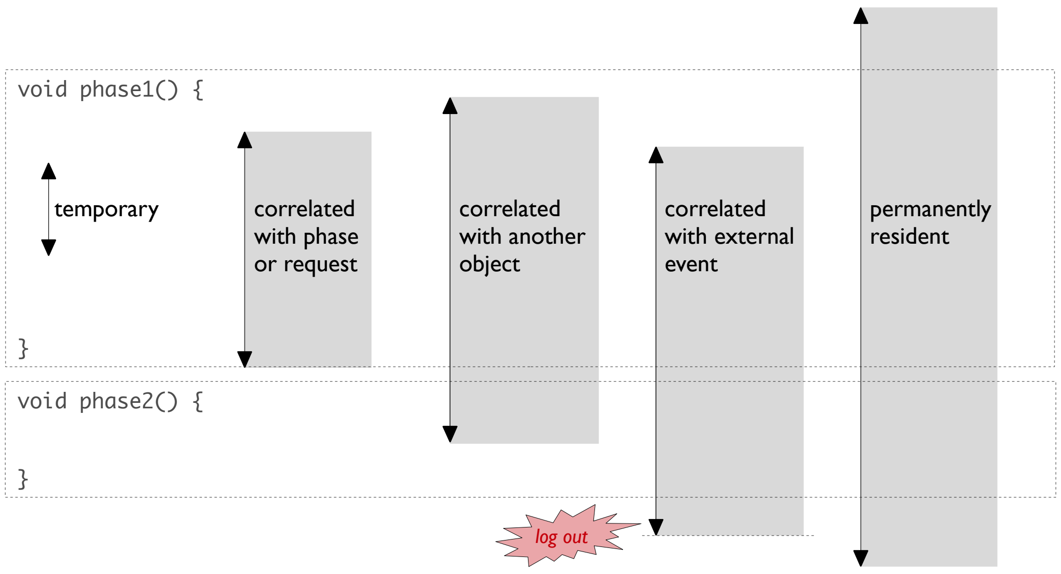

void phase1() {
temporary
correlated with phase or request
correlated with another object
correlated with external event
permanently resident
}
void phase2() {
}
log out

Figure 11.1. 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.

11.2 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.

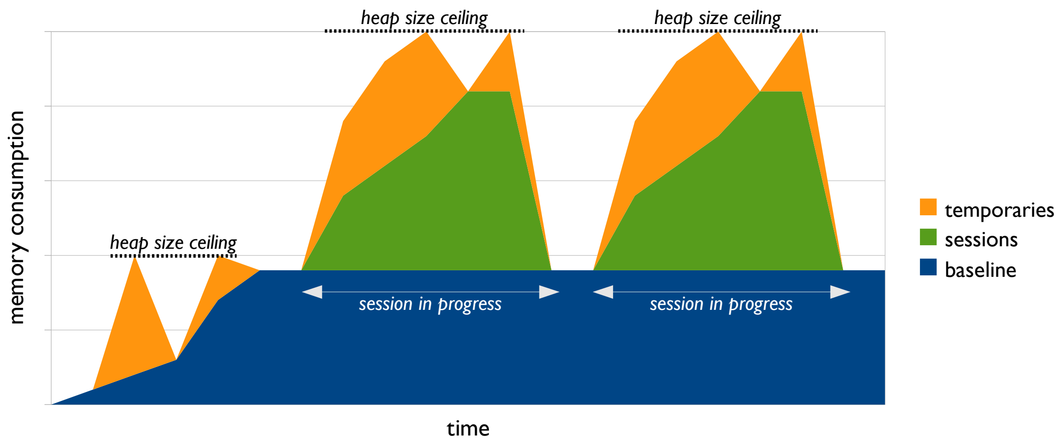


Figure 11.2. Memory consumption, over time, typical of a web application server.

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).

11.3 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 11.3 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.

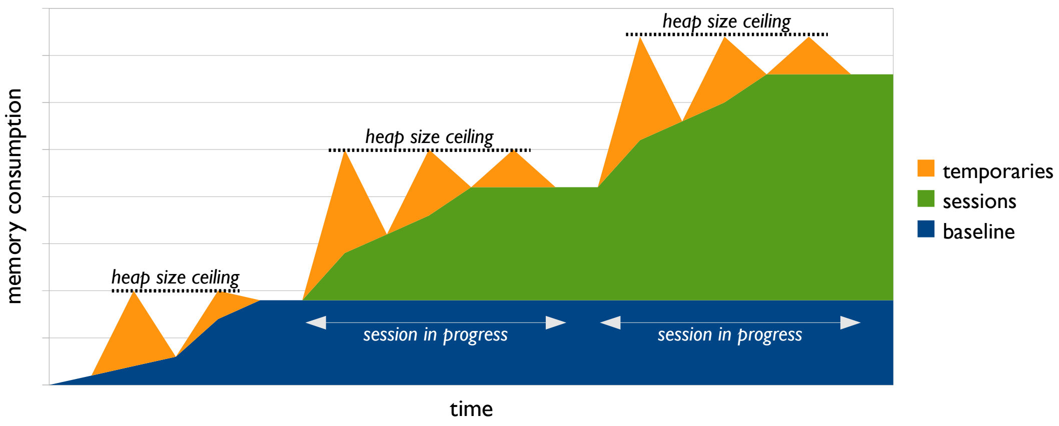


Figure 11.3. If session state is not cleaned up properly, a memory leak is the result. This means more frequent garbage collections, and ever increasing heap size.

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.

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.

11.4 Time-space Tradeoffs

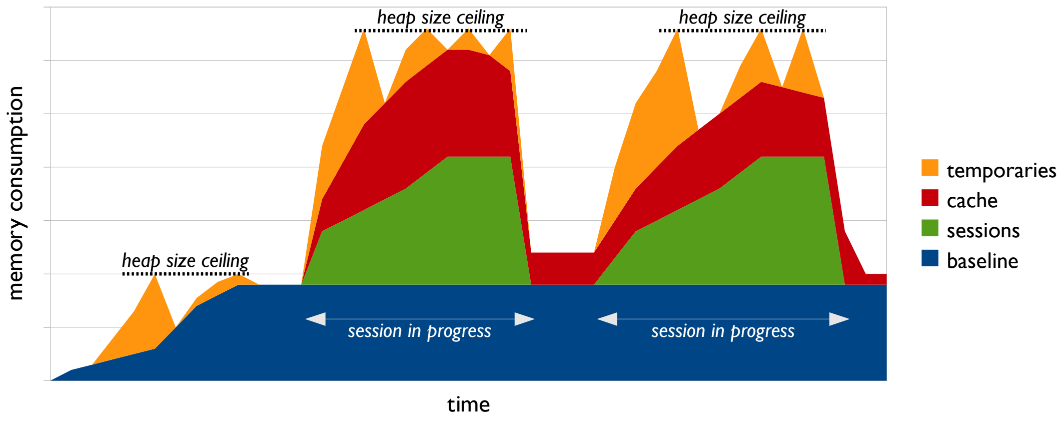


Figure 11.4. When a cache is in use, this leaves less headroom for temporary object allocation, often resulting in more frequent garbage collections.

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 13 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 11.4 shows an example where the cache has 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 19.2 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.

11.5 Permanently Resident

Figure 11.2 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 18 discusses strategies for coping with this situation.

11.6 Summary

Chapter 12

MEMORY MANAGEMENT BASICS

The Java language has a *managed runtime*. As part of being a managed runtime, as a Java program runs, a supporting JIT compilation and other support threads are spawned. These thread work on your program's behalf and, together, compose a runtime system that manage important aspects of execution. One of these tasks is memory allocation and reclamation. The runtime automatically takes care of many important cases of memory management. You needn't, for example, be concerned with explicitly deallocating *most* of the objects you allocate, because the runtime includes automatic garbage collection of instances. The Java runtime also provides built-in support that let you manually take care of some of the more complex cases, such as implementing appropriate caching policies.

Taking advantage of these facilities requires some care, from issues as straight-forward sounding as choosing a reasonable bound for memory consumption, to deciphering the cause of failures due to memory exhaustion. Surprisingly, memory leaks are possible, even common, in Java, and can easily lead to application failures without good design and testing. Furthermore, some of the runtime facilities appear in the form of low-level JVM hooks, or implicit behavior that you have to carefully govern, and so require careful coding to make correct use of them.

12.1 Heaps, Stacks, Address Space, and Other Native Resources

As your program runs, there is a good chance that quite a bit of memory will be allocated and reclaimed. Some of the memory allocations will come directly from your code, as it calls `new` to instantiate classes. Other times, memory will be allocated behind the scenes, such as by the class loading or JIT compilation mechanisms, or by native code that your code uses. Under the covers, the managed runtime will service allocation requests from a number of memory pools. Most of these will be allocated on the *Java heap*, but there are other memory areas that you need to be aware of. Each area has its own sizing and growth policies, and its own constraints on maximum capacity.

The Heap Usually, a call to `new` results in a memory allocation on the Java heap. The heap is a region of memory that the JVM allocates on startup, sized to your specification. It contains Java objects, linked together in the way that they reference eachother. In Java, each object can contain either primitive data or references to other objects. One Java object cannot contain, inline, the fields another Java object. This is possible in languages such as C or C# through the use of `struct`s.

In Java, the heap is bounded in size, and so it will never consume more than a fixed amount of memory. The maximum size of the heap, along with the initial size and a policy for growing the heap, are all under your control. If you don't make a choice for any of these, the JRE will choose some reasonable defaults. Always experiment to see whether the defaults suit your needs. The defaults vary, from one JRE to the next. Older JREs, typically those prior to Java 5, tend to have hard-coded default values, indepdendent of how much physical memory your machine actually has. Most JREs now attempt to adjust the the heap sizing criteria based on the physical memory capacity of your machine.

It often makes sense to keep a small initial Java heap size and a large maximum size. If your application requires a varying amount of memory, an amount that depends on the size of the input data, then this strategy can pay off. For example, say you run multiple copies of the application on one machine, and, most of the time, the inputs are on the small side. Then this strategy will allow you to run more copies simultaneously, while still allowing for the rare cases when an input requires more memory. You needn't worry too much about this affecting performance for the cases of larger inputs. JREs are pretty smart these days, and make a good effort to adjust the size of the heap from the initial size to the maximum size, and back down, as it finds that your application memory needs change. Still, you should always experiment! JREs don't always get this right.

Experimentation is also important, because the default choice of initial and maximum Java heap size may result in your application running more slowly than it could. Sometimes, this sub-par performance is due to having an *initial* size that is too low. It takes a while for the JRE to learn that it should increase the heap size beyond the initial size. If your application only runs for a short time, and your application commonly needs more than the default initial size, why should you burden the JRE with learning something that you have already learned by your own experimentation?

In terms of finding a good maximum heap size, you should be very careful never to set it to be more than the amount of physical memory on your machine. If your application runs as multiple processes, you have to make sure to divide physical memory between them. Also, the operating system itself, and other processes running on the machine will consume physical memory, taking away from the resources available to your application. Further complicating matters, behind the scenes aspects of the managed runtime will consume resources even within your process. This practice of continual experimentation is important, given the disparity between the

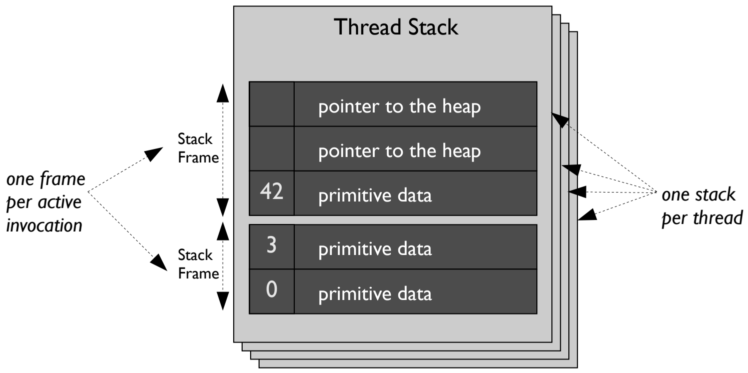


Figure 12.1. The compiler takes care managing the stack, by pushing and popping the storage (called *stack frames*) that hold your local variables and method parameters.

speed of accessing RAM versus the speed of accessing disk; these days, swapping memory to and from disk is too slow to be worth even considering. In particular, the way that the managed runtime reclaims memory, via automatic garbage collection, usually doesn't mix well at all with swapping memory to and from disk.

