Building Memory-Efficient Java Programs

Nick Mitchell   Edith Schonberg   Gary Sevitsky

PREFACE

For over a decade, we have helped developers, testers, and performance analysts with memory-related problems in large Java applications. We have discovered that, 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. Fifteen 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.

By the time we are called in to help with a memory problem, the situation is often critical. The application is either in the final stages of testing or about to be deployed. Fixing memory problems this late in the development cycle is often very costly, and may require major code refactoring. It is clearly much better to anticipate memory requirements in the design phase, but this is rarely done. A key reason is a lack of information. This observation is the motivation for this book. 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 and correctly.

This book presents practical techniques, and a comprehensive approach, for making informed choices about memory. It covers best practices and traps, and addresses three distinct aspects of using memory well:

1. **Representing data efficiently**. The book illustrates common modeling patterns, shows how to estimate their costs, and discusses tradeoffs that can be made.

2. **Managing object lifetimes**, from very short-lived temporaries to longer-lived structures such as caches, and especially, avoiding memory leaks.

3. **Estimating the scalability of designs**, by identifying the extent to which memory overhead governs the amount of load that can be supported.

Java developers face some unique challenges when it comes to memory. First, memory usage is hidden from developers, who are encouraged to treat the Java heap as a black box, and instead to let the runtime manage it. When an application is

run for the first time, the size of the heap is largely unpredictable, some unknowable function of the engineering and architectural choices the team has made. A Java developer, assembling a system out of reusable libraries and frameworks, is truly faced with an "iceberg", where a single API call or constructor may involve many layers of hidden framework code. We have seen over and over again how easy it is for space costs to pile up across these layers.

Secondly, Java heaps are not just big, but filled with bloat. The bytes in every application's heap can be divided into two parts. Some of the heap is "real data", such as the names of employees, their identification numbers and their ages. The rest is, in various forms, the overhead of storing this data. The ratio of these two numbers gives a good sense of the memory efficiency of the implementation. A bloated heap has a high ratio of overhead to real data.

In our experience, we have seen ratios as high as 19:1, where as much as 95% of memory is devoted to overhead. In other words, only a tiny fraction of the heap is storing real data! This level of overhead often impacts the scalability of the application. And this doesn't include duplicate data and data that is not really needed, two other common sources of waste. For example, in one application, a simple transaction needed 500 kilobytes for the session state for one user. This figure is the slope of the curve that governs the number of simultaneous users that system could support.

The design of the Java language and standard libraries contribute to high overhead ratios. Java's data modeling features and managed runtime give developers fewer options than languages like C++, that allow 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.

The good news is that it is possible to make intelligent choices that save memory with existing Java building blocks — there is no need to avoid good coding practices nor rewrite libraries. However, developers do need to understand how to estimate the cost of various options and more consciously engineer the lifetime of objects. Through the extensive use of examples, this book guides the reader through common memory usage patterns. For each pattern, we help you understand what it costs, and how to choose among alternative solutions. We also help you focus your efforts where it matters the most.

The examples are chosen to illustrate common idioms, with the expectation that the reader will be able to carry the ideas presented into other situations (such as evaluating library classes we haven't discussed). As far-fetched as some of the examples may seem, most of them are distilled from cases we have seen in deployed applications. As in other domains, truth can be stranger than fiction.

The content is appropriate for a wide range of Java practitioners, and only basic knowledge of Java is assumed. Practitioners involved with either framework or application development can take this knowledge to produce more efficient applica-

tions and ones that are less prone to memory leaks. Technical managers and testers can benefit from the knowledge of how memory consumption scales up, as they help the team sanity check its designs in larger scale environments. This material should be of interest to students and teachers of software engineering, as it offers insight into the challenges of engineering at scale.

CONTENTS

vi

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 responds to your program's operation. There are lots of libraries and frameworks, written by experts, that provide powerful functionality. 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. 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 of architectural decisions, such as the choice of frameworks you use.

Alternatively, a methodology based on sound design can prevent this type of scenario from happening. Memory usage, like any other aspect of software, needs to be planned. At its core, programming is an engineering discipline, and there is no escaping the fact that the consumption of any finite resource must be measured and managed.

1.1   A Short Quiz

To start you thinking about memory consumption, here is a quiz to test how good you are at estimating sizes of Java objects. If you look inside a typical Java heap, it is mostly filled with the kinds of objects used in the quiz — boxed scalars, strings, and collections. Assume a 32-bit JVM.

Question 1.   What is the size ratio in bytes of Integer to int?

    a. 1:1
    b. 1.5:1
    c. 2:1
    d. 4:1
    e. 8:1

Question 2.   How many bytes in an 8-character String?

    a. 8 bytes
    b. 16 bytes
    c. 20 bytes
    d. 40 bytes
    e. 56 bytes

Question 3.   Arrange the following 2-element collections in size order:

    ArrayList, HashSet, LinkedList, HashMap

Question 4.   How many collections are there in a typical heap?

    a. between five and ten
    b. tens
    c. hundreds
    d. thousands
    e. two or more orders of magnitude bigger than the above

Question 5.   What is the size of an empty ConcurrentHashMap?

    a. 17 bytes
    b. 170 bytes
    c. 1700 bytes
    d. 17000 bytes
    e. 500 bytes

Question 1. Correct answer: d.

Every time a program instantiates a class, there is an object created in the heap, and as shown by the 4:1 size ratio of `Integer` to `int`, objects are not cheap.

Question 2. Correct answer: e.

Strings often consume 40-50% of the heap. They are surprisingly costly. If you are a C programmer, you might think that an 8-character string should consume 9 bytes of memory: 8 bytes for characters and 1 byte to indicate the end of the string. What could possibly be taking up 56 bytes? Part of the cost is because Java uses the 16-bit Unicode character set, but this accounts for only 16 of the 56 bytes. The rest is various kinds of overhead.

Questions 3. Correct answer: ArrayList, LinkedList, HashMap, HashSet.

After strings, collections are the largest consumers of memory in most applications. Surprisingly, a `HashSet` is bigger than a `HashMap`, even though it does less. This is an interesting fact that will be explained later.

Question 4. Correct answer: e.

In typical real programs, having hundreds of thousands, or even millions of collection instances in a heap is not at all unusual. If there are a million collections in the heap, then the collection choice matters. One collection type might use 80 bytes more than another, which may seem insignificant, but in a production execution, a few wrong choices can add hundreds of megabytes.

Question 5. Correct answer: c.

`ConcurrentHashMap`, compared to the more common collections in the standard library, is surprisingly expensive. If you are used to creating thousands of `HashMaps`, then you might think that it is not a problem to create thousands of `ConcurrentHashMaps`. There is certainly nothing in the API to warn you that `HashMaps` and `ConcurrentHashMaps` are completely different when it comes to memory usage. This example shows why understanding memory costs is important.

The quiz gives some sense of how surprising the sizes are for the Java basic objects, like boxed scalars, strings, and collections. When code is layered with multiple abstractions, memory costs become more and more difficult to predict.

1.2   Facts and Fictions

In addition to the technical and engineering challenges that managing memory pose, there are other reasons why memory problems are so common. In particular, the software culture and popular beliefs can lead you to ignore memory costs. Some of these beliefs are actually myths. Here are several.

You Can Ignore Memory In Java.

Java's garbage collector and JIT will optimize your memory usage. To avoid code complexity, you should not even think about how much memory is being used.

There is a widespread belief that objects are free, at least temporaries are, and everything will be taken care of for you by the garbage collector and the JIT compiler. Given all of the research on garbage collection and runtime optimization, this is not a surprising perception. Therefore, much of software engineering focuses on design from a maintainability standpoint, and assumes the performance will magically be taken care of.

In fact, all of the commercial JITs we know of do absolutely nothing in terms of storage optimization, so that every single object, with all of its fields, ends up as an object in the heap, taking up space.[1] Yes, garbage collectors do a good job of cleaning up temporary objects, though having a large number of temporaries will still require extra space and time. And many objects are not temporaries. It is these longer-lived objects that cause memory problems.

Libraries and Frameworks are Optimized.

Another common myth is that the libraries and frameworks are already optimized, because these were written by experts.

Standard libraries and frameworks are usually optimized for speed, not necessarily memory usage. Even when framework authors try to pay attention to memory, they often have a hard time predicting what the usage of their framework is going to be. Sometimes they guess wrong and sometimes they don't know where to start. So it's very unlikely that a framework has been optimized for your particular use case. And of course, the nesting of frameworks results in layers of unoptimized memory costs.

A related myth is that library writers and other systems programmers are the only ones who need be concerned with memory efficiency. In fact, many problems are caused by everyday data modeling and implementation decisions at every level

---

[1]Some JITs do optimize a small percentage of temporaries by allocating them on the stack.

of code. These choices, of how classes are designed and frameworks are used, can make a big difference.

There are No Memory Leaks in Java.

Since Java has a garbage collector, it isn't possible to have a memory leak.

Java has a managed runtime. An important part of this is automatic garbage collection. The standard sales pitch claims that manual reclamation of memory isn't required, and, for the most part, this is actually true. Memory leaks occur frequently in languages like C++, where the programmer can easily forget to free an object when the variable pointing to it goes away. However, in Java, the garbage collector automatically finds free objects. So how can you get a memory leak?

Problems arise when long-lived data structures hold on to objects longer than they are needed, and these structures continue to grow indefinitely. The garbage collector cannot collect objects when it thinks they are still needed. If structures continue to grow without bound, you will eventually run out of memory. This is, in fact, a memory leak, and can be a particularly hard problem to solve. Deciding how long each object should stay in memory is a necessary part of the developer's job, even with a managed runtime. The second part of this book deals extensively with this topic.

Improving Memory Usage Hurts Performance.

There will always be a performance/memory tradeoff. If you improve memory usage, then the application will run slower. Similarly, if you improve performance, the memory usage will suffer.

This is a false dichotomy in a lot of people's minds: if an application is using a lot of memory, then it must be buying you something in terms of performance. But in reality, that is not always the case. In fact, sometimes bad memory usage will result in poor performance as well. For example, if the heap is holding on to a lot of long-lived objects, then there is very little headroom for temporaries, and so the garbage collector has to run much more often. Similarly, if there are too many long-lived objects, then your cache may be too small for optimal performance.

Nothing Can Be Done.

Java is expensive, and there is nothing you can do about it. This is just the cost of object-oriented programming in Java.

The purpose of this book is to show that there are many techniques that you can employ to improve memory usage, armed with some knowledge of what things cost. There is almost always something you can do that will make a difference!

1.3   Building Memory-Efficient Java Programs

Problem-focused   This book presents a practical approach to making effective use of memory. It is organized by common design problems that are likely to have an impact on memory usage. For each design topic we point out traps, provide solutions, and help you compare the costs of alternatives.

The book is also designed to give you a comprehensive 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. At the same time, most chapters and sections are written to stand on their own, so that you can pick and choose what you need when you need it. That way, you can focus your efforts where you'll get the most benefit right away for a particular part of your system or stage of development. This approach fits nicely with different styles of development, including agile.

Estimation   A lot of the book is about estimating memory costs. Estimating memory usage early, either from source code or by measuring a running system or prototype, will let you make better choices in your designs. It will also help you focus your optimization efforts where they matter most. The approach separates the problem into a few parts. First, what does a data structure cost at a particular size? Second, how much room is there for improvement? And finally, how will it scale up?

It's also very possible to waste memory when trying to optimize, if you're not careful, since there are so many hidden memory costs in Java. At relevant points we give you formulas to estimate the savings (or waste), based on the characteristics of your data. That way you can determine whether an optimization is worthwhile.

Requirements   In practice, many problems that look like implementation errors are actually caused by an incomplete understanding of requirements. Throughout the book we give you questions to ask about your data. For example: Do you need random access into this data? Will entries ever be deleted from these collections? Is the lifetime of this object tied to another object or event? Which collections will grow larger as various aspects of load increase? Asking these questions can help you choose efficient representations that are specialized for your application's use cases. They can also help you predict scalability, and avoid lifetime management bugs like memory leaks.

Java mechanisms   Throughout the book we look at some lower-level aspects of Java that have an impact on memory. We'll look inside some library classes such as strings and collections, to give you insight into how these and similar classes make use of memory. We'll also show you how to make the best use of lifetime management mechanisms, such as weak and soft references and thread-local storage.

Roadmap   Part I is about designing and implementing memory-efficient data models. Chapter 2, on Memory Health, lays the foundation with two concepts that are used throughout the book. If you read one chapter first, this is the one to focus on. It shows you how to compute an accounting metric, the *bloat factor*, which measures the density of actual data in your data models. Memory spent on everything else, such as Java's internal object headers, is the overhead of the representation. Poor designs waste more space on these overheads, and thus have a high bloat factor. This chapter also introduces a diagramming technique, the *Entity-Collection diagram*, that makes it easy to see memory costs and overheads in a design, and to compare design alternatives. It separates out the two kinds of memory consumers: the application's entities, such as the business objects, and the choice of collections used to access them. Entities and collections have different patterns in the way they use memory, and they require different solutions.

Chapters 3 through 5 cover the design of entities. Chapter 3 gives you the nuts and bolts of estimating the memory costs of classes. You may benefit from reading this chapter early as well. It covers the basic costs of fields and objects, and the overhead when constructing entities from multiple classes. Chapter 4 covers efficient patterns for laying out classes from fields. Chapter 5 compares alternative representations for common field datatypes. Chapter 6 provides techniques for reducing the amount of duplicate data.

Chapters 7 through 10 are about using collections. Chapter 7 is an overview of collection resources and costs. The next three chapters cover the typical uses of collections: as one-to-many relationships (Chapter 8), as large data access structures such as indexes (Chapter 9), and as dynamic records and attribute maps (Chapter 10).

Part II is about managing the lifetime of objects. In Chapter 11, you will learn how to establish the lifetime requirements of your data, mapping them to one of a number of common lifetime patterns. For example, you may need to correlate some objects' lifetimes with related objects; you may want to extend some other objects' lifetimes to buy performance by avoiding recomputation. Chapter 12 provides a primer on the memory management facilities that Java offers, to help you implement the lifetime requirements of your data.

Part III covers issues related to scalability. Chapter 13 give an overview of issues to consider. Chapter 14 teaches strategies for estimating how a design will scale up as load increases. It builds on the Entity-Collection Diagram and Bloat Factor introduced in Chapter 2. Chapter 15 covers cases when your data model does not fit, even after applying the tuning methodology of Part I. This chapter presents bulk storage techniques for achieving greater scalability, without resorting to the use of secondary storage. This approach remains within the confines of Java, but gives up many of the benefits of object orientation.

The memory estimates in the examples are from the 32-bit Oracle Java 6 JRE. Appendix A is a reference for estimating memory costs for some common JREs and

architectures. Appendix B gives memory configuration options for these same JREs.

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.
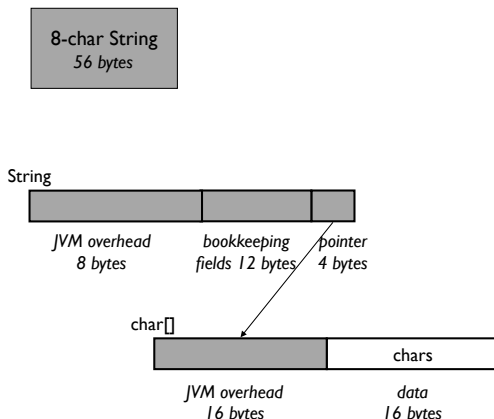
Figure 2.1. An eight-character string in Java 6.

- *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 1 that an 8-character string occupies 56 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 40 bytes are pure overhead. This structure has a *bloat factor* of 71%. The actual data occupies only 29%. These numbers vary from one JRE to another, but the overhead is always very high. (Unless otherwise noted, all measurements in this book are from the 32-bit Oracle Java 6 JRE.)

   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 gluing 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 40 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 80 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 40 bytes, so overhead of a string approaches 0% for larger and larger strings. For some data structures it is not possible to amortize 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 on 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
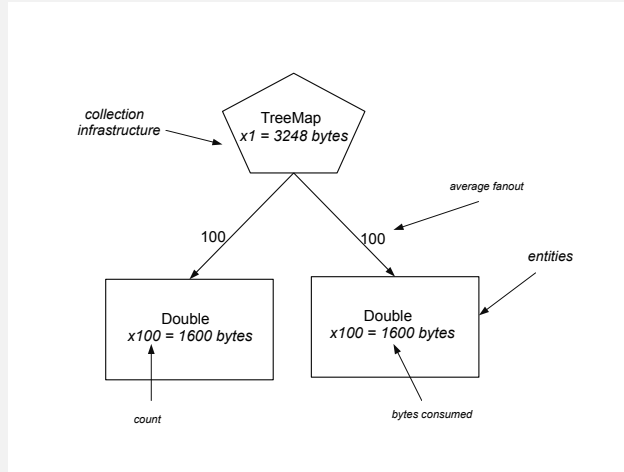
In an EC diagram, there are two types of boxes: pentagons and rectangles. Each box summarizes the implementation details of a portion of a data structure. Pentagons represent collection infrastructure, and rectangles represent the entities of your application.

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 $\times N = M$ 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 an EC diagram is labeled with the average fanout from the source node to the target node.

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 size number shown in each node. Where these other diagrams would use an edge to show a relation or role, an EC diagram uses a node to summarize the collections that implement a 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 is to display samples in chronological order, after all of the data has been
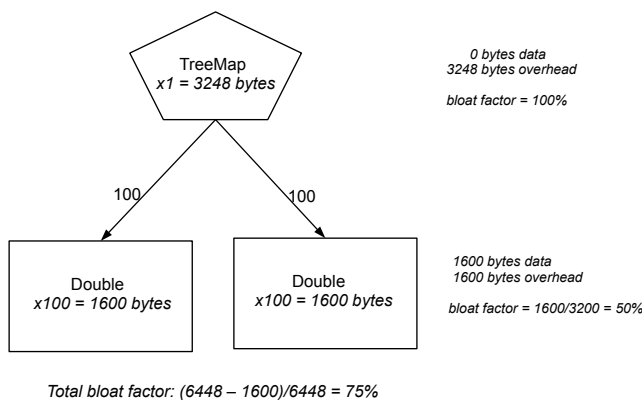
Figure 2.2. EC Diagram for 100 samples stored in a `TreeMap`

collected. At that point the user may also perform an occasional lookup of a sample by timestamp. 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 6,448 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 75%.

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 16 bytes, so 200 `Doubles` occupy 3,200 bytes. Since the data is only 1,600 bytes, 50% of the `Double` objects is actual data, and 50% is overhead. This is a high price for a basic data type.

The `TreeMap` infrastructure occupies an additional 3,248 bytes of memory. All of this is overhead. What is taking up so much space? `TreeMap`, like other collections in Java, has a wrapper object, the `TreeMap` object itself, along with 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. It has one node object for every value stored in the map. Nodes maintain pointers to parents and siblings, as well as to the keys and values[1].

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

---

[1]There is some variation among JREs.

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 the needed 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) {
```
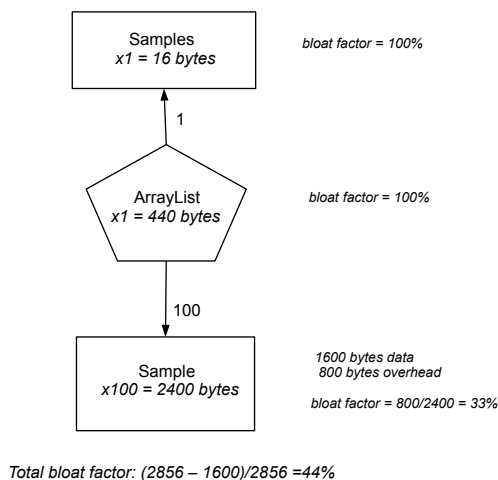
Figure 2.3. EC Diagram for 100 samples stored in an `ArrayList` of `Samples`

```
        return NOT_FOUND;
    }
    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 6,448 to 2,856 bytes, and the overhead is reduced from 75% to 44%. 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, 44% overhead still seems high. Almost half the memory is being wasted. How hard is it to get rid of this overhead completely?
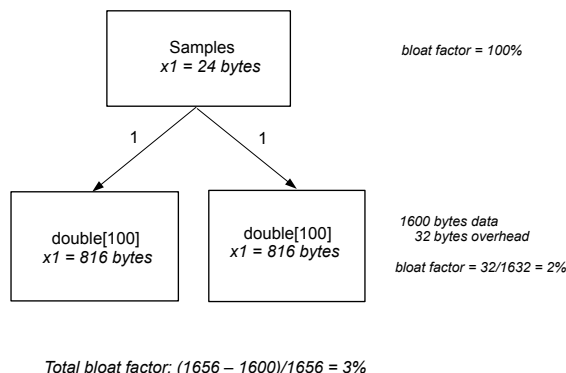
Figure 2.4. EC Diagram for 100 samples stored in two parallel arrays

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 nonetheless 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 3%.