To specify the initial size of the Java heap, pass the `-Xms` flag on the launch command line; to specify the maximum size, pass the `-Xmx` flag. For example, by passing `-Xms100M -XmX1G`, you are telling the JRE to use an initial size of 100 megabytes, and a maximum size of 1 gigabyte. Chapter 15 goes into more detail on the tuning parameters at your disposal.

```
void f() {
   X x = new X();
   X y = new X();
   int z = ...;
   ...
}
```

The Stack The Java heap is the main storage for your objects, including all of their fields and primitive data. When your code has a local variable that references an object, this pointer is stored on the *stack*. In the example on the right, for the duration of an invocation of the method `f`, your code needs to store references to those two instances of `X` and the primitive data value `z`. The JIT compiler sets aside space for these three local variables on the stack, as illustrated in Figure 12.1. Each method invocation, in each thread, has memory associated with it to store these local variables. This space is pushed and popped, as the thread invokes and returns from the execution of methods. The stack memory that is set aside for a method invocation is commonly called a *stack frame*.[1] As is the case with Java objects, it is also the case that the stack can only contain primitive data or references to objects.

Like the heap, the stack also has limits to its size. Each thread in your program can have a stack no deeper than a fixed limit, in bytes. If, during the execution of a thread, these stack limits are exceeded, you may see a `java.lang.StackOverflowError`

[1]Historically, it has also been referred to as an *activation frame* or *activation record*.

exception thrown. You can configure this property via the `-oss` command-line parameter.

A JIT Optimization: Stack Allocation Sometimes, the JIT compiler is clever enough to observe that a call to `new` can safely be allocated on the stack. This optimization is called *stack allocation* of objects. It is enabled, by default, as of Sun Java 6 Update 21 and IBM Java 6 Service Release 2. When this optimization is possible, objects created and used *only* as local variables will be stored on a stack frame; it is as if you were using C `structs`. Though the JIT compiler can optimize away a fair number of short-lived objects via stack allocation, you should not depend upon this optimization. It is a tricky thing for the JIT compiler to do correctly, and so the compiler is very conservative in its application.[2]

Java Memory vs. Native Memory In addition to the heap and stack space that are devoted to Java data, every Java program also has separate memory areas devoted to native data. The native heap stores a variety of things, including memory allocations made by native code that your application uses and the JIT compiled code of your application. The JRE imposes an upper limit on the Java heap. In contrast, the operating system sometimes imposes no limit on the total amount of memory consumed by a process. Therefore you should be careful, and observe whether your application is indeed swapping any native allocations to and from disk.

In addition, every thread in a Java program actually has two stacks, one for the Java stack frames and one to hold the stack frames of any native methods that either your Java code invokes or invokes your Java code. There is a command-line parameter that you can use to configure the size of every native stack: `-ss`.

When your application exceeds any native limits you may confusingly see the same error that you would see for exhaustion of Java memory resources: the `OutOfMemoryError`. Therefore, you should be aware of the Java and native heap limits, in order to confirm the true source of the resource exhaustion: was it due to running out of Java heap, or running out of native resources?

Physical Memory versus Address Space Physical memory is one of the primary underlying constraints on how big you can size your heaps. There is another limit that is independent of how much physical memory you install on your machine. The limit depends on the number of memory locations that can be addressesed, given the size of pointers on your machine. The *address space* of a process is the set of addresses that the process can read or write via pointers. Therefore, this limit depends upon the size of pointers on your platform, and, to a lesser extent, upon the underlying operating system. Table 12.1 gives numbers for some common platforms. For example, if your application runs on a 32-bit Windows operating system, the

[2]If you are curious, the best you can do is to analyze performance both with and without the underlying anlaysis. On Sun JVMs, you can disable the analysis by adding `-XX:-DoEscapeAnalysis` to your application's command line.

Platform	Pointer Size	maximum memory consumption per process
z/OS	31-bit	1.3GB
Microsoft Windows	32-bit	1.8GB
UNIX-based	32-bit	2GB
Microsoft Windows	32-bit `/3GB`	3GB
all	64-bit	$\geq$ 256TB

Table 12.1. Even with plenty of physical memory installed, every process of your application is still constrained by the limits of the address space. On some versions of Microsoft Windows, you may specify a boot parameter `/3GB` to increase this limit.

total amount of memory that each process can access is 1.8 gigabytes.[3] A pointer that is 32 bits wide can address 4 gigabytes, of which the operating system reserves roughly half for its own use. If a process of your application attempts to allocate memory beyond this address space limit, a failure will occur. It is quite often the case that this failure will manifest itself also as a `java.lang.OutOfMemoryError`. Since both Java heap exhaustion and address space exhaustion can manifest as the same error, you will need to dig down to root out the true nature of the failure.

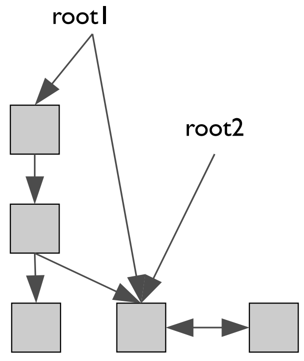
Some operating systems let you specify limits on the amount of address space that a process can use. This is similar to the way that you can specify `-Xmx` to limit the maximum amount of Java heap that a process should consume. On UNIX platforms, you can use the `ulimit` command. For example, on Linux, to limit the amount of address space any process launched from the current shell can access, say to 1 gigabyte, issue this command in Figure 12.2.

```
% ulimit -m 1048576
```

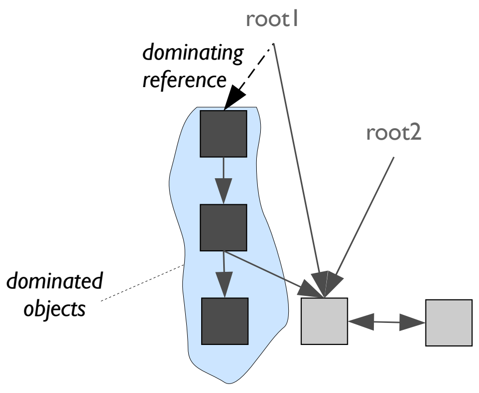
Figure 12.2. Limiting the addressible memory of a process to 1 gigabyte.

Native Resources: File Descriptors, etc. Address space constraints exist on every operating system. Depending on your operating system, there are often other resource limits that can lead to program failures. For example, processes may be limited in the number of open files they may have at any one time. You may see failures in your application, despite having lots of memory and stack space free, because an application process has exhausted file descriptors. On Windows platforms, there are other system-imposed limits, such as the number of open font handles. You should be aware of these common resource constraints, because they may impact the scalability of your application. For example, you may have designed your application to keep many font handles open for each thread, rather than keeping a common pool of them, and sharing them across threads.

[3]Some versions of 32-bit Windows let you specify a `/3GB` boot option, which increases this limit from 1.8GB to 3GB.



(a) A live data structure.



(b) Then, the dominating reference is clipped.

Figure 12.3. 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.

12.2 The Garbage Collector

Garbage collection is the mechanism that determines when the memory allocation of an object can be reclaimed for future use. To determine whether an object is dead, the garbage collector looks at the structure of object interconnections in the heap. Any objects that future code cannot possibly access are certainly ready to be reclaimed.

Your application uses objects by traversing references to them. For example `o.f` gives access to the contents of the object referenced by field `f` of instance `o`. Your code has to start this chain of field references somewhere, of course, and the garbage collector simulates this process. It traverses references from all possible variables that your code might possibly access. Each time a garbage collection occurs, the collector scans the heap for *live* objects in this way. A live object is one that might possibly be used in the future.

Reachability The collector treats the heap as a graph of objects. The nodes are the objects themselves, and the edges are the non-null fields and non-null entries of arrays. Liveness is a recursive concept: an object is *live* if it is referenced either by a live object or, in the base case, by a *root*. The roots of garbage collection 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 is ready for collection.

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 12.3 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 12.3(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 clipped reference is said to dominate those objects that it uniquely owns.

Dominance (Unique Ownership)

One node in a graph *dominates* another if the only way to reach the latter is by traveling through zero or more nodes from the former. This property is important for determining which other objects will be garbage collectable when an object is reclaimed: those it dominates. If an object only dominates itself, but isn't a root of the graph, then you know that it has *shared ownership*. Otherwise, it is *uniquely owned* by that dominating node.

Most of the time, the garbage collector does what you'd expect, and you needn't worry about freeing up memory. You will learn about bugs, such as memory leaks, and cases where memory allocations *drag* out their lifetime, beyond when your code needs them. In these, and other more complex cases, such as those involving caching or the use of native resources, things can get tricky. For these cases, you need to be aware of the how the managed runtime treats an object in various stages of its life.

Running Out of Java Heap If your application exhausts the Java heap, you will observe an `OutOfMemoryError` exception thrown. On Sun JVMs, you will sometimes see a variant of this: `GC overhead limit exceeded`. On all JVMs, it is possible that the runtime will enter a nasty state where it is spending all of its time reclaiming memory.

GC Death

Sometimes, your application will appear to grind to a halt, without any exceptions appearing on the error console. If you observe this situation, then your application may be suffering from what is affectionately called *GC death*. In this situation, the JRE is frantically, but ineffectually, trying to find free space. The situation becomes dire when the time it takes the garbage collector to perform one scan is large relative to the time before your application next runs out of space.

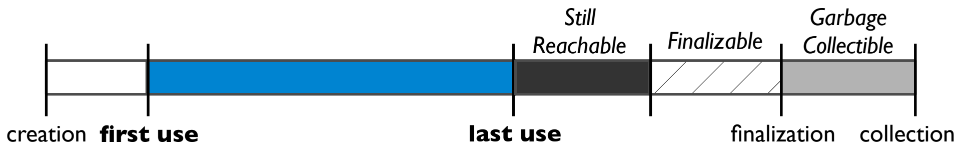


Figure 12.4. Timeline of the life of a typical object.

It is easy to know whether your application is suffering from GC death. A universal way to do this, one that works on any JVM, is to enable verbose garbage collection. This requires modifying your application's command line, and therefore restarting it, if this option is not already enabled. Each JRE provider offers good alternatives to verbose garbage collection. If you are using a Sun JDK build of the JRE, then you can use the `jstat` tool.[4] If you are using an IBM JRE, then you can request `javacores`, which contain a small historical window of garbage collection events. Neither of these require modifying your application command line, and so are very nice alternatives. In any case, by inspecting the resulting output, you will observe that the collector is active 99% or more of the time.

12.3 The Object Lifecycle

Memory in a managed runtime goes through a complex lifecycle, from allocation to eventual reclamation. In contrast, memory in the C language has a very simple lifecycle. In C, memory is allocated by calls to `malloc` and reclaimed by explicit calls to `free`. For C, that's all there is to it: from your program's perspective, a piece of memory is either in use or it isn't. In Java, objects go through many more stages.

In a well-behaved application, an object's lifetime starts with its allocation, continues with the application making use of it, and concludes with the (hopefully) short period during which the JRE takes control and reclaims the space. Figure 12.4 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();
  parser.parse(string, pos);
  ...
}
```

[4]The `jstat` tool is only available with JDK builds. It is neither available with JRE builds, nor with any builds prior Java 5.

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 behind the scenes. 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.

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*.

An object can either:

- **Drag forever**. This can happen when you allocate and use a data structure at application startup, but never again.
- **Drag with lexical scope**. This can happen when a data structure is allocated early on in a method invocation, but not used after some early point in the execution of that method.
- **Drag until GC**, i.e. the next time the JRE decides to scan the heap, looking for dead objects. This can be a problem if your application creates temporary objects at a low rate, or if you have configured a very large heap. In either case, the next garbage collection may be far in the future.

In the date formatting loop above, 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. The object now enters a second stage of this limbo. There are no pointers to `pos` that should keep its memory around, but the memory will stay around until the next garbage collection. Only when the garbage collector performs a sweep over memory, looking for reclaimable objects, will the memory for `pos` be ready for new objects. Most of the time, this second stage of limbo is short, because garbage collections typically run ever few seconds.

However, if there is a long interval of time during which your application allocates very few objects in the Java heap, it could be quite a while before these dragging objects are reclaimed. You should be cautious here if, during these lulls in Java object creation, your application is heavily exercising the native heap. For example, say your application has small Java objects that hold on to large native resources. Even if, once the Java objects are collected you can up the associated nativ resources, you may still exhaust native resources. This is because dragging Java objects will only be collected when you run out of Java heap space. The JRE doesn't know to schedule a garbage collection when you exhust native resources.

12.4 The Basic Ways of Keeping an Object Alive

An object will stay around as long as it is reachable from some chain of references. The nature of the references along any chains (for there may be multiple such chains!) leading up to your object will determine how long it'll stick around. After your program creates an object, it references the object in one or more of three basic ways shown on the right. Each comes with its own guidelines as to how it governs lifetime, and how you can control it.

The basic ways of referencing an object:

- instance field of any other object
- static field of a class
- local variable of a method
- a *shared* combination of the above

The Lifetime of Instance Field References If object `B` is dominated by an instance field of object `A`, then `B` will become garbage collectable only under two circumstances. Once `A` becomes collectable, then so will `B`. Notice how, in this way, a dominating reference is one way to implement the correlated lifetime pattern of Section 11.3. The other possibility is that you insert code that, at some point, assigns this reference to `null`. Since `A` dominates `B`, therefore, at this point, there will be no way for the garbage collector to reach `B`, and so it will become garbage collectable.

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.