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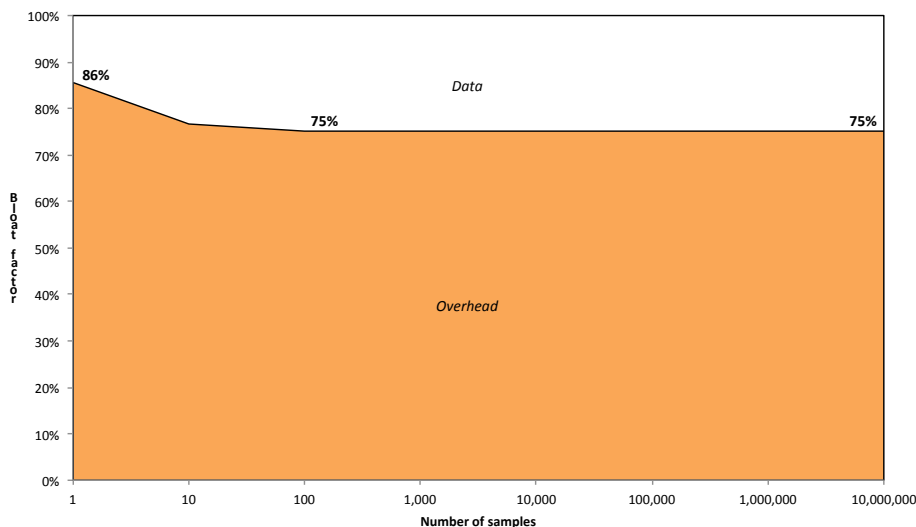
Figure 2.5. Health Measure for the `TreeMap` Design Shows Poor Scalability

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 is possible to predict much earlier how well a data structure design will scale.

The basic question for predicting scalability is what happens to the bloat factor as a data structure grows. The `TreeMap` design has 75% 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 graph in Figure 2.5 shows how the `TreeMap` design scales as the number of samples increase. The graph is split into two parts: the percentage of overhead and the percentage of data. For a single sample, the overhead is 86%. As more samples are added, the bloat factor drops to 75%. Unfortunately, with 100,000 samples and 1,000,000 samples, the bloat factor is still 75%. The `TreeMap` design is not only bloated, but it also does not scale. It is constantly bloated.

To understand why this happens, it helps to first distinguish between two kinds of overhead in collections. Recall that the infrastructure of `TreeMap` is a search tree made up of nodes. As samples are added, the infrastructure grows, with a new node for every sample. This is the `TreeMap`'s *variable overhead*, the additional space needed to store each entry. In this case it's 32 bytes, the cost of an internal node object. Each sample also adds its own overhead, namely the JVM overhead in the
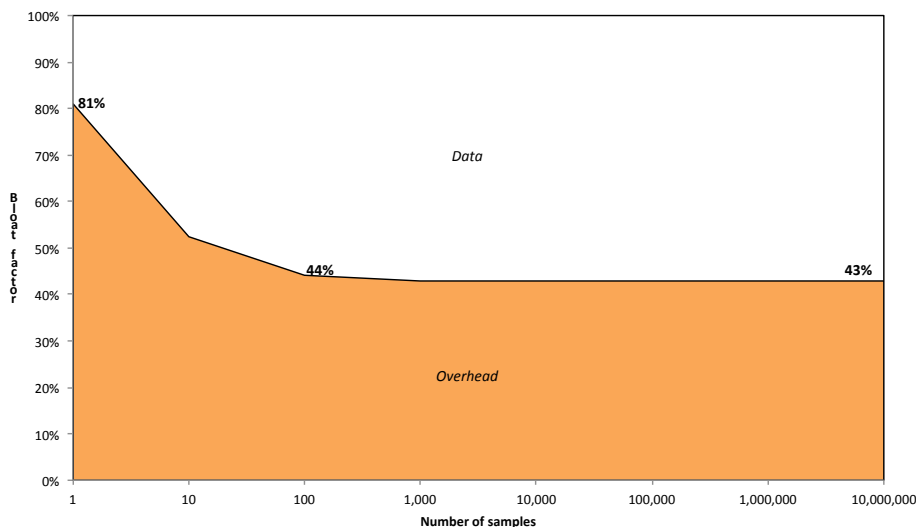
Figure 2.6. Health Measure for the `ArrayList` Design

two `Double` objects. When the map becomes large enough, the overhead per entry
— from the `TreeMap` and the `Doubles` — dominates, and hovers around 75%.

The bloat factor is larger when the `TreeMap` is small. That's because for small
`TreeMaps`, the *fixed overhead* is a large component of the cost. A collection's fixed
overhead is the base memory needed, independent of the number of entries stored.
For `TreeMap`, it's the 48-byte `TreeMap` wrapper object. As the collection grows, this
fixed cost quickly becomes less significant, as the cost is spread over more and more
samples.

Fixed vs. Variable Collection Overhead

Every collection has two types of memory overhead: *fixed* and *vari-able*. Together they determine the cost of the collection for a given
number of entries.

Fixed overhead is the minimum space needed, no matter how many
entries are stored in the collection. In a collection with few entries, a
large fixed overhead matters more. The fixed overhead is amortized as
the collection grows.

Variable overhead is the memory needed, on average, to store a new
entry in the collection. Collections with a large variable overhead do
not scale well, since the cost of the collection grows by the variable
overhead with each new entry.
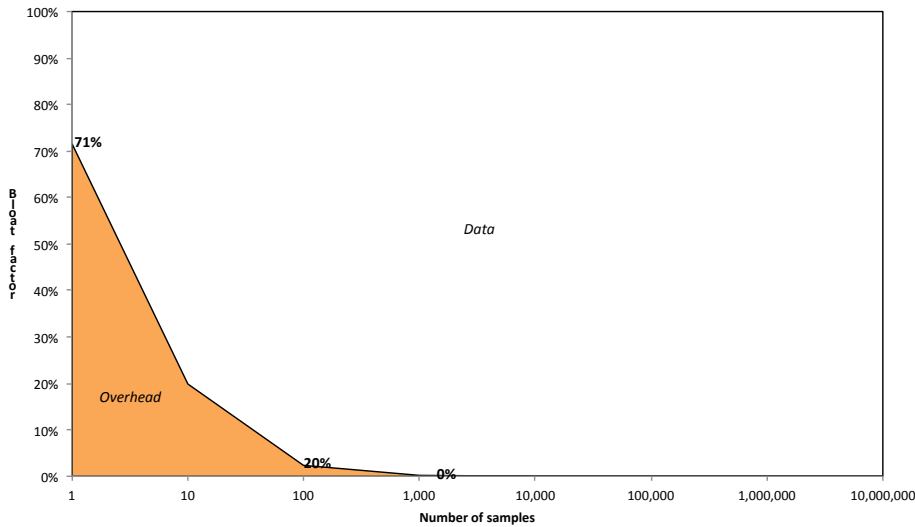
Figure 2.7. Health Measure for the Array-Based Design Shows Perfect Scalability

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, but there is still a per-entry overhead of 43% that remains constant. That's the combined effect of the `ArrayList`'s variable overhead and the overhead of each `Sample` object.

For the last design that uses arrays, there is only fixed overhead, namely, the single `Samples` object plus the JVM overhead for the arrays. There is no additional overhead for each entry, since a scalar array has no variable overhead, and the samples themselves are pure data. Figure 2.7 shows how the initial 71% fixed overhead is quickly amortized. When more samples are added, the bloat factor becomes 0%.

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 75%, so you will need 610MB to store the samples. For the `ArrayList` design, you will need 267MB. 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 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 *variable*, you can predict how much memory you will need to store collections of different sizes. Being able to predict scalability is critical to meeting the requirements of large 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 3. To estimate scalability, you will need to know what the fixed and variable costs are for the collection classes you are using. These are covered in Chapters 7 through 10.

Chapter 3

OBJECTS AND 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 prevalent 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 [1], and are given in Table 3.1. Fields can also be reference fields, pointing to other objects. Their size depends on the architecture of the JRE. They are 4 bytes on a 32-bit JRE.

Object-level overhead   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

Data type	Number of bytes
boolean, byte	1
char, short	2
int, float	4
long, double	8
reference (32-bit JRE)	4

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

	Oracle Java 6	IBM Java 6
Object header size	8 bytes	12 bytes
Array header size	12 bytes	16 bytes
Object alignment	8 byte boundary	8 byte boundary
Minimum field alignment	1 byte boundary	4 byte boundary

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

Class	Data size	Oracle Java 6			IBM Java 6		
		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, for 32-bit architectures.

header and alignment costs imposed by two JREs, Oracle Java 6 and IBM Java 6, both for 32-bit architectures. [1]

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 takes at least 16 bytes. Table 3.3 gives the sizes of boxed scalars.

Field-level overhead   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 id;                 // 4 bytes
```

---

[1]Unless otherwise noted, all of the numbers throughout the book are based on the Oracle JRE for 32-bit architectures. Appendix A gives information needed to estimate sizes in various environments.

```
        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 Oracle 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 Oracle 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+4+1+8+2+4) = 31 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 id;                // 4 bytes
        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+4+8+4+4) = 40 bytes, with no object alignment
        needed
```

Arrays   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 object
        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 object alignment cost, which become less significant when there is more data. For example, for the Oracle 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 28%. The bloat factor for an array of 100 `chars` is insignificant. An exception to this rule is when objects have a lot of fields, such as `booleans`, that carry very little data, on a JRE that doesn't pack fields tightly. These objects will have a high bloat factor. In that case, the more fields, the more overhead.

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, this kind of field is implemented using *delegation*, that is, by storing a reference to another object. Some of the most common field types are modeled this way, requiring one or more separate objects. Table 3.4 shows the space needs of the most common ones.

Class	Number of objects	Size in bytes
String (8-character)	2	56
Date	usually 1, up to 4	24, when 1 object
BigDecimal	usually 1, up to 5	32, when 1 object
Enum	0, uses a shared object	0

Table 3.4. Memory requirements for some common field types that require one or more separate objects. Sizes do not include a referring pointer.

Here is a more realistic employee class with several reference fields. An employee now has a name, which is a `String`, instead of an integer id. It also has a start date,

which is a `Date`. The type of `salary` has been changed from `double` to `BigDecimal`. `BigDecimal` avoids potential roundoff errors.

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

On 32-bit architectures, a pointer is 4 bytes. You can calculate the size of any object as described in Section 3.1. Assuming the Oracle JRE, the size of an Employee object is 32 bytes:

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

While an instance of the `Employee` class is a single 32 byte object, an entire employee, including name, salary, and start date consists of at least 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.) `BigDecimal` usually requires 1 object but can take up to 5 depending on the usage. The memory layout for a specific employee "John Doe" is shown in Figure 3.1.

A comparison of an `Employee` 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 28% 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, and a pointer for each delegated object, including empty pointer slots for uninitialized object fields. In this example, the 4 new object headers and 7 pointer slots add up to 60 bytes, or 42% of the total memory of an `Employee` record. Delegation may also force additional alignment costs, since each new delegated object has to be aligned to an 8-byte boundary.

Java's Delegation Bias   In the spirit of keeping things simple, Java does not allow you to embed one object in another. You also cannot nest an array inside an object, or store objects directly in an array. You can only point from one object to another. Even the basic data type `String` consists of two objects. Single inheritance is the only language feature that can be used instead of delegation to compose two types, but single inheritance has limited flexibility. This means that delegation is pervasive in Java programs. In contrast, C++ gives you many options: you can embed one type within another, overlay types using unions, employ subclasses with single or multiple inheritance, and point from one type to another.
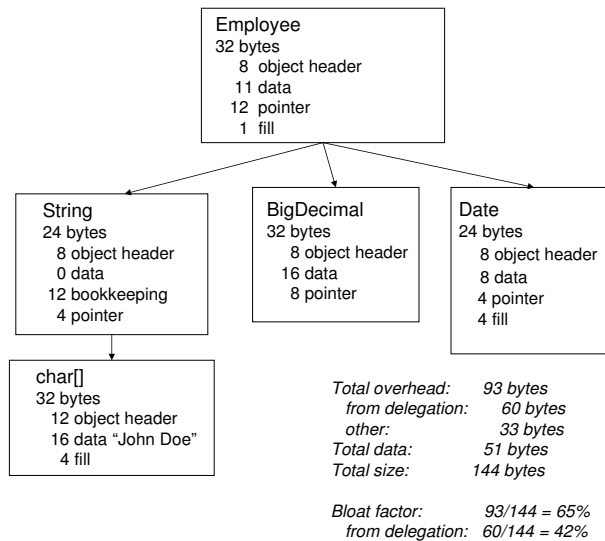
Figure 3.1. The memory layout for an employee, with some field data delegated to additional objects.

Calculating the Cost of a `String`

The size of a `String` can be computed by adding:

- The size of the `String` wrapper = 24 bytes

- The size of the `char[]` = 12 bytes + 2 * number of characters, then round up to a multiple of 8

This is for the Oracle JRE on the 32-bit architecture.

Because of the design of Java, there is a basic delegation cost that is hard to eliminate. 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.

Example   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 preferredMethod;
}

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

abstract class ContactMethod {
}

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 each, 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.

Example, optimized   One object that looks superfluous is `EmergencyContact`, which encapsulates the contact person and the preferred contact method. Removing this delegation involves moving the fields of `EmergencyContact` into other classes, and eliminating the `EmergencyContact` class. Here are the refactored classes:
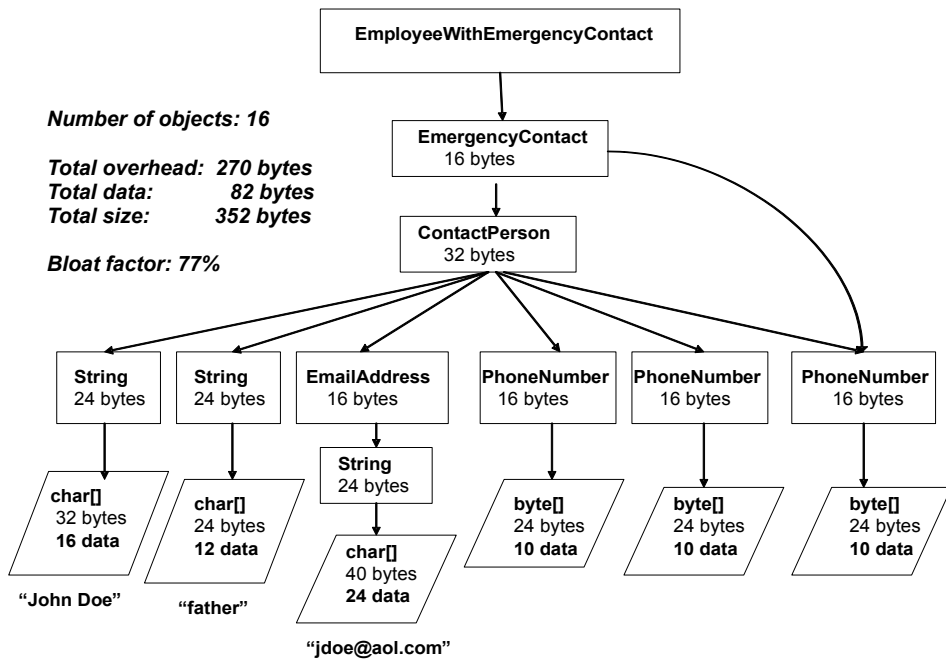
Figure 3.2. The memory layout for an employee with an emergency contact, using a fine-grained design.

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

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

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 enumerated 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 preferredMethod;
    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

Number of objects: 11

Total overhead: 198 bytes
Total data: 82 bytes
Total size: 280 bytes

Bloat factor: 71%

EmployeeWithEmergencyContact

ContactPerson
40 bytes

String
24 bytes

String
24 bytes

String
24 bytes

byte[]
24 bytes
10 data

byte[]
24 bytes
10 data

byte[]
24 bytes
10 data

char[]
32 bytes
16 data

char[]
24 bytes
12 data

char[]
40 bytes
24 data

"John Doe"

"father"

"jdoe@aol.com"
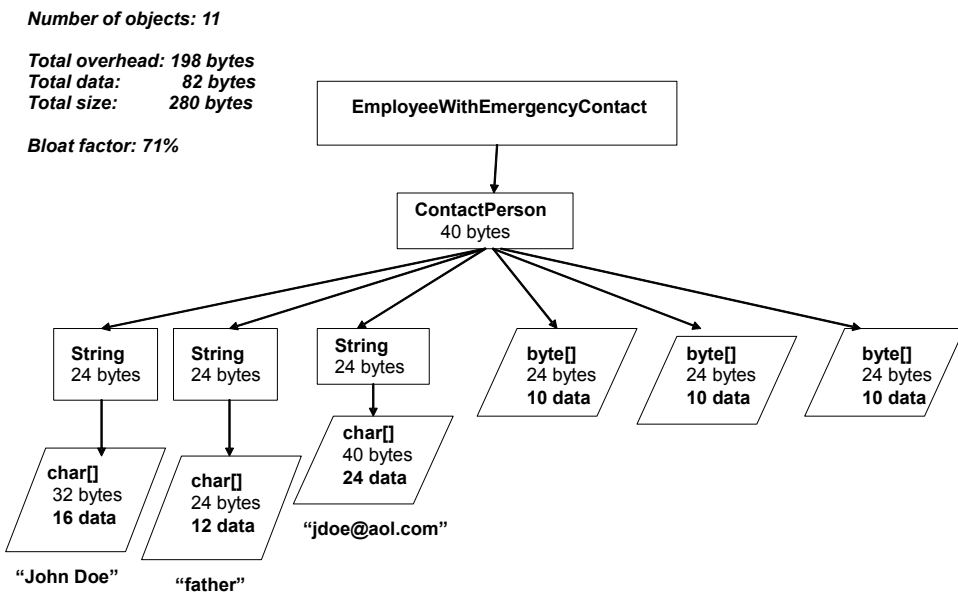
Figure 3.3. The memory layout for an emergency contact, with a more streamlined design.
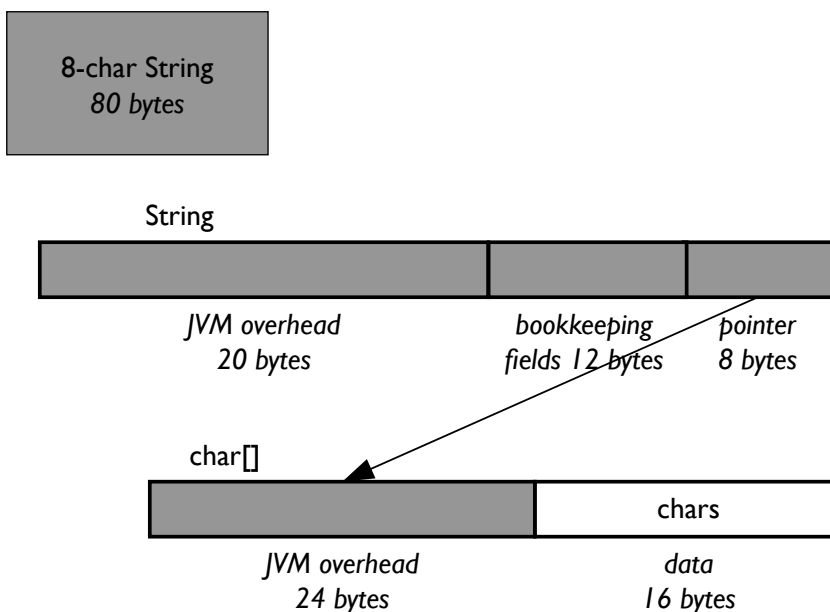
Figure 3.4. The memory layout for an 8-character string on a 64-bit JRE.

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   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 [2] 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 3.4. The 64-bit string is 43% bigger than the 32-bit string. All of the additional cost is overhead. Fine-grained designs and those with a lot of pointers will suffer the most when moving to 64-bit architectures.

Fortunately, in practice things are not always so bad. Both the Oracle and the IBM JREs have implemented a scheme for compressing addresses that avoids this blowup, provided that the heap is less than 32 gigabytes. Address compression is implemented by stealing a few unused bits from 32-bit pointers. Objects are laid out just like in a 32-bit address space. During execution, 32-bit compressed addresses are converted into 64-bit native addresses, and vice-versa. See Appendix A for the specifics of using this feature.

3.5   Summary

The decisions you make about the *granularity* of your design — the number of objects you use to model each record — will have a big impact on the memory needs of your system. Software engineering best practices and Java's design both encourage having many small objects. Since each object comes with memory overhead, it's easy to produce a data model with a high bloat factor. Estimating your memory costs will let you weigh the importance of flexibility and other software engineering goals as you map your data into classes. To estimate the size of your data, consider:

- The size of each object is the sum of the object header and the field sizes, rounded up to an alignment boundary. Header size, alignment, and field packing rules all depend on the JRE (see Appendix A).

- Each time fields are *delegated* to a separate object, a new object header and possibly an alignment overhead are introduced, plus space for a pointer. While one level of indirection may not seem like much, these overheads add up quickly in a fine-grained design with many small objects. This is a common reason for memory bloat in many applications.

- Field types that require one or more separate objects, like `String`, `Date` and `BigDecimal`, can add a lot of overhead. To estimate the cost of a data type, include all of the objects directly or indirectly hanging off of it.

- Memory overhead will be *much higher* when you switch to a 64-bit architecture, unless you can take advantage of compressed addressing. That's especially true for fine-grained designs.

Java gives you few modeling options compared to a language like C++, as shown in Table 3.5. You are often forced to create extra objects to solve a design problem, making your alternatives expensive. The next two chapters guide you through the cost tradeoffs of the most common patterns when designing data types.

Feature	Java	C++
Point from one class or array to another	yes	yes
Embed one class or array within another		yes
Single inheritance	yes	yes
Multiple inheritance		yes
Union types		yes

Table 3.5. Java's simpler approach to modeling means designs rely heavily on delegation.

Chapter 4

FIELD PATTERNS FOR EFFICIENT OBJECTS

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

Side Objects   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;
    ..
    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: an extra pointer and object header, totaling 12 bytes. 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 8 bytes per object on average, when the fill rate is 0%. That's 67% of the size of the fields that were delegated. When the fill rate is 10%, only 50% of the delegated field bytes are saved on average. 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 and their sizes. The more bytes delegated, the larger the memory savings, assuming the same fill rate. Figure 4.2 shows the memory saved or wasted for different fill rates and 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 delegation overhead becomes less important. At larger sizes there is less of a chance that underestimating the fill rate will cause you to waste memory, and if it does, the memory lost will be relatively small.
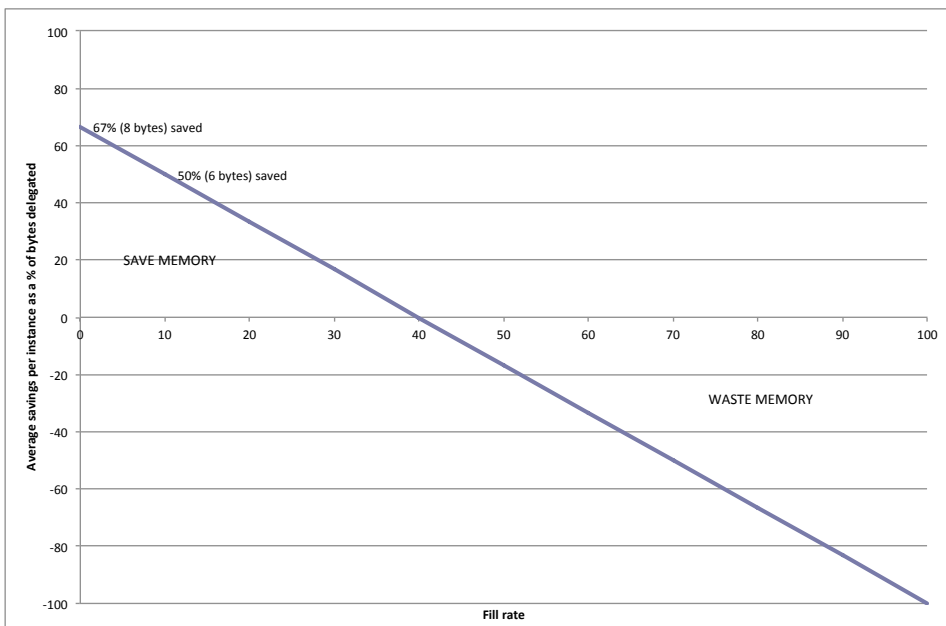
Figure 4.1. This plot shows how much memory is saved or wasted by delegating 12 bytes of memory to a side object. The x-axis is the fill rate. The y-axis is the average memory saved per instance, as a percentage of the bytes delegated.
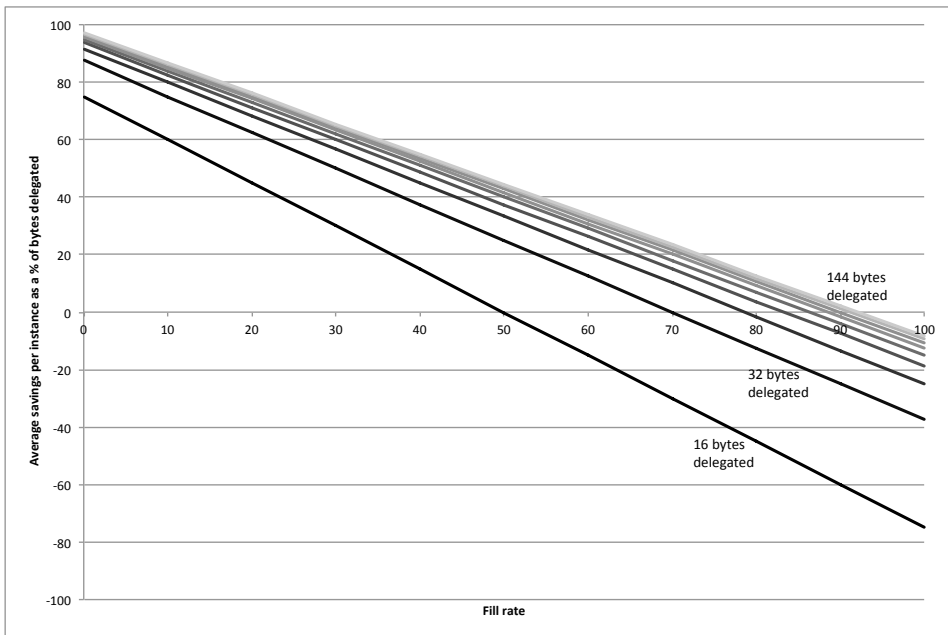
Figure 4.2. This plot shows how much memory is saved or wasted depending on how many bytes are delegated to a side object. The x-axis is the fill rate. The y-axis is the average memory saved per instance, as a percentage of the bytes delegated. Each line represents a different delegated-fields size, from 16 to 144 bytes, in increments of 16 bytes.

Delegation Savings Calculation

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

$$B = \textit{the size in bytes of the delegated fields}$$
$$F = \textit{the fill rate as a number between } 0 \textit{ and } 1$$

While every object pays a 4-byte pointer cost, only F of them pay for a side object. Therefore, using a side object will result in an average savings per object of:

$$B - 4 \quad - \quad F(B + 8)$$

For the savings to be positive, the following must be true: [a]

$$F < \frac{B - 4}{B + 8}$$

---

[a]This calculation does not include alignment overhead. Delegating fields to a side object may increase alignment costs, depending on the other fields in your object. If the header size is not a multiple of the alignment (e.g. on the IBM 32-bit JRE), delegation can sometimes reduce alignment costs.

A common error is to put rarely used fields in a side class with lazy allocation, but have code paths that cause the side object to be allocated all the time, even when it's not needed. 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 also be error-prone because it may require testing whether the object exists at every use. If you need concurrent access to the data in the side object, you have to take special care to code the checks correctly to avoid race conditions. This complexity has to be weighed against potential memory savings.

Side Tables   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 table that maps objects to attribute values.   For example, suppose that only 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 that maps a product `sku` to an award.

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

Even though a `HashMap` has its own high overhead, this design can come out ahead if there are a small number of major awards. You can do a similar analysis as you would for side objects to decide if this approach is worth it. A hash entry typically takes more memory than a side object (see Section 9.1). Therefore, a side table will start wasting memory at a lower fill rate than would a side object approach. Whenever you add a new table like this you also have to be careful not to introduce a memory leak. If products are no longer needed and are garbage collected, the corresponding entries in the table must be cleaned up. This topic is discussed at length in Section 12.5.2.

Subclassing   The least expensive way to model rarely used fields is to move them to a subclass. Unlike the optimizations we've discussed, subclassing costs no extra space. It can have some disadvantages from a software engineering standpoint, however, when compared with delegation or a side table approach. It can make your code less flexible, making it more difficult to add or rearrange functionality later on. Since Java supports only single inheritance, defining a subclass for rarely used fields will prevent you from having subclasses for other purposes. Using delegation or a side table also allows your data to be more dynamic. Rarely used fields can be added after an instance is created.

4.2   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 {
    ..
    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:

```
    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;
    }

    class Clothing extends Product {
        ..
        private ClothingSize   size;
        ..
        public SMLSize getSMLSize() {..}
        public void setSMLSize(SMLSize smlSize) {..}

        public NumericSize getNumericSize() {..}
        public void setNumericSize(NumericSize numericSize) {..}

        public PetiteSize getPetiteSize() {..}
        public void setPetiteSize(PetiteSize PetiteSize) {..}
```

```
    public void setNumericSize(WomensSize womensSize) {..}
    public WomensSize getWomensSize() {..}
    ..
}
```

You can easily write access and update methods that translate to and from the original types, so that the caller is insulated from this storage trick.

If the mutually exclusive fields are of different classes, you can generalize their 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.

Just like rarely used fields, mutually exclusive fields can also be modeled using subclassing. Again, the space savings have to be weighed against other software engineering goals.

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 the class `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:

```
public abstract class Product {
    public abstract Category getCategory();
    ..
}

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

```
        public Category getCategory() {return bookCategory;}
        ..
    }

    class Music extends Product {
        static Category musicCategory;  // Points to the
                                        // music category
                                        //    object
        public Category getCategory() {return musicCategory;}
        ..
    }

    class Clothing extends Product {
        static Category clothingCategory; // Points to the
                                          // clothing category
                                          // object
        public Category getCategory() {return clothingCategory;}
        ..
    }
```

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   Nonstatic Member Classes

Sometimes it is useful to define a class within a larger class or method. Java lets you create four kinds of nested classes, each for a different purpose. They are *static member*, *nonstatic member*, *local*, and *anonymous* classes. Instances of the latter three are always created within the context of an instance of the enclosing class. Their methods can refer back to the enclosing instance. To accomplish this, these three kinds of nested classes maintain an extra, hidden field, the `this` pointer of the enclosing instance. In contrast, instances of a static member class do not have this extra field. They are created independently of any instances of the enclosing class.