```
class F {
  static Object static_obj;

  void f() {
    Object obj = new Object();
    static_obj = obj;
    ...
  }
}
```

Figure 12.5. When `f` returns, `obj` ownership automatically ends, but `static_obj` ownership persists.

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, when the body of a clause of an if/then/else statement finishes, or when the method invocation returns, there is a good chance that the object referenced by that variable will be reclaimable. If the local variable was the sole owner of the object, it will indeed become reclaimable.

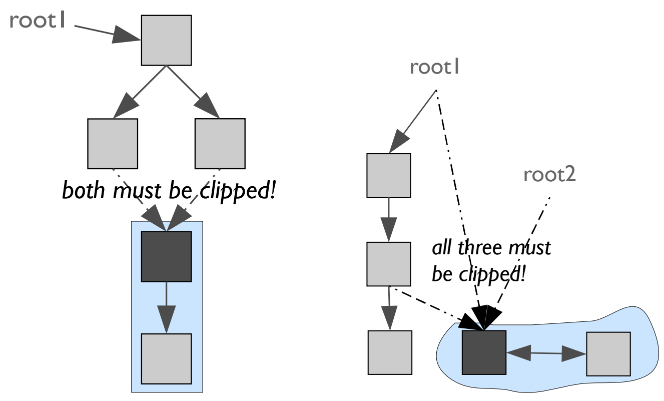
There are situations where an object may *escape* the local scope in which it was declared. This is an example of an object escaping a local variable scope, so that, beyond the duration of the local scope, it will remain alive, now owned by a static field of a class object: The next section discusses the lifetime of static fields.

```
static public void main(String[]
    args) {
  Object obj = new Object() {
    protected void finalize() {
      System.out.println("Yes");
    }
  };
  for (int i = 0; i < 1000000; i
      ++) {
    new HashSet().add(i);
  }
  System.out.println("End");
}
```

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 this, by attempting to infer a more precise line of code after which an object will not be used. For this reason, a *few* cases of memory drag that would have been a problem with older JREs, are no longer an issue. If you're curious to know whether your JRE does something clever, you can run the test program on the right. The `obj` object is owned by a local variable that is declared in the scope of the `main` method. This method does not return until the program exits. If you see the message before the program terminates, this means that the JRE has smartly determined that `obj` is not used beyond a certain point. If you see the `Yes` 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 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.

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* paths to an object must be clipped. This is tricky in many cases, because it may not be easy to know where all of path paths emanate from. Figure 12.6 illustrates a situation where three references must be clipped before an object, the darkly shaded one, becomes a candidate for garbage collection. There are



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

Figure 12.6. 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.

Advanced Reference	Purpose	Chapter
weak	correlated lifetimes	Chapter 13
soft	caching, safety valves	Chapter 14
final, phantom	cleaning up external resources	Chapter 13
thread-local	avoiding lock contention	Chapter 14

Table 12.2. Java offers several more advanced ways of referencing objects.

two other important things to note in this example. First, just as in Figure 12.3(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.

12.5 The Advanced Ways of Keeping an Object Alive

The Java language provides mechanisms that allow you more flexibility in implementing lifetime management policies. These advanced features are exposed via soft, weak, and phantom references, finalization, and thread-local storage. In contrast, the term used for the normal way of referencing objects is a *strong* reference; i.e. this is what you get when you use fields of objects or local variables,

Soft and Weak References An object is *strongly reachable* if it is reachable only from strong references. For a strongly reachable object, the normal garbage collection rules apply: when it is no longer reachable from the current set of roots, it is a candidate for garbage collection. Soft and weak references are features of the

Java language that let you guide the normal process, so that you can more easily implement certain lifetime patterns. Programmers are often confused by these two, and web searches will reveal some degree of misinformation. It is common for web sites misdefine these two mechanisms, e.g. by swapping their roles.

Soft references are useful when you need to implement any kind of logic that needs reclamation only when memory is tight. In this way, soft references can form the basis of caching, or to implement safety valves. If an object is reachable by only a soft reference, then the garbage collector is free to reclaim it whenever it wants.

The Java language specification places no firm criteria upon JRE implementations as to when they should clip soft references. In practice, soft references won't be clipped at the random whim of the JRE. All JREs these days will attempt to clip soft references in a roughly LRU (least-recently used) fashion: soft references that haven't been traversed in a while will be clipped before those that are frequently used. This is why soft references can form the basis for cache implementations.

Weak references can help you implement a correlated lifetime pattern. If an object is referenced by both a strong and a weak reference, then of course the object remains live, due to the strong reference. Your code has two ways to access the object, via either the strong or the weak reference, but only one of them is keeping the object alive. As soon as the strong reference goes away, the referenced object will become garbage collectable.

If it's not immediately obvious to you how this behavior can be used to implement a correlated lifetime pattern, you're not alone. It's tricky! Say you'd like to correlate an annotation `A` with an object `X`. How should you link these together with strong and weak references to make the magic happen, i.e. that when `X` is reclaimed, then `A` must also be reclaimed? Should the weak reference point to `A`, since it is the object you'd like to go away automatically? You can't just only weakly reference `A`, otherwise it'd be reclaimed in very short order (at the next garbage collection). It turns out that getting this correct isn't easy. Section 13.1 goes into more detail on this topic, and shows you how to use the standard `WeakHashMap` class to avoid most of the messy details.

In real code, things will be a bit more complicated. It is possible that an object is referenced entirely by strong references, but that it is *reachable* only from a soft reference. In this case, everything dominated by that soft reference will become garbage collectable as soon as memory gets tight.

Softly and Weakly Reachable

An object is *softly reachable* if it is not strongly reachable, but there is at least one path that contains a *soft reference.* An object is *weakly reachable* if it is neither strong nor softly reachable, but there exists at least one path that contains a *weak reference.*

```
void foo(DateFormat f) {
  SoftReference<DateFormat> ref;
  ref = new SoftReference<
     DateFormat>(f);
  ...
}
```

Figure 12.7. Creating soft references.

You should not use these references lightly. Table 12.3 summarizes the costs of the ways of one object referencing another. A strong reference costs one pointer, which is 4 bytes (32 bits) on a 32-bit JRE. In comparison, a weak reference is *seven times* more costly, at 28 bytes per reference, and a soft reference is nine times more costly. The reason for these expenses is that you must create an extra `Reference` object for every soft or weak reference you use. To softly reference a date formatter, you would write code such as appears in Figure 12.7.

Reference Queues Normally, when you create a soft reference, and the JRE decides to clip it, then the associated `Reference` object becomes immediately garbage collectable. You get no warning that the JRE has clipped this reference. Sometimes, this is good, because it lets you very easily take advantage of the soft and weak referencing mechanisms. More often, however, your code will need some warning that a reference has been clipped. For example, if you are using soft references to implement a cache (this is discussed in more detail in subsequent chapters), then you will probably need to perform some cleanup work when the reference is clipped.

reference type	memory cost
strong	4 bytes (one pointer)
finalization	28 bytes
phantom	28 bytes
weak	28 bytes
soft	36 bytes

Table 12.3. The per-reference cost one object referencing another.

Java provides reference queues for exactly this purpose. When you construct a soft or weak reference, and associate it with a reference queue, then something different happens when the reference is clipped. Instead of becoming collectable, when clipped, it is placed on the associated reference queue. It is your job to periodically *poll* the reference queue in order to enquire as to whether any references have been clipped. If this somes tedious and error prone, it is! Subsequent chapters show how to get it right.

Another point of caution: reference queues add an extra pointer to the already high memory expense of using soft or weak references. For example, a weak reference with an associated reference queue would cost 32 bytes per reference.

Handling External Resources: Finalization and Phantom References Weak references can help you to correlate the life of one object with that of another. Sometimes, it is necessary instead to correlate an *action* with the reclamation of an object. This is helpful if there are non-Java resources associated with a Java object that need to be cleaned up along with that object. The JVM knows nothing about those resources, since they aren't Java objects; they may not even be memory, per se, or may be memory on an entirely different machine. For example, a Java `DatabaseConnection` object has implicit linkage to a whole slew of resources, scattered across several machines. There are operating resources on the local machine that are involved. The remote database machine has these, too, and the database process also has some internal state about that connection. All of this state needs to be cleaned up when the Java facade for it is reclaimed.

As long as the resources your application uses fall within the JRE's boundaries, then the automated mechanisms, such as garbage collection and class loading, work without much intervention on your part. For the more complex cases involving external resources, such as a database connection, the nicely automated world starts to fall apart a bit. The JRE doesn't know about inherent limits on the capacity of external resources, nor the relationship between Java objects and them, and so cannot manage the complete picture.

One recourse is to fall back to the C style of resource management, for these cases of external resources. You can design the API specifications so as to require users of the API to adopt a convention that requires explicit closing, or freeing, of resources.

Java provides *finalization* as a way around this problem, though it comes with many gotchas. Finalization is very much like the system of weak references, in that it is a way to correlate an action with the death of an object. If you write a class with a `finalize` method, then, when that object is reclaimed, the finalize method will be called. Behind the scenes, the JRE allocates an extra object (of type `java.lang.ref.FinalReference`) and stores these in a reference queue (called the finalization queue). For the most part, these days, the `FinalReference` objects are allocated when the object is queued up for finalization Some older JREs allocate this object up front, when the object (that will eventually be queued for finalization) is initially allocated. In both cases, there is an extra memory overhead associated with using finalization. Phantom references are a minor variation of finalization that lets you manage the queues more flexibly: you can use separate queues for different groups of objects, and manage how often to poll the queues, rather than having a single queue and relying on the JRE to poll the queue when it feels like it.

Now for the gotchas. First, you can't assume that the finalize method will be called immediately after the correlated objects death. The Java language specification does not dictate a timeliness criterion for invoking your `finalize` method. This can lead to dragging of the resources, exactly like the problem of memory drag described earlier. Second, using finalization increases memory bloat. For every instance of a class with a `finalize` method, the JRE, behind the scenes, allocates an extra object (of type `java.lang.ref.FinalReference`). This object consumes 28 bytes; the pointer from the finalization queue adds an extra pointer cost. Third, it is possible to overwhelm the finalization thread. This can happen if your program creates instances of objects with `finalize` methods at a high rate.

Thread-local Storage When you reference an object from a static field of a class, the object will stay around pretty much for the life of the program. While this lets you implement a permanently resident lifetime pattern, it is not thread safe. In order to protect write access to static fields, you will need to guard these operations with locks. This is possible, but tedious and often results in poor performance due to contention of threads trying to acquire those locks: they need to sleep before they can enter the contended critical section.

Java provides an alternative way to implement a permanently resident pattern called *thread-local storage*. thread-local storage provides a way to clone data so that each thread has its own copy. It is impossible for one thread to access another thread's local storage. When you store objects in a thread's local storage, the objects will live for as long as the thread does, unless you `null` them out or otherwise otherwrite the entries first.

To use thread-local storage, you need to create a *thread-local variable.* A thread-local variable represents a piece of data that you want to clone across threads. For example, if you'd like each thread to have it's own date formatter, because your date formatter isn't thread safe, then you would create a thread-local variable for it:

```
class C {
   static ThreadLocal<DateFormat> dateFormatter = new
       ThreadLocal<DateFormat> {
      protected DateFormat initialValue() {
         // called the first time a thread tries
         // to access this thread-local variable
         return new ...;
      }
   };

   String format(Date d) {
      return dateFormatter.get().format(d);
   }
}
```

The main thing you have to be cautious of is memory drag. If you store a large structure in the local storage of a thread that lasts for the duration of the program, but only use the structure early in your program's execution, then that structure will be needlessly consuming space. Your options are either to design things so that the thread terminates, rather than staying alive forever; or you can explicitly overwrite the entry, by calling `dateFormatter.put(null)`, but this must be done by the thread itself, since there is no way for one thread to access another thread's local storage; or, finally, you can have the thread-local variable store a soft reference to the data:

```
ThreadLocal<SoftReference<DateFormat>> dateFormatter;
```

Now, each thread's copy of the date formatter will be cleaned up by the garbage collector if it ever needs the space, and the formatter hasn't been used in a while. You need to take care to handle the case when the JRE does get around to clipping the soft reference, and you come back to format new dates. Still, this hybrid approach, of combining soft references and thread-local storage, can be quite powerful.

12.6 Summary

- Every Java process has multiple memory regions, each with separate size limits and separate configuration options for adjusting these limits.
- Local variables in Java programs can only store primitive data and pointers to heap-allocated objects.
- Memory leaks can easily occur in a Java application, despite it having a garbage collector.

Chapter 13

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 **??**.

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 11.3, 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 11.4, 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.

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, sharing pools, listeners, and phase/request-scoped objects.

13.1 Tools: Weak References and Weak Maps

Weak References 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.

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 13.1 then the key will very likely never be uniquely owned by the weak reference, even after the expected strong reference is reclaimed.

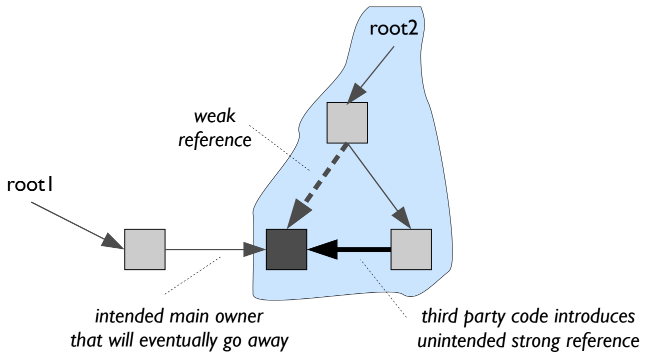

root2
weak reference
root1
intended main owner that will eventually go away
third party code introduces unintended strong reference

Figure 13.1. 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 13.1, 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;
}
```

13.2 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 13.1 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 15.3 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 13.1. 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`.

13.3 Sharing Immutable Data Safely

Review of Sharing Pools

A Leak-free Sharing Pool

A Trickier Example: An Annotation Pool For the second requirement, Java provides some support for releasing unused items from collections, namely, `WeakReference`s and `WeakHashMap`s. 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 `Node`s 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 **??** .

13.4 Cleaning Up: 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().

13.5 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 12.4) 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 12.4. 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.

13.6 Cleaning up associated resources

Finalizers 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.

13.7 The Single Strong Owner Pattern

Lifetime management for temporaries (the lifetime pattern discussed in Section 11.2), 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 12.4, 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 13.1 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 13.1). 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.

13.8 Summary

Chapter 14

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

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

14.1 Tools: 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 Java 5 and Java 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!

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.

14.2 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.

14.3 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.

14.4 Avoiding Leaks When Optimizing for Time

Sometimes a data structure can have multiple lifetime requirements. Care is required to make sure that we satisfy all of the requirements. In this section we look at two cases where we are balancing the need to save both space and recomputation time.

Example: Session State

Example: A Caching Sharing Pool

14.5 Avoiding Contention: Thread-local Stores

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.

14.6 Summary

Chapter 15

CUSTOMIZATION AND TUNING: ADVANCED LIFETIME MANAGEMENT TECHNIQUES

15.1 Understanding and Tuning the Garbage Collector

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 11.4.

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

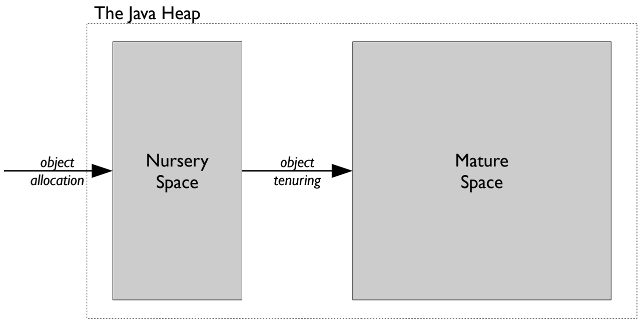


Figure 15.1. Oftentimes, the Java heap is split into two sub-heaps. The nursery space stores newly-allocated objects, and the mature space stores objects that have been tenured. The garbage collector tenures an object after it survives a sufficient number of nursery garbage collections.

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.

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 11.3.

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 15.1. 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`.)

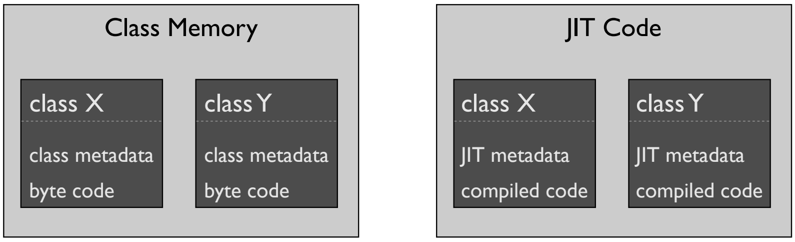


Figure 15.2. The JRE also manages classes and compiled code, but in separate heaps. With some JREs, this data is combined into a single heap, called the *Permspace* heap, which is sized via `-XX:MaxPermSize`.

The Permspace Heap In addition to separating new and old objects, the JRE stores information about classes in memory areas that are separate from those for instances of classes. Some JREs create a distinct heap for this data, one that can be sized like the other heaps. Other JREs store this data in an undifferentiated part of the general native heap. The Sun JRE is in the former camp, while the IBM and JRockit JREs are in the latter camp.

In either case, the JRE needs to allocate memory 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.

15.2 Building Your Own Lifetime Mechanisms

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 15.3 discusses solutions to this concurrency problem.

15.3 Implementing a Concurrent Cache

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 15.2, 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.

Part III

Scalability

Chapter 16

ASSESSING SCALABILITY

Despite heroic efforts at tuning classes and optimizing the use of collections, you may still be unable to fit your application into available physical memory or address space. Sometimes, heroic efforts are not even possible, because this problem was discovered very late in the development cycle. If you don't discover till the final stages of testing that your application won't fit, you will have difficulty finding the resources to perform extensive tuning. Optimizing earlier in the development process would help, but these things need to be budgeted. Resources spent on earlier tuning are resources taken away from other aspects of development, such as architectural design and coding. To avoid wasting time blindly tuning everything, it is helpful to have a quick way of knowing whether an inefficient data structure will really matter in the grand scheme of things. A particular structure might have a bloat factor of 95%, being composed of only 5% actual data, but if that structure only contributes to a few percent of overall memory consumption, who cares?

You can focus your tuning efforts by adopting a design strategy that assesses the *scalability* of your data structures. By focusing on scalability, you can answer two kinds of questions:

- **Will It Fit?** As you scale up your application, adding more users for example, it'd be nice to know whether it will fit within a given memory budget.
- **Should I Bother Tuning?** Your tuning efforts should focus on those structures that can benefit from the tuning tasks described in the Part I.

The rest of this part of the book will help you in answering these questions, and in developing solutions for the cases when the answers to both questions are no. If so, then you need to consider stepping outside of the Java box.

Will My Design Fit? Chapter 17 introduces a metric that you can use to estimate the answers to these questions. The metric, called the Maximum Room For Improvement, quantifies how much your design can benefit from tuning. From this number, you can extrapolate how much memory will be required to fit your data. If the room for improvement is high, then you should consider applying the tuning regimen detailed in Part I.

If your design doesn't fit in physical memory constraints, but you still have address space available (see Section 12.1), then the first solution you should consider is buying more physical memory. If your budget allows for this, then by all means you should strongly consider doing so.

Making it Fit by Breaking the Java Mold Despite being an object-oriented language, there are still some tricks you can use that let you continue to program in Java, but store objects in a non-object oriented way. These tricks come at the expense of some xtra programming time and maintenance expense, but can dramatically increase the scalability of your data models. Chapter 18 describes these *bulk storage* techniques.

It Still Won't Fit 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.

Chapter 17

ESTIMATING HOW WELL A DESIGN WILL SCALE

It would be nice to be able to predict how well a data model design and implementation will scale up, as you increase the complexity or number of entities. The bloat factor of a data structure tells you, for a given size, what fraction of its size is overhead versus actual data. To judge whether it will scale requires extrapolating these costs out. This chapter introduces a metric, and some facilitating techniques, that enable you to more quickly make these predictions. The metric is called the Maximum Room for Improvement, which is the highest factor of improvement in scalability you can expect to wring out of a given design, after all possibly amortizable costs have been amortized away.

17.1 The Asymptotic Nature of Bloat

Up till now, bloat factor has been treated as a scalar quantity: e.g., 95% of space is devoted to implementation overhead rather than actual data. More accurately, the bloat factor of a data structure is a function of its size. The bloat factor of a collection of objects decreases until it reaches an *asymptote*, the lowest bloat factor a data structure car achieve, no matter how big it grows.

Data Structure Scaling: Asymptotic Bloat Factor

A data structure scales well, or not, depending upon how its bloat factor changes as it grows. It starts by decreasing, as the collection's fixed costs are amortized over a larger amount of actual data, and soon levels off. This *asymptotic bloat factor* is governed by the collection's variable cost and the bloat factor and *unit cost* of the contents.

Figure 17.1 illustrates an example of the asymptotic behavior of bloat factor for two collection types. With a small number of elements, the fixed costs dominate. For this example, for a `HashMap`, with more than about 10 elements, fixed costs are pretty much fully amortized; the fixed cost of a `ConcurrentHashMap` requires more

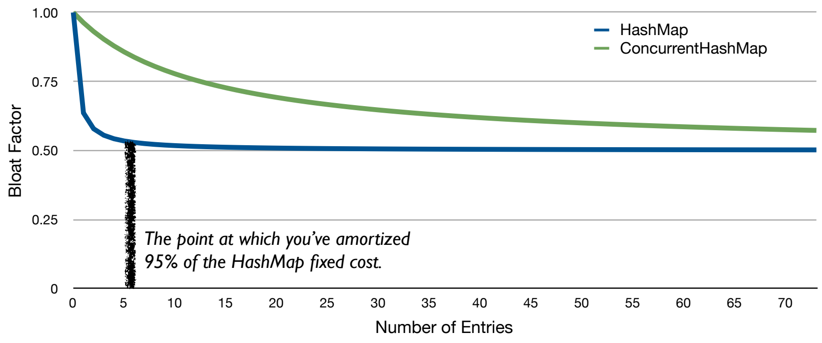


Figure 17.1. An example of the asymptotic behavior of bloat factor.

elements to amortize away. In either case, at this point, the bloat factor of the elements being stored in the collection, along with the collection's variable costs, become the dominant factors. In this example, the total cost per element is 128 bytes, 64 bytes of which is overhead. Ultimately, it is that ratio of 64/128, or 0.5, which governs the asymptotic bloat factor of this structure.

Amortizing Fixed Costs Data structures that can change in size, as opposed to having a fixed size, do so because they make use of collection. As you learned in Part I, collections of objects have a *fixed cost* that is independent of the number of entries in contains. For a simple array, this is the JRE object header. For more complex collections of objects, such as `java.util.HashSet`, the fixed cost includes a extra wrapper cost. This fixed cost is quickly amortized, usually once the data structure grows to have more than a dozen or two elements.

The number of elements that it takes to amortize a collections fixed cost depends on that cost in comparison to the memory cost of storing elements. If the collection's fixed cost is 48 bytes, and it costs 128 bytes per stored element, then the collection must have at least 6 elements before the fixed overhead contributes less than 5% to the overall size of the collection:

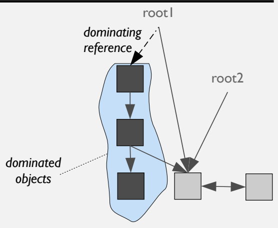
$$\frac{\text{collection fixed cost}}{\text{collection total cost}} = \frac{48}{48 + 128N} \leq 0.05 \Longrightarrow N \geq 5.45$$

Once fixed costs have largely amortized, the asymptotic bloat factor is reached. For this reason, at least for collections that you expect to be at least moderately sized, it is the asymptotic bloat factor that should be your primary concern.

Unit Costs However, the fixed costs of collections still play an important role in the ultimate scalability of most data structure. When one collection is nested inside of another, the fixed cost of nested collection contributes to the *unit cost* of storing things in the outer collection.

The Unit Cost of Storing Data in Collections

The *unit cost* of storing elements in a collection is the average size of each contained element. This cost includes everything uniquely owned by the elements. See Figure 12.2 for more discussion of *dominance*, which is a property of a graph of objects as illustrated on the right.



Every class has a *unit cost*, the cost for every additional instance. You can determine the unit cost of a class by counting up the size of its fields, taking into account JRE-imposed costs. Similarly, every interconnected group of instances has a unit cost, the sum of the sizes of the classes involved in that group. Section 3.1 introduced this style of accounting. Figure 17.2 gives an example EC diagram of a HashMap that contains an interconnected group of four objects. The unit cost of each entry in this structure is given by the sum of the sizes of those classes: 88 bytes, in this case.