A common error is to declare a nonstatic member class when you don't really need to maintain a link with an enclosing instance. To illustrate, let's return to the example from Section 4.1, where we moved some rarely used fields into a side class, `Supplier`. Suppose we wanted to hide this decision, so that we would have the freedom to change our minds later if the optimization didn't work out. We could make `Supplier` a member class of Product, as follows:

```
    class Product {
        class Supplier {
            String supplierName;
```

```
        String supplierAddress;
        String sku;
    }


    ..
    Supplier alternateSupplier;
    ..
    public void setAlternateSupplierName(String name) {
        // Lazily allocate the alternate supplier object
        if (alternateSupplier == null) {
            alternateSupplier = new Supplier();
        }
        alternateSupplier.supplierName = name;
    }
    ..
}
```

Since we didn't declare `Supplier` to be static, every instance of `Supplier` will contain an extra pointer back to the product which created it. In cases like this, a static member class can work just as well, and save the 4-byte pointer field. The only code change is to add the keyword `static` to the declaration of `Supplier`. Notice how Java makes it easy to make this mistake, since the `new` statement that creates an instance of `Supplier` looks the same either way. In the nonstatic case, it hides the fact that it's passing in the enclosing `this` pointer.

The book Effective Java [3] gives more reasons to "favor static member classes over nonstatic" when you don't need to maintain a link to the outer instance. For example, the hidden pointer can lead to a memory leak, by inadvertently holding on to the enclosing instance longer than it's needed.

4.5   Redundant Fields

A field is redundant if it can be computed on the fly from other fields, and, in principle, can be eliminated. In the simplest case, two fields store the same information but in different forms, since the two fields are used for different purposes. For example, product IDs are more efficiently compared as `ints`, but more easily printed as `Strings`. Since it is possible to convert one representation into the other, storing both forms is not necessary, and only makes sense if there is a large performance 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:

- Computed `String` fields should be used only if there's a good reason, 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 computations. 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.

If you do need to store a computed field, make sure that you are using the most efficient representation. For example, `StringBuffer` is useful for building a character string, but is less space-efficient than `String` for storing the final result. Chapter 5 compares different ways to represent some common datatypes. Another thing to watch for is storing many copies of the same computed value. Even something as simple as an empty `String` will add up if you have a lot of them. Chapter 6 looks at ways to share read-only data.

4.6   Large Base Classes and Fine-grained Designs

As discussed in the previous chapter, 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 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 creation and update information.

```
class UpdateInfo {
    Date creationDate;
    Party enteredBy;
    Date updateDate;
    Party updatedBy;
}
```

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

Returning to the original, unoptimized version of `EmployeeWithEmergencyContact` in Section 3.3, suppose that updates to employee emergency contacts need to be tracked. You need to decide how fine the tracking should be. Should every update
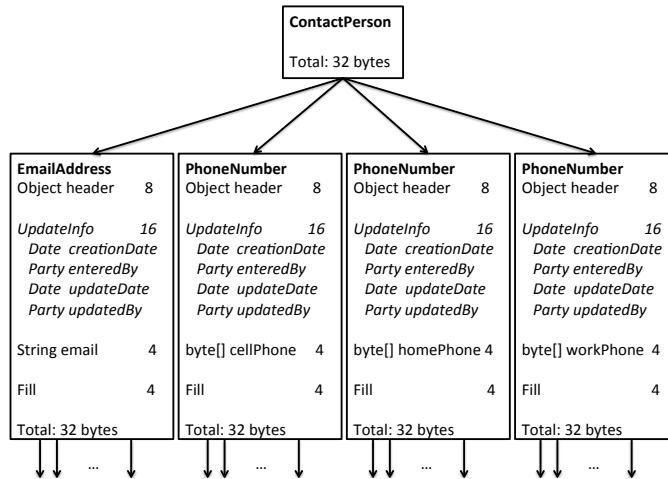
Figure 4.3. The cost of associating `UpdateInfo` with every `ContactMethod`.

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:

```
abstract class ContactMethod extends UpdateInfo {
}
```

Figure 4.3 shows an instance of a 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 4.3 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 scalability 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 to define a subclass without looking

closely at the memory size of its superclasses, especially if the inheritance chain is long.

4.7   Writing Efficient 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 4.4. 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 used. In practice, most Java applications have far 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. [1]

---

[1]There may be some benefit in saving the hashcode in a `String` that's used for looking up a map entry, but only when the same `String` object is used for lookup repeatedly.
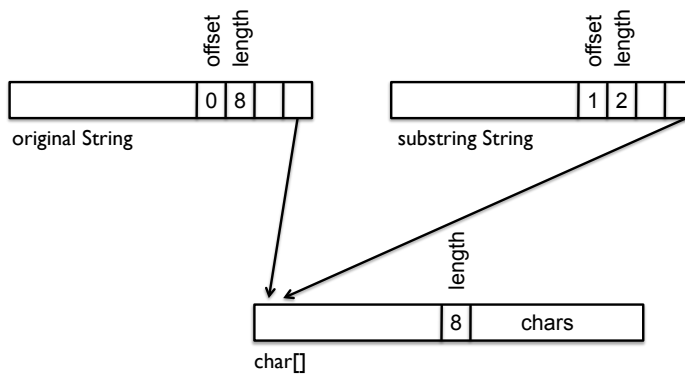
Figure 4.4. A string and a substring share the same character array. The length and offset fields are needed only in substrings. In all other strings, the offset is 0 and the length is redundant.

This is a cautionary tale of premature optimization. Framework decisions can have a long-lived impact. At the same time, it is difficult to test new frameworks in realistic settings, or to even predict how they will be used. For these `String` optimizations, it's not clear that there is any performance gain in real-world applications, as opposed to benchmarks. Meanwhile all applications must pay the price in memory footprint.

4.8   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.

- Mutually exclusive fields can share the same field, provided they have the same type.

- Fields that have the same value in all instances of a class can be declared static.

- Redundant fields whose value depends on the value of other fields can be eliminated, and recomputed each time they are used.

- Inner classes which can be made static will avoid storing a hidden `this` pointer.

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 a lot of instances, then these small optimizations turn out to be significant. They are especially useful in base classes, where their effect can be multiplied.

Many of these optimizations do not come for free. They trade one kind of cost for another, and can easily make things worse, depending on the context. So it's important to look at how your data will be used, and to estimate and then measure the effect of any optimizations. It's also a good idea to design your APIs so that classes can be easily refactored later on, by:

- Exposing interfaces rather than concrete classes.

- Exposing factory methods rather than constructors.

Chapter 5

REPRESENTING FIELD VALUES

So far, we have been concerned with the wasteful overhead that can result when modeling entities as classes and fields. But what about the field data itself? Java gives you a number of different ways to represent common datatypes, such as strings, numbers, dates, and bit flags. Depending on which representation you choose, the overhead costs can vary quite a bit. These costs, hidden in the implementation, are not obvious, and can be surprisingly high. In this chapter, we look at the costs of different representations for the most common datatypes.

5.1 Character Strings

`Strings` are the most common non-trivial data type found in Java programs. They are the single largest consumer of memory in most applications, typically taking up 40-50% of the heap. We discuss when to use a Java `String` to represent data, and when not to.

5.1.1 Scalars vs. Character Strings

Character strings are the universal data type, in that all data can be represented in string form. For example, I am now typing the integer 347 as a string of characters in this paragraph, and it is stored in a file as a sequence of characters. Similarly decimal numbers, dates, and boolean values can be represented as character strings.

In Java programs, it's very common to see scalar data represented as instances of the Java `String` class. This representation makes some sense if the data is read in and/or written out as characters, since it avoids the cost of conversion. However, there are several good reasons why it's better to represent data as scalars whenever possible. First, if you need to perform any kind of operations on the data, you will need to convert it to use available datatype operators. If these conversions are performed repeatedly, then there may be lots of temporaries generated needlessly. Second, representing data with specific data types leads to better type checking to avoid bugs. Third, specific data types provide a simple form of documentation. Last but not least, the memory overhead of a string representation is much higher than the overhead of scalars and even of boxed forms.

Table 5.1 shows three examples of the cost of different representations of the

	Example	Size
integer	int anInt = 47;	4
	Integer anInt = new Integer(47);	20 (4+16)
	String anInt = new String("47");	44 (4+40)
boolean	boolean aBool = true;	1
	Boolean aBool = new Boolean(true):	20 (4+16)
	String aBool = new String("T");	44 (4+40)
enumerated type	enum Gender {MASCULINE, FEMININE, NEUTER}; Gender aGender = MASCULINE;	4
	String aGender = new String("masculine");	60 (4+56)

Table 5.1. The cost of different ways to represent an integer, a boolean, and an enumerated type. The size column shows both the field size and the cost of additional delegated objects.

same value. In the first example, an integer 47 can be represented as a 4-byte scalar field, a boxed scalar, or a Java `String`. The `String` is by far the costliest representation, requiring 11 times the memory of the scalar representation. It has a bloat factor of 95%. The effect is similar in the other two examples, a boolean and an enumerated type. The blowup in memory cost for the `String` representation is a factor of 44 and 15, respectively. In Java, `Strings` are a very expensive way to represent scalar values, much more so than in many other languages.

5.1.2   StringBuffer vs. String

Java provides a few different datatypes for character strings, each one addressing a common use case. `Strings` are immutable, meant for string data that never changes once initialized. `StringBuffers` and `StringBuilders`, on the other hand, are for string data that continues to be updated over time.

Using long-lived `StringBuffers` to store stable character strings can waste memory[1]. That's because `StringBuffers` were designed for building string data over time. They allocate excess capacity to reduce the time needed for reallocating the character array and copying the data. `StringBuffers` will usually have significant 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 a `StringBuffer` to facilitate building a `String` is fine, as long as it is only used as a temporary.

5.2   Bit Flags

A value that can be represented by a single bit seems pretty innocuous from a memory point of view. However, when an entity contains a number of bit fields, then it's worth looking at the cost implications of the different ways you can represent these fields. As an example, let's consider a business, open seven days a week, where employees are assigned to work on different fixed days. We compare three different ways of representing work days as bit flags in an employee record.