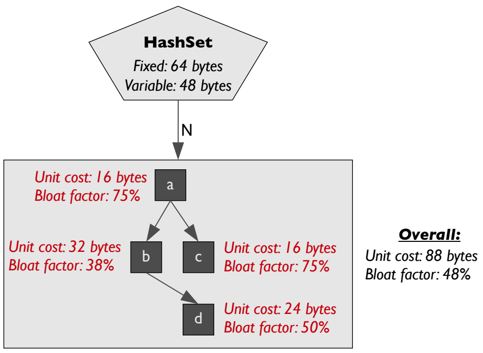


Figure 17.2. EC diagram for a `HashSet` that contains data structures, each composed of four interconnected objects.

Quite often, the elements you are adding themselves consist of nested collections. If you don't expect those nested collections to grow, then they contribute to the unit cost of additional elements in the structure that *is* growing. A fixed-size collection has a unit cost which is that collection's fixed cost, plus the total size of all of the variable costs and unit costs, counted once for each entry. So, if you have a fixed-size collection with four entries, then the total cost is the fixed cost plus four times the sum of the variable costs of the collection plus the unit cost of what's inside.

Determining the Maximum Room For Improvement Figure 17.2 is annotated with the four numbers that govern the scalability of this structure: the fixed and variable costs of the collection you use, and the unit cost and bloat factor of the data you're storing inside of it. The maximum room for improvement of a collection of data structures is given by:

$$V = \text{collection variable cost}$$
$$U = \text{unit cost of actual data}$$
$$B = \text{bloat factor of contained structures}$$

$$\textbf{maximum room for improvement} = \frac{V}{U} + \frac{1}{1-B}$$

Rule of Thumb: When Should I Bother Tuning?

A good rule of thumb to following, when deciding whether to tune or to buy more hardware in order to increase scalability, is to look at the asymptotic bloat factor of your dominant structures. For each data structure that are expected to grow the largest, tune it only when its asymptotic bloat factor is above 50%. You can estimate which structures are the dominant ones by looking at the unit cost of each, and multiplying these figures out by how many elements you'd like to have in each.

For example, the variable cost of a `HashMap` is 48 bytes. If you're storing structures with a unit cost of 40 bytes each with a bloat factor of 50%, then the maximum room for improvement is $\frac{48}{40} + \frac{1}{1-0.5}$ or 3.2x. This means that there is a fairly good scalability benefit you'll see from tuning.

Will It Fit? Should I Tune? From unit costs and the maximum room for improvement, you can estimate answers to the these two important scalability questions. If you have a fixed amount of heap size that each process has access to, then your data model designs will fit if memory capacity divided by unit cost, which is the number of elements you can afford, is at least as large as you need. For example, if you have one gigabyte of memory available, and each user comes with a unit cost of one megabyte, then you can support at most 1000 simultaneous users. Is this enough?

If not, then you have two options: buy more hardware, or tune. If possible, you could buy more memory, or more machines if your current machines cannot accept any more physical memory. You must also pay attention to address space limits, as discussed in Section 12.1: if your 32-bit processes cannot fit anything more into the constrained address space, then the answer to "Will It Fit?" is no! In this case, buying more physical memory won't help, and your only option is to tune your data models.

The maximum room for improvement of your dominant data structures can help you to understand whether you should bother tuning. If a data structure has a low bloat factor, then there isn't much bloat to optimize away. For example, in a web application server, session state will scale up roughly depending on the number of concurrent users. If the unit cost per user is one megabyte, and you'd like to support 1000 concurrent users, then this is clearly a dominant structure. If you can't afford one gigabyte for session state, then and if the bloat factor of your session state data models is greater than 50%, then you should consider tuning these models.

17.2 Quickly Estimating the Scalabilty of an Application

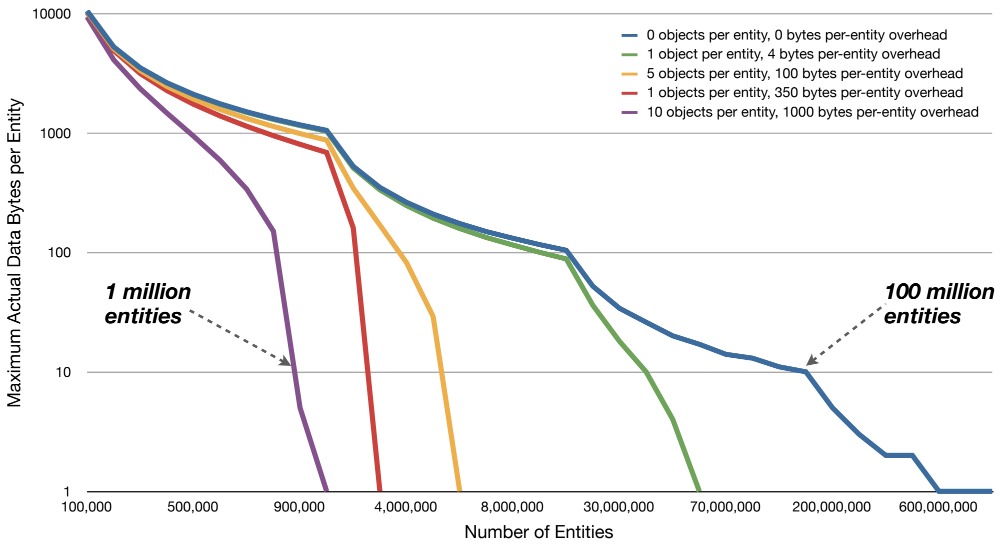
It can be tedious to construct formulas in order to estimate the scalability of your data models. Estimating scalability can require navigating a space with many dimensions of freedom. Sometimes, you have a fixed amount of memory, and a fixed amount of actual data to store, and want to know how much you need to tune, in order to make it fit. Sometimes, you have some flexibility in how much data you're keeping around. Luckily, there are some simplyfing studies that you can do, where you fix certain parameers, and let others vary. Figure 17.3(a) and Figure 17.3(b) show two of these. In both cases, the amount of memory available is fixed at one gigabyte. The first chart plots how much actual data you can afford to keep around, for various degrees of bloat (the four level curves). The second chart plots how high an asymptotic bloat factor you can afford, for various amounts of actual data (the four level curves). For many cases, you can simply consult these charts. However, it isn't difficult to construct your own, as described in Figure 17.2.

17.3 Example: Designing a Graph Model for Scalability

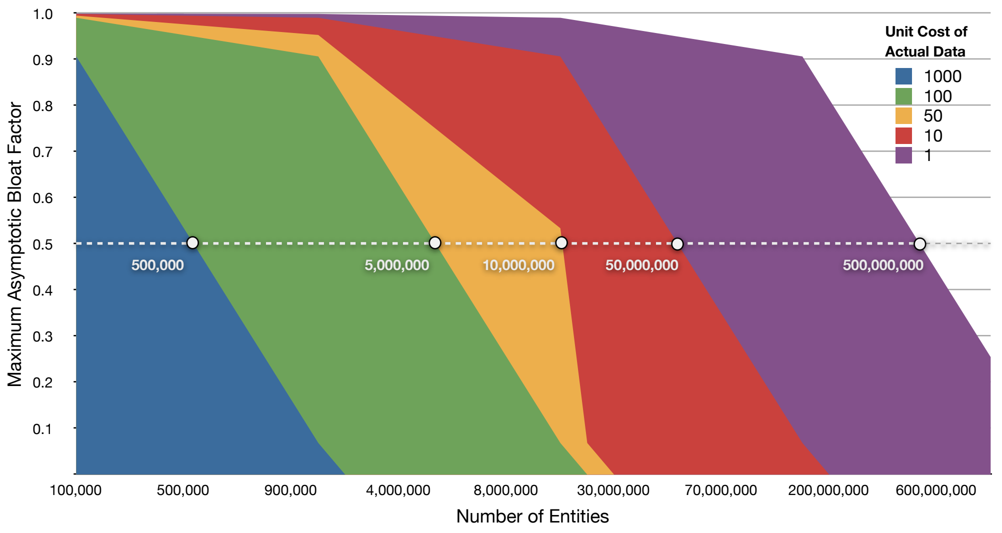
Let's step through an example of getting a data model to scale. This extended example focuses on modeling relationships between entities, like the employees within a department, or the books on a certain topic. This is a modeling task you face whenever bringing in data from some relational data store, loading in trace or log data, or modeling XML information. There are many examples that fall into this general space.

This is a task you'll often face, but getting a relational data model design to scale is hard. Database developers from Oracle, IBM, and others, have worked for decades to tune the way their databases store this kind of information. When this data is loaded into Java, we all pay much less attention to the way that same data is laid out in the Java heap. A general-purpose storage strategy, using Java objects in the natural way, is very likely not to scale well. Something you should keep in mind is that, at each step along the way, as you tune your data model for certain use cases, is to keep a focus on the two important aspects of scalability: unit costs and asymptotic bloat factor. These factors will determine the scalability success of your design.

Modeling Relationships Means Modeling Graphs Representing relationships between entities is no different than representing a graph of nodes and edges: entities are nodes, and relationships are edges. A small graph is illustrated in Figure 17.4. When caching data from a relational database in the Java heap, each database row is an entity. Columns containing numbers, dates, string, and binary large objects (BLOBs) data are all attributes of these entities. So far, the way you'd store this entity information in Java is somewhat similar to the way you'd store things in a relational database.



(a) 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.



(b) The area under each curve shows the bloat factor you can afford, for various amounts of actual data that you need to store.

Figure 17.3. You can consult these charts to get a quick sense of where your design fits in the space of scalability. These charts are based upon having 1 gigabyte of Java heap. Note the logarithmic scale of the horizontal axis.

Deriving Scalability Formulas **[For Experts]**

It isn't hard to make your own scalability charts, with some simple algebra and your favorite spreadsheet software. Before you can plot anything, you will need to construct a formula that governs how things scale. Consider the chart in Figure 17.3(b), which plots the bloat factor that will let you fit *at least* one element of data in memory. The formula behind this chart depends upon these quantities: let M be the bytes of memory available, N be the number of entities, and D be the unit cost of your actual data, and x be the unknown maximum room for improvement that you need to solve for.

The *maximum* number of bytes per entity you can afford is the ratio of memory capacity to the number of entities: M/N. The exact cost per entity, including overhead is $\frac{1}{1-x}D$; e.g. 20 bytes of actual data with a bloat factor of 60% means the total size per entity is 50 bytes.

$$\frac{M}{N} \geq \frac{1}{1-x}D \quad \Longrightarrow \quad x \leq 1 - \frac{ND}{M}$$

```
interface INode {
  Color color();
  Collection<INode> children();
  Collection<INode> parents();
}
enum Color {
  White, LightGray, DarkGray
}
```

(a) If you don't need edge properties.

```
interface INode {
  Color color();
  Collection<IEdge> children();
  Collection<IEdge> parents();
}
interface IEdge<Property> {
   Property property();
   INode from();
   INode to();
}
```

(b) If you do!

Figure 17.5. Java interfaces that define the abstract data types for nodes and edges.

Storing relationships is where things start to diverge. In Java, it is natural to store the relationship information, such as the employees under a manager, as references to collections of other entities: a manager object points to a set of employee objects. In a database, this information is usually stored in a separate table; e.g. a table that maps managers to the employees they supervise. In Java, this isn't a very natural way to model things.

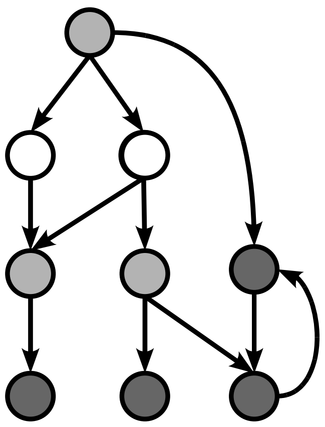


Figure 17.4. Example graph.

The implementation strategy you choose depends heavily upon the use cases that you need to handle. Do your edges have properties, such as an edge weight? Do you need random access to the edges? Do you need edges at all, or only nodes along with edge fanouts? How will you be traversing the edges? The Java interfaces help to shape your implementation to these use cases.

Every interface fixes some things, while sitll allowing some degree of freedom in the implementation. Figure 17.5 shows two interfaces that this example will work through. In one case, there are no edge properties, and in the second case, each edge as an associated weight. If your edges don't have any properties associated with them, then there is no need for an `IEdge` interface.

17.3.1 The Straightforward Implementation, and Some Tweaks

A reasonable place to start is with a straightforward mapping of interfaces to concrete classes. Figure 17.6 shows such an implementation for the `Node` data type. Following this strategy, the `Node` class has three fields, one to store its `Color`, and two for the relations to children and parents. These two relations are implemented

```
class Node<E> {
   Color color;
   Set<E> parent = new HashSet<E>();
   Set<E> children = new HashSet<E>();
}
```

(a) Node implementation.

```
class Edge<Property> {
   Property property;
   Node<Edge> from, to;
}
```

(b) Edge implementation, if you have edge properties.

Figure 17.6. A straightforward implementation of the `INode` interface, one that is parameterized the type used for parents and children; e.g. a node without edge properties would be a subclass of `Node<Node>`, because the parents and children point directly to other nodes.

with a standard `HashSet` collection. In the case where the design requires edge properties, there is an `Edge` class with one field that stores the edge property, and two reference fields that store the source and target nodes.

This is a pretty natural expression of a graph in Java, and it's easy to implement and maintain. There are no corner cases to handle, in terms of adding or removing nodes or edges. It's easy for new project members to map between interface and implementation, because the two are parallel versions of each other. The nodes and edges, and relations between them, are objects that can be manipulated using normal object oriented practices; e.g. you can write `node.children().get(5).getTo()-.color()` and, later, quickly understand what is going on. Contrast this with interacting with a non-object-oriented data storage, such as memcached[?] or a relational database. To access an attribute in this non-OO data would be more work, and would not read as cleanly.[1]

In this implementation, the unit cost per node is three pointers plus two collections of default size. If a typical node has one parent and two children, then the unit cost of a node will be 404 bytes: one object header, plus three pointer fields, plus two 136-byte collection fixed costs, plus three 28-byte collection variable costs (Table 8.1 shows the fixed and variable costs of standard collections). At this unit cost, you can fit at most 2.6 million nodes into one gigabyte of Java heap. Is it worth tuning? That depends on the maximum room For improvement of this design, as things scale up.

Every node stores 16 bytes of actual data (its color, parent, and two children), compared to its 380 byte total unit cost. Let's assume that the nodes are stored in a simple array. From these values, we can compute the maximum room for

[1]There are frameworks that hide the details of accessing this information, such as Hibernate[?]. Under the covers, though, the same mismatch exists, and now you are faced with the added complexities of interacting with these APIs.

improvement:

$$\begin{aligned}
V &= 4 && \text{(variable cost of node array)}\\
U &= 16 && \text{(actual data per node)}\\
B &= 1 - 16/380 && \text{(bloat factor of node)}\\
\textbf{maximum room for improvement} &= \frac{V}{U} + \frac{1}{1-B} = 24x
\end{aligned}$$

The maximum room for improvement is a factor of 24x, which is very high. Table 17.1 tracks the maximum room for improvement of various storage designs.

If you need to support edge properties, then the scalability story gets worse. In this case, In addition to the cost of the nodes are the cost of the edge objects. By objectifying each edge, this implementation pays a cost of one object header, one 4-byte data field (let's assume that the edge properties are simple weights, and you store the primitive integer inline with the edge object), and two pointers, or 24 bytes. For the example an average of one parent and two children per node means three `Edge` objects per node. This increases the effective cost per node to $380+3*24$, or 428 bytes. In this case, the Maximum Room for Improvement is 27x.

An easy way to increase scalability is to use a list, rather than a set, to store the edges. An `ArrayList` has much lower fixed and variable costs than a `HashSet`. This should be a fine replacement, as long as you don't need the ability to randomly delete edges from the graph, or check for duplicate edges in a node with many edges. An `ArrayList` handles random deletions just fine, but deletions from the middle of a list are a hidden cost of which you should be cautious. What's worse, whereas a `HashSet` quickly and transparently eliminates duplicates, you have to manage duplicate elimination yourself, and with expensive linear scans. This linear scans won't be a problem if the number of parent or child edges per node is less than around three. Using `ArrayList` instead of a hash map would lower the total unit cost per node from 404 bytes down to 224 bytes. This small change has increased the number of nodes you can fit in a gigabyte from 2.8 million up to 4.8 million. The maximum room for improvement of this design is 14x (19x with edge properties).

storage design	MRI without edge properties	MRI with edge properties
`HashSet`	24x	27x
`ArrayList`	14x	19x
`ArrayList(2)`	10x	15x
no collections	2x	7x

Table 17.1. The Maximum Room For Improvement (MRI) of storage designs for a graph, both without and with edge properties.

This means that there remains quite a bit more bloat to be squeezed out. If you know that the number of parents and children will be typically no more than two, then you can use a smaller initial size for the edge lists. If you update the constructor call for the `ArrayLists` to request an initial capacity of two elements, then the total unit cost of the nodes drops to 160 bytes. You can now fit 6.7 million nodes in a gigabyte heap, and the remaining maximum room for improvement is now 10x (15x with edge properties).

These easy tweaks can bring you a great return on a minimal investment of your time. They don't in any great way negatively impact the maintainability of the code. But there's still a very large amount of bloat remaining. The variations so far haven't greatly impacted the functionality of the implementation. By specializing your code to handle only a limited degree of functionality, you can achieve a fair degree of compactness without much additional effort.

17.3.2 Specializing the Implementation to Remove Collections

One of the main remaining sources of bloat lies in the use of collections to store edges. Every node has two collections, even if it only has one or two outgoing edges. As a result, every node pays for the fixed cost of a collection and, with only a small number of incident or outgoing edges, does not amortize these fixed costs.

If every node has exactly the same small number of incoming and outgoing edges, an alternative implementation presents itself. Figure 17.7a shows an implementation of the `Node` interface that does away with collections. Even if many nodes have no parents, or fewer than two children, this specialized implementation remains preferable to the original one that uses collections. The flexibility of collections simply does not pay off at this small scale. For each node, on top of the 16 bytes of actual data ideal cost, this implementation costs only one object header. The bloat factor is 43%, which reduces the maximum room for improvement to $\frac{V}{U} - \frac{1}{1-B} = \frac{4}{16} - \frac{1}{1-.43}$, or 2x. You can now support around 38 million nodes per gigabyte of heap! If you need edge properties, then the maximum room for improvement is still high, at 7x.

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

Figure 17.7. No collections: a specialized design for the case that no object has more than one parent and two children.

Though quite scalable, this implementation presents several complications. The `INode.parents()` interface is specified to return a `Collection`. How can one efficiently support an interface that expects a collection, if the storage contains only a single pointer? If you don't make this API change, and instead choose to return a *facade* that routes the edge operations to the underlying storage, users of the API will be in for some surprises:

```
public Collection<E> parents() {
   return Collections<E>.singletonList(parent);
}
public Collection<E> children() {
   List<E> l = new ArrayList<E>(2);
   l.add(child1); l.add(child2);
```

```
        return 1;
    }
```

Firstly, if a caller of `children()` does so in a loop, then making this change will result in a potentially big slowdown. A loop that depends heavily upon calls to this method run an order of magnitude slower after this change. Secondly, this implementation will not reflect any updates that the caller makes to the returned collections. Finally, it violates a contract that is implicit in the interface: that two calls to the `parents()` interface have reference (i.e. `==`) equality. The only solution, short of caching these collections (and thus undoing this optimization entirely), requires that you revise the `INode` interface. This is not a very appealing requirement, to expose implementation details to the interface.

Third, in addition to being quite expensive, in creating a set for every call to `parent()`, 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.

17.3.3 Supporting Edge Properties in An Optimized Implementation

If you do need edge properties, but want to avoid the expense of `Edge` objects, things can get tricky in a language like Java. In this design, the API method `Node.children()` returns a collection of nodes, not one of edges. Thus, the code to access an edge label requires an API change. You could play a trick similar to the one above, where transient collections were created in order to avoid the cost of persistant collections while preserving a collections-oriented API for accessing edges. This change also requires that you store the edge properties somewhere other than in `Edge` objects:

```
    class Node<Property> {
        Node parent, child1, child2;
        Property parentEdgeProp;

        public Collection<IEdge> parent() {
            return Collections<IEdge>.singleton(new EdgeFacade(
                parent, this, parentProperty));
        }
    }
```

Notice how, since, in this special form of a graph, each node has at most one parent, therefore you needn't store edge properties of a node's outgoing (children) edges. These edge properties can be accessed from `child1.parentEdgeProp`. With this design, you add no bloat in supporting edge properties, and so you can achieve the same maximum room for improvement with or without edge properties. The downside comes from the computational expense of the `parent()` and `children()` calls. These can be quite expensive, as each call to these methods entails creating and initializing two temporary objects. Also, as with the previous facade-based solution, the breakage of reference equality needs to be strongly documented along with the API.

An alternative is to store edge properties in a side map. This 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. Each edge property now consumes an extra 28 bytes for a `HashMap` entry object, plus an object header (you will be forced, if you use the standard Java collections, to fully objectify the property values, even if each is a scalar quantity), or 30 bytes more than the clever implementation that inlines the properties into the node object.

This activity of tuning the original graph implementation has raised two problems. 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. But even these have issues, such as requiring users, via documentation rather than compiler assisted analysis, to avoid using reference equality. 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.

17.3.4 Supporting Growable Singleton Collections

Section 17.3.2 shows how to specialize an implementation for the case when nodes have only one or two incident and outgoing edges. This implementation is pretty efficient, but inflexible: it has to be the case that every node has this property. If, as you populate the graph, you realize that even one node violates it, then you'll need to code up complicated fallback cases. You would need to create a more general-purpose form of the node, copy over the edges you've added up until this point, remove the old node instance from the node set of the graph, and then iterate over all of the existing nodes, rerouting their edges to point to the new node instance. This is tedious code to write and maintain. Plus, if this happens even moderately often while populating the graph, can result in bad performance.

```
interface Singleton<T> extends Collection<T> {
}
```

Figure 17.8. The Singleton Collection pattern.

It is possible to maintain a degree of flexibility, allowing nodes with widely varying edge counts, while keeping the updates localized to a node, as its edge counts exceed special case boundaries. This is especially true for the special case of single-element collections. A node can also be a singleton collection if it obeys a simple contract, the *singleton collection pattern* shown in Figure 17.8: `class Node` `implements Singleton<Node>`.

In this way, it becomes possible to have a single node implementation that supports arbitrary numbers of parents, while remaining optimized those nodes with only a single parent. Furthermore, you needn't change the fields of the `Node` class from the straightforward, most general, implementation of Section 17.3.1: a field `Collection<Node> parents` can, transparently, be either a single node or a more general collection.

In order to take advantage of this technique, you will need to modify the node's `addParent()` method. 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 = parent;
     else if (parents.size() == 1) {
        Node firstParent = parents.iterator().next();
        parents = new ArrayList<Node>(2);
        parents.add(firstParent);
     } else {
        parents.add(parent);
     }
  }
}
```

It would be nice if this transition, from a singleton to a fully formed collection, could be done more transparently. But this would involve rerouting all pointers to this singleton to point to a proper set. In Java, it is not possible to write a program that transparently changes an object's reference identity, without using wrapper objects. An `ArrayList` does this, by wrapping an object around the underlying array, thus allowing the identity of the array to change as it is reallocated to be of larger or smaller size.[2]

17.4 Summary

- When designing and implementing your data models, you should keep two questions in mind: "Will it Fit?", and "Should I Bother Tuning?".

[2]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.

- It is often not possible to fully amortize the amount of bloat in a data structure. Eventually, the amount of bloat will reach an asymptote, no matter how many elements you add to it.
- The Maximum Room For Improvement is a useful metric in helping you to answer the two important questions. It is based on the asymptotic value of bloat in a data structure.
- 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.

We are left in a bit of a hard spot. If your nodes have no attributes, then you needn't waste any space on `Node` objects. All of the information lies in the edges, yet a design that includes a `INode` interface (which supports `getChildren()` and related calls such as shown in Figure 17.5), is forced to creation node objects at some point. You could overhaul the design to optimize for this case, but the new design will be foiled as soon as a use case comes along that requires node attributes. There is no degree of flexibility here. Achieving this kind of flexibility requires coding in a style that is not conventionally object oriented. This is the subject of the next chapter: bulk storage.

Chapter 18

WHEN IT WON'T FIT: BULK STORAGE

There are limits to how well an object-oriented data model can scale. At some point, you will be faced with the ineluctable limits of tuning objects. Each object has its header, and this is an unavoidable cost. The only way to avoid delegation costs, and thus amortize the cost of a header over more data fields, is to go through the manual, iterative, process of inlining the fields of one class into another. Some amount of this manual inlining makes sense, but too much runs counter to principled engineering. Often, you are blocked because you run into code that you do not own, or classes whose field layout is, for one reason or another, set in stone.

In this way, a conventional storage strategy, one which maps entities to objects and relations to collections, suffers from two problems that impact scaling up the size and complexity of a design.

Amortizing away header costs. Since entities are objects, and each object pays a header cost imposed by the JRE, you need to craft your design so that there is enough data in each object to amortize the cost of the headers, and to avoid high delegation costs.

The Fragile Base Class problem. You may (wisely!) be unwilling to touch a class that is also used by other teams for fear of adversely impacting their correctness or memory footprint. For example, say class `X` delegates some of its behavior to an instance of class `A`, and that both `B` (your class) and `C` (the other team's class) are instances of `A`. Ignoring the other teams needs, you might prefer to collapse the fields of `B` into the `X` class, removing this need for delegation and that extra object header. To do so requires either modifying the implementation of `X`, or duplicating the implementation of `X` into a modified version of `B`. Neither alternative seems very appealing. This is a variant of what is known as the *fragile base class problem.*