First, you can represent bit flags as boolean fields in an object. For example employee working days can be directly stored in an employee record:

```
public class Employee {
    boolean monday, tuesday, wednesday, thursday, friday
        , saturday, sunday;

    public void setWorkMonday(boolean flag) {
        monday = flag;
```

---

[1]This discussion applies equally to `StringBuffer` and `StringBuilder`.

```
        }
        ..
    }
```

Each boolean field takes one byte, so each employee requires at least seven bytes to store workday information.

Alternatively, you can represent bit flags very compactly as actual bits, and manipulate the bits indirectly via accessor and update methods. There is more code involved, but it is isolated in these few methods. Seven days of the week can be stored as bits in a single `byte` field:

```java
public class Employee {

    public final static byte Monday = 0x01;
    public final static byte Tuesday = 0x02;
    public final static byte Wednesday = 0x04;
    public final static byte Thursday = 0x08;
    public final static byte Friday = 0x10;
    public final static byte Saturday = 0x20;
    public final static byte Sunday = 0x40;

    private byte workdays;

    public void setWorkMonday(boolean flag) {
        if (flag) {
            workdays = (byte)(workdays | Monday);
        } else {
            workdays = (byte)(workdays & ~Monday);
        }
    }
    ..
}
```

While compact, this representation is awkward from a coding and stylistic point of view. A better practice is to represent each bit flag as a constant in an `enum` type, and to store the values of the group of flags as an `EnumSet`.

```java
public class Employee {

    public enum Day {MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY, SUNDAY};
```

Representation	Size
bits in a byte	1
boolean fields	7
EnumSet	28 (4+24)

Table 5.2.   The cost of different ways of storing the seven bit flags in our example, representing days of the week.

```
private EnumSet<Day> workdays = EnumSet.noneOf(Day.
    class);

public void setWorkday(Day day) {
    if (flag) {
        workdays.add(day);
    } else {
        workdays.remove(day);
    }
}
}
```

Since an `EnumSet` is an object, this representation is going to cost more, in part because of delegation. On the Oracle JRE, the storage is pretty well optimized. If the underlying `enum` type has up to 64 constants, then an `EnumSet` requires just a single object of size 24 bytes. If the `enum` type has more than 64 constants, then an `EnumSet` requires two objects, a wrapper plus a `long[]` array. The total cost in that case is 40 bytes plus enough 8-byte `longs` in the array to hold the bit flags. In our example, the cost of storing workdays is 28 bytes per employee: 4 bytes for the reference field and 24 bytes for the `EnumSet`. This is considerably more expensive than storing the bit flags as either bits or booleans. That is, unless your entity has a large number of bit flags that form a natural grouping. The crossover point in cost, versus a boolean representation, is 28 flags[2].

The `EnumSet` representation allows for sharing in some cases, which can reduce the cost considerably. Suppose that there are 1000 employees. The cost of storing the workdays for these employees using an `EnumSet` is 28,000 bytes. Now suppose 900 employees work Monday through Friday. These employees can share an `EnumSet` representing these normal working days, reducing the cost to 6,424 bytes. See Chapter 6 for more on sharing data.

---

[2]On a JRE that doesn't pack `boolean` fields tightly, such as the IBM JRE, the crossover point is much lower.

5.3   Dates

Dates are very common in applications, but representing a date as a data type can be complex. The complexity comes from the need to represent universal time and to support conversions, arithmetic operations, and external representations. In Java, there are different ways of representing times and dates, and the more functionality you want, the more memory the representation takes up.

The simplest and most compact way is to represent a time and date as a `long` integer:

```
long timeNow = System.currentTimeMillis();
```

The method call `System.currentTimeMillis()` returns the current date and time in milliseconds since January 1, 1970. This representation is perfect for time stamping, performance timings, and relative time comparisons. However, it is clearly limited.

For more functionality and more precise typing you can use the `Date` class:

```
Date date = new Date();
```

This creates a relatively small object (24 bytes) that stores the current date and time in milliseconds since January 1, 1970. The class `Date` supports little other functionality itself. Most of its original methods for parsing, formatting and conversion have been deprecated and moved to supporting classes.

If you should call one of `Date`'s deprecated methods, such as `getYear()`, the date will create and retain a large (96-byte) internal object, bringing the total cost to 120 bytes for the two objects. Unfortunately, the `toString()` method, which is not deprecated, causes this same effect. So beware of calling `toString()`, explicitly or implicitly, to format long-lived dates, as in:

```
static LogEvent log[];
..
log.println("Timestamp: " + log[i].date);
```

The recommended way to parse and format `Dates` is to use a `DateFormat` object. For example:

```
SimpleDateFormat dateFormat =
    new SimpleDateFormat("dd/MM/yy");
System.out.println(dateFormat.format(date));
```

To work with human-oriented date and time units, such as months, days of the week, and hours, use a `Calendar`. For example:

Representation	Size	Comments
long	8	When minimal functionality is needed
`Date`	28 (24 + 4)	Expands if deprecated methods or `toString` are called
`Calendar`	428 (424+4)	Not recommended for storage of dates

Table 5.3. The cost of different ways of storing a date field.

```
GregorianCalendar calendar = new GregorianCalendar();

// Compute a date 3 months out from the given date
calendar.setTime(myDate);
int month = calendar.get(Calendar.MONTH);
calendar.add(Calendar.MONTH, 3);
newDate = calendar.getDate();
..
```

This functionality can come at a high cost in space and time if not used carefully. It turns out that `SimpleDateFormat` and `GregorianCalendar` each require a lot of storage and have lengthy initialization sequences. A `GregorianCalendar` created with the default constructor, for example, requires six objects, totaling 424 bytes. This doesn't include the many temporaries thrown off during the initialization process. This means that if you are operating on a large number of `Dates` over time, it's best to maintain a small number of formatting and calendar objects and reuse them. Keep in mind that neither `SimpleDateFormat` nor `GregorianCalendar` is thread safe, so for concurrent applications it's useful to maintain a separate object for each thread, either via a local variable or thread-local storage.

Since `GregorianCalendar` stores the `Date` you are currently working on, some applications use `GregorianCalendar` as their storage format for date fields. This is an extremely expensive representation, at 424 + 4 bytes per date! This is not recommended for long-lived storage of more than a few dates.

5.4   BigInteger and BigDecimal

BigInteger   It is rare that an integer is too big to be represented as an `int` or `long`. However, there are occasions when a `BigInteger` is needed. In one extreme example, the largest prime just discovered is over 17 million digits long! A `BigInteger` provides arbitrary-precision integer arithmetic functions. It is immutable, and exactly mimics the functions of the standard integer, while providing some additional mathematical functions as well. A `BigInteger` almost always requires two objects, a `BigInteger` plus an `int[]`, for a minimum of 48 bytes total. The cost can be

Type	Cost
BigInteger	Common case: 2 objects, minimum 48 bytes
	If value is 0, with certain constructors will be 1 32-byte object
BigDecimal	Common case: 1 32-byte object
	Can switch to 3-object inflated form, minimum of 80 bytes
	On `toString()` and some formatting calls retains formatted `String`

Table 5.4.   The memory costs of `BigInteger` and `BigDecimal`.   Common cases and exceptions.

more, depending on the number of digits. The formula for the total size in bytes is $44 + 4$ times the number of `ints` needed to represent the integer[3]. Because of the cost, unless you need the extra digits, it's best to just use an `int` or `long`.

BigDecimal   Like `BigInteger`, `BigDecimal` performs arbitrary-precision arithmetic (floating point in this case), and is immutable. Primarily, `BigDecimal` is critical for many accounting and financial applications that involve currency, for two reasons. First, Java's `float` and `double` lose precision in computations, whereas `BigDecimal` provides precision to an arbitrary number of digits. It allows you to control the scale, which is the number of digits to the right of the decimal place. Second, `BigDecimal` gives you a choice among a variety of rounding methods, a requirement of many of these applications.

BigDecimal has two forms: a compact form and an inflated form. `BigDecimal` attempts to use the compact form when it can, which is a single 32-byte object that can store a number whose significand's absolute value is less than or equal to `Long.MAX_VALUE`. All 18-digit decimal numbers will fit in that, and some 19-digit. Bigger than that, the inflated form is required. In the inflated form, `BigDecimal` delegates the storage of the significand to a `BigInteger`, which almost always means two additional objects, bringing the total for BigDecimal to 80 bytes at a minimum.

Unfortunately, certain arithmetic operations will cause a `BigDecimal` to switch to its inflated form in order to perform the operation, and once inflated, there's no switching back. That's true for `BigDecimals` that are the `this` object, as well as for some that are innocently passed as arguments. This can also occur indirectly as the result of some constructors and formatting operations. The cases are too numerous to predict, so if there's a concern it's best to look at the heap and see what happens based on actual usage in the application.

In addition, calling `toString()` on a `BigDecimal` causes it to save its string representation in case it is needed again. The same thing occurs in some cases when formatting using `DecimalFormat`. This is an example of a performance optimization

---

[3]If the value is 0, and you use one of the constructors that takes a `String` as argument, the constructor will arrange to share a singleton `int[]`, making the total cost only 32 bytes.

built into the library, one that may not be needed in your application. The result is an extra two objects attached to the `BigDecimal`.

In summary, a `BigDecimal` will require one object under most circumstances, but can take up to five.

5.5   Summary

There are common data types just beyond simple primitives that can take up a lot of space. These include character strings, bit sets, dates, and arbitrary-precision numeric data. For each, there are various representations to choose from. Not surprisingly, the cost varies according to functionality, and the main lesson is not to use expensive representations unless you need the functionality.

- Avoid using `String` to store data that can be naturally represented in a more compact data form, such as `int` or `boolean`.

- Storing long-lived strings in `StringBuffers` or `StringBuilders` can waste a lot of space, since they are often sized much bigger than the data they store.

- Even simple bit flags can cause bloat if you have a lot of these fields per object.

- Beware of storing dates as `GregorianCalendars`. A `GregorianCalendar` should only be used (and reused) as a converter object, to perform operations on `Dates`.

- Use `BigInteger` and `BigDecimal` only when their functionality is really needed.

- When using `Date` and `BigDecimal`, watch out for methods such as `toString()` that cause hidden objects to be created and retained behind the scenes.

Chapter 6

SHARING IMMUTABLE DATA

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 are never modified once they are initialized. And, of course, duplicate strings are extremely common. This chapter describes techniques for sharing read-only data to avoid duplication, including a few low-level mechanisms that Java provides. Section 6.1 looks at sharing literal data, known at compile time. The rest of the chapter describes techniques for sharing more dynamic data.