```
class X {
  A a;
}
class B extends A {
  int w, x;
}
class C extends A {
  int z;
}
```

Figure 18.1. Delegation costs one object header, a cost that is easy to amortize.

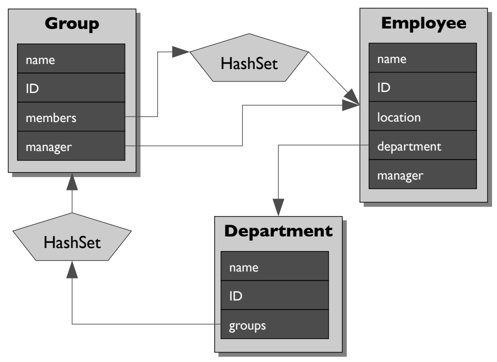


Figure 18.2. A conventional storage strategy maps entities to objects, attributes to fields, and relations to collections.

If you break out of the Java mold, you may be able to overcome these issues. An Entity-Collection diagram such as Figure 18.2 maps pretty inefficiently to a straightforward Java implementation. Each entity pays a header cost, but has only a few data fields. Most of the collections, representing relationships such as the number of employees under a manager, will be small.

Storing data in bulk form means that the relations are stored in a small and fixed number of collections, and the entities are stored in a small and fixed number of objects — across *all* entities and relations between them. They key is that the number of allocations is small and fixed, and therefore object header overheads (and allocation costs) are easily amortized. The trick to accomplishing a bulk storage approach is to store data in large arrays. This can be accomplished in two ways: arrays of records and column-oriented storage. An array of records stores the fields of the entities back to back in the array, without intervening pointers or headers. Arrays of records are not an option in Java (they are in languages such as C, Pascal, COBOL, and C#) and so this chapter focuses on the latter approach: column-oriented storage.

Warning! Bulk storage violates some basic tenets of object-oriented data modeling. Everything is a stored in an array, and thus there are no objects to encapsulate the state of an individual entity. You will learn that subclassing is more difficult to express, and that a column-oriented approach may not suit the way your data is accessed and updated. Nonetheless, with careful consideration of these issues, you can reap substantial benefits.

```
struct Node {
  enum Color color;
} nodes[20]
```

Figure 18.3. In C, `nodes` is an array of 20 integers, not pointers to 20 separate heap allocations.

18.1 Storing Your Data in Columns

In a column-oriented storage strategy, attributes are stored as arrays of data, and an entity is implicitly represented by an index into these parallel arrays. Figure 18.4 gives an example which takes the graph of objects from Figure 18.2 and represents the attributes of the `Employee` entities in this fashion. Every group of entities, such as the set of all employees, is stored in what amounts to a table of data. The range of indices, 0 to 6 in this case, over the domain of attributes (`name`, `ID`, etc.), altogether represent what was previously a graph of individual objects.

Column-oriented Storage

A column-oriented strategy stores everything, your data and the relations between them, as sets of parallel arrays. Entities, rather than being individual allocations, are indicies into these arrays.

A column-oriented storage strategy consists of four tasks. First is storing the primitive attributes of your entities, such as the `boolean` and `int` data. Second is storing the relations between entities. Third is storing variable-length attributes, such as string data. Finally is the task of establishing the set of tables. There will be one per set of entities, one per relation between entities, and one per source of variable length data. Let's step through these tasks now, continuing the example from the previous chapter: storing a graph model.

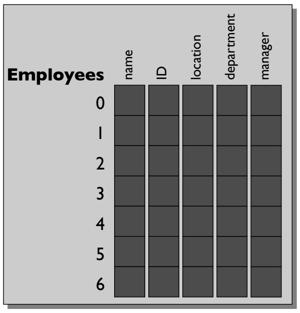


Figure 18.4. Storing attributes in parallel arrays.

18.2 Bulk Storage of Scalar Attributes

The example graph of the previous chapter finished with a bit of a condundrum. You could store a flexible number of nodes and edges, but at high level of memory bloat. Alternatively, by severely restricting the flexibility of the implementation, could you achieve a pretty good level of scalability. Within the normal confines of Java, these were the two choices. You should be able to achieve a better balance by storing the graph in a bulk form.

```
class Attribute<T> {
  T[] data;
  T get(int node) {
     return data[node];
  }
  int extent() {
    return data.length;
  }
}
```

A graph model is a good candidate for storing in a column-oriented fashion. Stored in this way, a graph without node attributes is simply one that has no arrays to store node attributes. The is a direct correspondence between need for and the existence of storage. This feature is hard to achieve in a purely object-oriented approach, where having an interface for an entity at some point demands that instances of that entity be created.

Figure 18.5(a) repeats the example graph from Figure 17.4, this time including an identifier for each node. Each identifier is a natural number that ranges, in this example, from 0–8 with no gaps. As far as node attributes are concerned, the identifier of each node need not be stored anywhere. The figure illustrates the identifiers for clarity, only. Once every node has been assigned a dense identifier, then the attribute value of a given node attribute can be stored and accessed in with that identifier.

```
interface INodes {
  int numNodes();
}

class ColorNodes implements INodes {
  Attribute<Color> colors;

  int numNodes() {
    return colors.extent();
  }

  Color getColor(int node) {
    return colors.get(node);
  }
}
```

In a column-oriented storage approach, rather than having interfaces and implementations for individual nodes, you instead have them for a *set* of nodes. An instance of INodes defines the range of node indices for the nodes of that model, and includes whatever combination of node attributes that you need for your given purpose. If, for one use case, your nodes have colors, then you include that attribute in your INodes model.

Freedom from Concensus Building If an object-oriented design is expected to be used by multiple groups, there is usually a painful, iterative, process of reaching a concensus. Many groups, with possibly competing trade-offs of time and space, and of what attributes are necessary or optional, must reach a consensus as to how to lay out the data in a class hierarchy. Deciding which attributes belong in base classes versus inherited classes, and of when to store attributes as fields or in a side object, cannot be made in isolation, one group at a time.

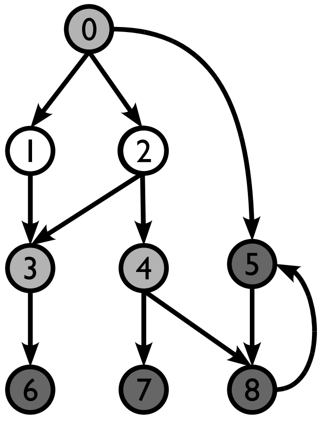
For the most part, these issues are much simpler when using column-oriented storage. Adding new attributes to nodes or edges is a trivial operation. Adding a new node attribute requires an extra array. You needn't reach a concensus among developers as to whether this is a good idea.

Transient Need for Attributes If you only need node colors for the first phase of a multi-step algorithm, then you can null out the colors attribute when you're done with it. This will clear out all memory associated with that attribute. If the colors were spready out into many individual node objects, rather than a single array, this would require essentially reforming the entire graph. Assigning the color fields of node objects to null would have no benefit, because in this case the color is a enumerated type.

```
class TransientNode {
  ColorNodes nodes;
  int node;

  Color getColor() {
    return nodes.getColor(node);
  }
}
```

Transient Boxing Since individual nodes are now just numbers, you may quickly run into some coding and maintenance problems. The Java compiler won't be of much use in giving you static typing guarantees, because a node is an int (serving as an index into arrays) just as much as integers that represent other quantities. Imagine code where methods commonly have 5 int



(a) Each node has an identifer.

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

(b) Node attribute.

source	→	target
0	→	1
0	→	2
0	→	5
1	→	3
2	→	3
2	→	4
3	→	6
4	→	7
4	→	8
5	→	8
8	→	5

(c) The edges.

Figure 18.5. A graph of stored in a column-oriented fashion.

parameters, and trying to make sure you're passing the integer values in the proper order.

Using transient node facades can help here. Instead of passing around integers to represent nodes, you can pass around temporary `Node` objects. As long as manage the lifetime of these facades properly, being careful never to store them in long-lived collections, then it is possible that you won't see a huge performance hit by using them. 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.

These transient node objects act as facades to the `INodes` model, and so most store both a reference to the `INodes` model and the node's identifier. Notice that these facades have two fields. If your nodes have only one attribute, a purely object-oriented implementation would have only a single field. Though it has one extra field, on top of the earlier node implementations, for the `INodes` reference, as long as it is transient, there is some chance that this won't result in an increase in garbage collection overhead. This is something that requires experimentation in your setup. 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 disallow, by convention, reference equality checks against transient nodes. If you are not careful, you will need to persist the transient facades, ruining any benefits of the column-oriented approach. Similarly, if the lifetime of the facades is neither temporary, nor correlated with a short-running method, you will certainly see negative impacts on garbage collection overheads.

18.3 Bulk Storage of Relationships

```
class WeightedEdges {
  int[] source, target,
     weight;

  int source(int edge) {
    return source[edge];
  }

  int target(int edge) {
    return target[edge];
  }

  int weight(int edge) {
    return weight[edge];
  }
}
```

The edges of a graph can be represented as two parallel arrays, storing the source and target node indices of each edge, as shown to the left. Figure 18.5c illustrates a concrete example for a graph with 11 edges. For example, row number 4 represents the edge from node 1 to node 3. Any edge attributes, such as an edge weight, can easily be represented as attributes parallel to the source and target arrays.

This representation is a nice first step towards storing relations in a bulk form, but it cannot efficiently support graph traversals. In order to traverse the edges of a graph from a given node, you need to know the outgoing edges of a given node. In this edge layout, the only way to get the outgoing edges of a node is to scan the entire edge table, pulling out those rows whose `source` attribute matches the given node identifier.

Indexing the Edges 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 `source` and `target` arrays are sorted by the **`source`** attribute, then all of the children of a node will be stored as contiguous rows. This is a nice trick, and is what is shown in Figure 18.5c.

Having sorted the edges, you can now leverage this property to allow for more efficient traversal, as well as more efficient storage of the edge data. Notice how the outgoing edges of node 2 start at row 5, and stop at row 6 (inclusive). To visit the children of this node, without having to traverse the entire edge model, you need to store these two row numbers somewhere. These two boundary values, 5 and 6, are attributes of node 2. This means that a natural place to store the edge index is as integer attributes in the node model; let's call these `start` and `end`.

Also notice how the `source` attribute of Figure 18.5c contains lots of duplicate data; e.g. the two rows that represent the outgoing edges of node 2 have the same `source` value (the value 2!). The `start` and `end` node attributes, combined with the `target` edge attribute is all you need to traverse the graph.

Therefore, a more compact representation can eliminate the `source` attribute entirely. Figure 18.6 shows an update to Figure 18.5, where the node model now has, for the outgoing edges, the two new attributes: `start` and `end`.

Parent Edges The same thing can be done for the incoming edges, if we instead sort the edges of Figure 18.5c by the **`to`** attribute. Figure 18.6a 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 18.6b, which shows the two

		Children		Parents	
node id	**Color**	**start**	**end**	**start**	**end**
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. `start` and `end` are edge identifiers.

edge id	**node id**
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.

edge id	**node id**
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 18.6. In order to support efficient traversal of the graph edges, you will need to index them. These tables update the example of Figure 18.5 to do so. For example, node 2 has two children and one parent. The children start at index 4 in Figure 18.6b, which shows the two children to be nodes 3 and 4.

children to be nodes 3 and 4.

18.4 Bulk Storage of Variable-length Data

The last remaining type of data is the strings and other data of variable length. Storing a node or edge attribute in an array works well for any attributes each of whose values is one of Java's primitive data types. If each attribute value is an array, such as the case with string attributes, then a single attribute array does not suffice.

Even though the standard column-oriented storage strategy doesn't suffice, there is still a big gain to be had to finding some way to store this data in a bulk form. If the length of an array is 10 characters, then it has a memory bloat factor of 61%. The problem grows worse if these primitive arrays are wrapped inside of objects such as `String`. If wrapped inside of `String` objects, then your the string has a bloat factor of 83%. This is a pretty common problem, and so you should consider bulk storage of your variable-length data, eve if you decide not to store your entities and relations in bulk form.

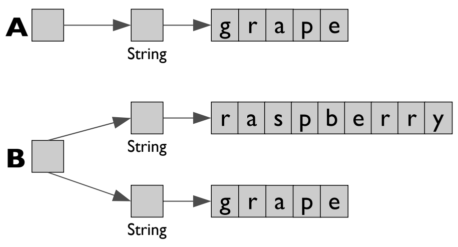
There are several ways, within the constraints of Java, to avoid the object headers of many small character arrays. One possibility is to use the built-in string interning mechanism covered in Section 6.3. By interning strings, your code will pay for the string wrapper and the character array only once, rather than once per occurrence of that string in the heap. Interning works well in reducing overall heap consumption — you're removing not only the many primitive array object headers, but the entire content of duplicated arrays. As discussed in that earlier chapter, you must pay careful attention, because interning is an expensive process, and you only see reductions in memory footprint if there are indeed duplicates to be found.

Figure 18.7(a) and Figure 18.7(b) 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; all of the `String` objects are eliminated by interning. Interning removes only the primitive array overheads of *duplicate* strings.

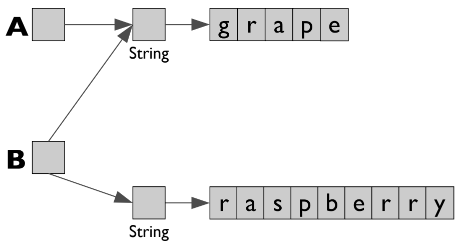
Of course, you can only use this built-in mechanism for `String` data. For non-string data, or when there isn't much in the way of duplication, you can still implement a solution that stores this data in a bulk form.

Bulk Sharing Pools If you will never synchronize or reflect on sequences, as objects, then the primitive array header and string wrapper are a needless expense. Another possibility is to concatenate your many small arrays into fewer, longer, arrays. Figure 18.7(c) 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.

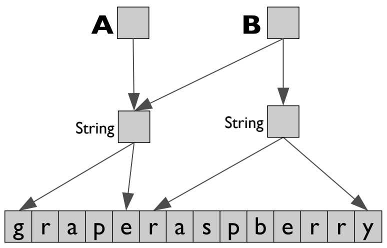
To achieve the ultimate in memory efficiency, you must also eliminate the `String` wrapper objects, as shown in Figure 18.7(d). This last step requires the most work



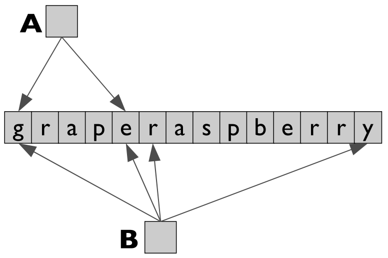
(a) Three normal `Strings`.



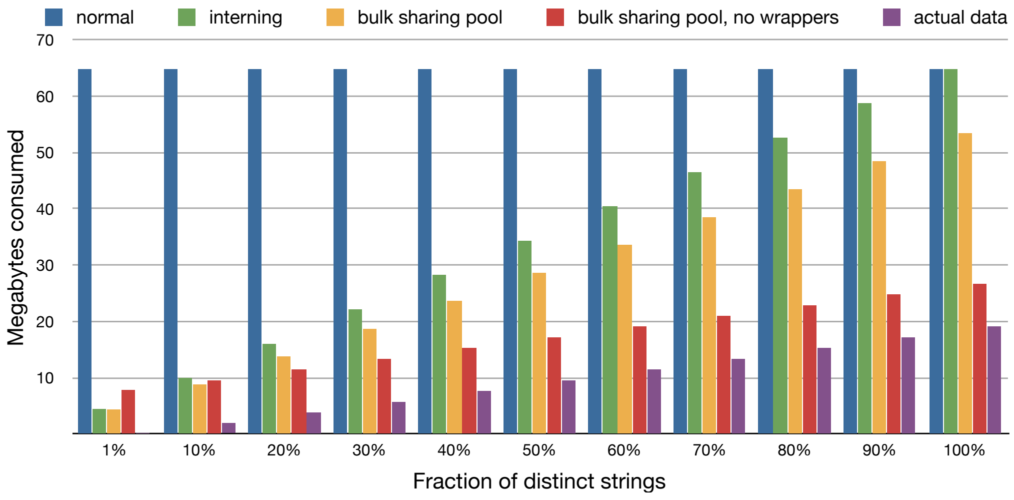
(b) Interning.



(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 18.7. 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.

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.

This last, most extreme, optimization can reap large benefits in scalability, but not always! You are paying an offset and length field in every object, even when the strings are the same. For example, in Figure 18.7(d), the offset and length fields for the two uses of "grape" contain the same data. In the previous two optimizations, these two fields are factored out into a separate `String` object, and so only stored once. If the fraction of distinct strings is small, then the cost of these duplicated fields outweighs the benefit of removing the wrappers; it even outweighs the cost of removing the character array headers. Where is the cross-over point?

Figure 18.7(e) shows 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 34 megabytes, the bulk sharing pool implementation consumes 27 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. You can see that the cross-over point, where the most extreme optimization begins to pay off, occurs when about 10% of the strings are distinct.

The chart shows just 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.

18.5 When to Consider Using Bulk Storage

The choice between bulk storage and using normal objects is analogous to the choice between using an `ArrayList` versus a `LinkedList` to store a list. An array-based list makes more efficient use of memory than its linked counterpart, but does not support efficient insertion and deletion of random list elements. Analogously, bulk storage removes all of the delegation links, and stores data and relations in arrays. Removing an entity therefore entails removing an entry from the arrays that stores the attributes of that type of entity.

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.

18.6 Summary

- Bulk storage is a technique for storing large volumes of data. It eliminates many of the usual overheads of storing objects, such as headers and delegation. Column-oriented storage is one way to store data in bulk form, and is well suited for use in Java applications. With this storage strategy, you can still program in Java and enjoy many of the benefits of the language, while simultaneusly enjoying large improvements in scalability.

- There are restrictions that will limit performance of column-oriented storage under certain circumstances. If the set of entities is in constant flux, then this strategy may not pay off.

- Your code quality will suffer somewhat. Entities are passed around as numbers, reducing the benefits of static type checking.

Chapter 19

WHEN IT WON'T FIT: WORKING WITH SECONDARY STORES

When you've tried every trick in the book, and your data still does not fit within the constraints of available memory or address space, your remaining option is to shuttle it in and out of the heap. While a subset of your data is stored in the heap, the entirety of the data needs to be persisted on some kind of *secondary storage* device. Commonly, data is persisted as files on a local filesystem or a distributed filesystems such as Hadoop's HDFS, in flat-file data stores such as sqlite or Berkeley db, as key-value pairs in distributed in-memory caches such as memcached, as tuples in a directory server, or tables in a relational database. You have lots of options!

Most often, the choice you make of how to store the data dictates how you get the data to and from the secondary store. The provider of the storage mechanism usually provides a Java library that implements a data access API. For example, every relational database comes with code that implements the Java DataBase Connectivity (JDBC) API. This API takes care of the task of turning database query results (a list of rows) into Java objects, and vice versa. These processes are termed *deserialization* and *serialization*. These terms are sometimes synonymously referred to as the steps of *marshalling* data.[1]

On top of the basic data access layers, there exist a number of libraries that hide the low-level details that are particular to any one secondary store. Most JDBC libraries will marshall data to and from Java objects, but these objects are not the objects you care about. THey are instead quite literal Java manifestations of relational database concepts: connections, prepared statements, and result sets. Hibernate, for example, raises the level of interfacing with relational databases so that you needn't be concerned with these lower level details.

[1]Some sticklers distinguish marshalling to be serialization and deserialization along with a protocol for ensuring compatibility between code releases. In this lexicon, a marshalled object is a record not only of the data, but also of the version of serialized form.

19.1 Serialization

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.

19.2 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 a high level of performance.

Additional Benefits of Memory Mapping 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.

```
ByteBuffer mapAnonymous(int numBytes) {
  return ByteBuffer.allocateDirect(numBytes);
}
```

(a) Mapping a native heap allocation into Java.

```
ByteBuffer mapFile(String file) {
  return new RandomAccessFile(file).getChannel().map(MapMode.
      READ_WRITE, 0, file.length());
}
```

(b) Mapping a file into Java.

```
IntBuffer asInts(ByteBuffer buffer) {
  return buffer.asIntBuffer().order(ByteOrder.nativeOrder());
}
```

(c) Java offers facades that let you operate on the underlying bytes as larger primitives. It is highly recommended that you use native byte ordering when possible.

Figure 19.1. Some of the memory mapping facilities offered by Java.

Memory Mapping in Java As of version 1.4, Java offers a memory mapping facility through the `java.nio` library. This Java library provides a `ByteBuffer` API for accessing data. This interface acts like an array of primitive data, even though the data may not be stored in the Java heap at all. It provides both random access and bulk `get` and `put` methods, but no insertion or deletion operations.

Using the `ByteBufer` interface, you can access data from four sources: network transmissions, files on disk, memory allocations in the native heap, and allocations in the Java heap. On UNIX platforms, these last two are often called *anonymous* maps. They have the same API and mostly the same performance characteristics as memory-mapped files, without the benefits and liabilities of a persistent backing store. 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.

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; 19.2 shows what your code might look like. If you don't force native byte ordering, and are running on hardware configured to run with little-endian bytes (e.g. an x86 core), then every call to `getInt` or `putInt` requires expensive byte swapping; if you do force native byte ordering, then the JIT compiler will compile away these method calls entirely, thus rendering `getInt` and `putInt` no more expensive than accessing an array. There can be an order of magnitude difference in performance hinging on the decision to use native byte ordering.

Memory Mapping is not Marshalling! The data being read and written is limited to the Java primitive data types, such as integers and floating point numbers. For this reason, memory mapping is not an immediate replacement for object serialization. If you already have code that is using, say, built-in Java object serialization, then don't expect the `java.nio` memory mapping facilities to provide a drop-in replacement for your marshalling needs. Any preexisting code that is expecting to interact with Java objects will either require modification. Either that code must be modified so that it operates directly on data stored in a memory mapped form, or the users of that code must be modified to create temporary facade objects that implement the expected interfaces.

Memory Mapping Your Column-oriented Bulk Storage If you've *already* committed to a column-oriented bulk storage approach (see Section 18.1) to storing your data, then you are in for a treat. In the context of a column-oriented approach, you've already made the necessary switch away from storing data as objects, and so any requisities for marshalling have already been done away with. What's better is that memory mapping and column-oriented bulk storage go hand-in-hand: the interface to all memory mappings is an array, and a column-oriented storage structure stores data as arrays. There are two variants to consider, depending on whether or not you need the data to be persisted.

If you don't need your column-oriented storage to be persisted, then the change required to shift from storing data as arrays to storing data as anonymous maps is minimal. For an integer-valued attribute over 100 elements, instead of allocating an array via `new int[100]`, you call `asInts(mapAnonymous(100))` (using the helper routines from Figure 19.1). This change should be minimal, and nicely localized.

With only a few more coding changes, you can have your models persisted. This is one of the several strengths of a storage approach that is based on memory mapping. The decision of whether or not to persist data to a file system does not involve extensive code work, nor does it require establishing and maintaining versions of serialized forms and all of the associated marshalling code.

To persist an attribute, you have to specify a place to persist it. The helper routine `mapFile` from 19.2 requires a `File` in which to store the data. If your application has a graph model such as the one discussed in Chapter 18, and needs only one of them, then choosing file names isn't very difficult. For example, the edge model can be stored in a file called "edges"; the nodes' weight attribute can be stored in a file called "nodeWeights". You need only choose an appropriate directory in which to keep these files.

If your application has multiple instances of graph models alive simultaneously, then the naming problem becomes more difficult. Now, the choice of directory becomes a critical one, so that you can avoid collisions of the edge and node models. To solve this problem, you will need to give a *name* to each graph model. For example, you may have two edge models, one that stores the department to employee relation, and one that stores the manager to employee relation. It should be easy to come up with distinctive names for these. The `WeightedEdges` example from Section 18.3, shown on the right updated to use file-backed memory mapping, looks pretty similar to the original one that used Java arrays to store the data.

```
class WeightedEdges {
 IntBuffer source, target, weight;

 WeightedEdges(String name) {
   from = asInts(mapFile(name+"from"));
   to = asInts(mapFile(name+"to"));
   weight = asInts(mapFile(name+"weight"));
 }

 int source(int edge) {
   return source.get(edge);
 }

 int target(int edge) {
   return target.get(edge);
 }

 int weight(int edge) {
   return weight.get(edge);
 }
}
```

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.

What's worse, 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 that are ready to be cleaned up. Therefore, you may suffer from `OutOfMemoryError` failures, due to running out of address space, despite the fact that many of your mapped regions are actually ready to be unmapped.

The second major 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. 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.

Appendix A

TOOLS TO HELP WITH MEMORY ANALYSIS

Appendix B

JRE OPTIONS RELATED TO MEMORY

Appendix C

A COMPARISON OF SIZINGS ON JREs