6.1   Sharing Literals

Duplicate strings are one of the most common sources of memory waste, since even small strings incur a large overhead. It is not uncommon to see heaps with tens of thousands of copies of strings such as "Y" or "N". At 40 bytes a piece, these quickly add up. 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 strings 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.

As an example, consider a document storage system that initializes each document object before reading in metadata about that document. A common mistake is to create a new `String` from a `String` literal:

```
public Document() {
    name = new String("unknown");
    description = new String("unknown");
}
```

Even though the standard library is smart enough to share the underlying character array, this code will still create two new `String` objects for every document. Compare that to the following:

```
public Document() {
    name = "unknown";
    description = "unknown";
}
```

The JRE shares the `String` literal wherever it appears in the code, even across classes. More important, in this example, is that now all instances of `Document` will share a single `String`. A more maintainable approach is to make the sharing explicit, by defining a `final static` constant:

```
public class Document {
    public final static unknown = "unknown";
    ..
    public Document() {
        name = unknown;
        author = unknown;
        ..
    }
}
```

A more dynamic version of the problem often occurs when data originates from an external source. Extending our example, suppose the application reads in the metadata for each document as a set of property name-value pairs. The code below builds each map directly from the input. `getNextString()` returns a new `String` for each property name and value.

```
public class DocumentProperties {
    protected Map<String, String> propertyMap;
    ..

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

The `Strings` stored in each map are created dynamically. In applications of this kind, there is usually a lot of repetition of both property names and values. 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 we know in advance what all of the property names are, then we can define them once as `String` literals. For example:

```
public class DocumentProperties {
    // Property names
    final public static String format = "format";
    final public static String comments = "comments";
    final public static String timestamp = "timestamp";
    ..
}
```

When reading in the property names, we could check against these literals, and share them as the keys in each map. Alternatively, a better stylistic choice would be to encode them as an enumerated type. Like `String` literals, the JRE maintains a single copy of each `enum` constant. Unlike `Strings`, you never have the option to allocate `enum` instances dynamically; they are always shared constants.

```
public class DocumentWithStaticProperties {
    public enum PropertyName =
        {format, comments, creationDate, ... };
    ..
    protected Map<PropertyName, String> properties;
    ..
    void handleNextEntry() {
        PropertyName propertyName =
            PropertyName.valueOf(PropertyName.class,
                getNextString());
        String propertyValue = getNextString();
        properties.put(propertyName, propertyValue); }
}
```

The code reads in a property name, and returns a pointer to a shared `enum` constant. `enum` types automatically provide for efficient lookup by name, in addition to giving you type safety. In our example, using an `enum` type would also allow us to save even more memory by switching from a `HashMap` to the smaller `EnumMap` to store each document's metadata.

Using literals to avoid duplication is only possible when values are known in advance. Section 6.2 introduces the notion of a sharing pool for sharing dynamic data.

6.2   The Sharing Pool Concept

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*, also known as a *canonicalizing map*. A sharing pool will eliminate not just duplicate data, but all of its associated overhead. Since many Java data structures
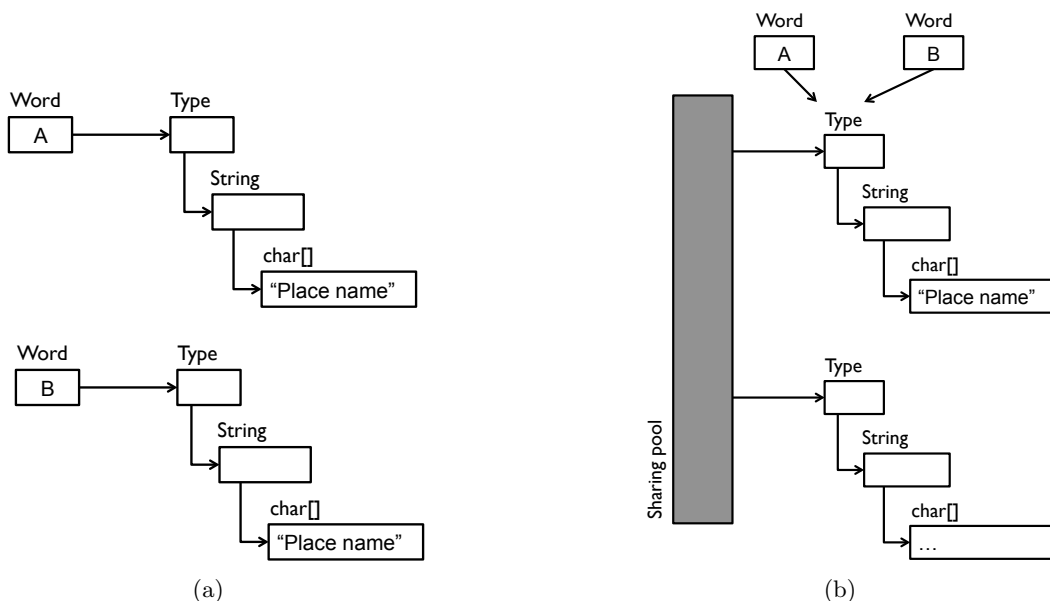
Figure 6.1. (a) Words A and B point to duplicated data. (b) Words A and B share the same data, stored in a sharing pool.

employ delegation in their designs, sharing duplicate data can avoid multiple levels of identical objects.

An example is shown in Figure 6.1. The example is from a text analysis system, which assigns a type to each word as it appears in each document. Each type is identified by a `String` type name. The complete set of word types is known only at run time, so an `enum` type cannot be used. In Figure 6.1(a), A and B are words that have been classified as having the same type. Figure 6.1(b) shows A and B sharing the type structure, which is stored in a sharing pool. In this example, each use of a shared structure saves three objects.

```java
public class Word {
    private int locationInDocument;
    private Type type; // The word's type in context
}

public class Type {
    final private String typeName;
    ..
}
```

> **Sharing Pool**
>
> A *sharing pool* is a centralized structure that stores canonical, read-only data that would otherwise be replicated in many instances. 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 the contents of the Type that A points to in Figure 6.1(b), would also change the type of B.

The result of `equals` **testing should be the same, whether or not objects are shared.** Always use `equals` to compare shared objects; never rely on `==`. Some sharing pool implementations do not guarantee that all duplicates will actually be shared, and the application may not be using the sharing pool for all instances of the type. In Figure 6.1, `A.type == B.type` is false in 6.1(a) and true in 6.1(b), which can lead to very subtle bugs. Avoiding `==` also allows you to safely change your design, should you later decide to share a different set of instances, or to not share data at all. [1]

Sharing pools should not be used if there is limited sharing.   A sharing pool itself can add memory costs, typically the cost of a map entry for each instance stored in the sharing pool. 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. In Section 6.6 is a sample analysis of when it's worth sharing strings.

Sharing pools can have performance costs.   Sharing pools can sometimes add a performance cost when creating new instances to be shared. First, each new instance requires a lookup and possible addition to the sharing pool. Second, in a multithreaded environment, checking and adding to the sharing pool can introduce latency if the sharing pool is synchronized. The use of shared instances, however, does not incur any extra time costs.

Sharing pools should be either stable or garbage collected.   In Figure 6.1(b), the sharing pool stores an object that no other object is pointing to. Because the sharing pool is holding onto the object, the garbage collector is unable to reclaim it, even though it is no longer needed. If the sharing pool is not purged of these unused items, it will cause a memory leak, using up more and more memory over time. However, if there are only a small number of shared instances, or if the set of shared instances is unchanging for the lifetime of the pool, then garbage collection need not be a concern.

---

[1] An argument sometimes heard for sharing data is that it will allow for speedier comparisons, by letting the application use `==` rather than `equals`. In fact, most `equals` methods already perform this optimization, with virtually identical performance to calling `==` directly.

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

6.3   Sharing Strings

Since `String` duplication is so common, Java provides a built-in pool for sharing `Strings`. To share a new `String`, you simply call the method `intern` on it, and everything is taken care of automatically. Both the `String` object and its underlying character array are shared. Since `Strings` are immutable, sharing is safe. However, the rule about not using `==` still holds for shared `Strings`. Remember that only some `Strings` will be shared in any application, namely those specific instances that you choose to intern.

In the example from Section 6.1, `DocumentWithStaticProperties` eliminates property name duplication, but only if all the property names are known in advance. Suppose you know there are not many distinct property names overall, but you don't know what they are in advance. In this case, property names are perfect candidates for `String` interning.

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

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

There is a memory overhead cost for each shared `String`, so interning `Strings` indiscriminately will waste memory. In this example there is likely to be a lot of duplication among some of the property values but not others. Rather than interning the value `Strings` for all the properties, we could add interning for just those properties with the most duplication. For example, the document format property would be a good candidate for interning, since it would have only a few distinct values. On the other hand, a timestamp property would be a poor candidate. In this way we would get the maximum benefit from sharing, while keeping the overhead to a minimum.

The JRE maintains a map in native memory to keep track of the interned `Strings`. The interned `Strings` themselves are stored in a separate heap known as the *perm space*. Exceeding the size of the perm space will result in an exception:

`java.lang.OutOfMemoryError:PermGen space` [2]. There are parameters to adjust the perm space size. See Appendix B for details. Fortunately, the JRE performs garbage collection on the internal string pool, so there is no danger of a memory leak.

The built-in interning mechanism is synchronized, and can incur a latency cost in a multithreaded environment, when new `Strings` are interned. The book Effective Java[3] gives an example of how to build a concurrent sharing mechanism for `Strings`.

6.4   Sharing Boxed Scalars

As of Java 5, the Java library provides sharing pools for `Integers` and some of the other boxed scalars. These pools work differently from `String` interning, where you must always create a new `String` first and then call its `intern` method. To take advantage of the `Integer` pool, simply call the factory method `Integer.valueOf(int i)` whenever you need a new `Integer`. The `Integer` pool is initialized with all the `Integers` in a fixed range, from -128 to 127 by default. The factory method returns a pointer to an `Integer` in the pool, provided `i` is in range; otherwise it returns a new `Integer`. The idea is that the most commonly used integers (at least in many applications) will be shared. There is a parameter to change the upper limit of the range – see Appendix B for details. Note that Java's autoboxing feature uses these same pools to share some of the instances it creates behind the scenes.

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, whenever you need a new `Integer`. The pooling aspect is very cheap, and you never have to worry about garbage collection, wasting memory, or concurrency issues. You do have to be careful, however, to avoid using `==` to compare `Integers`, since only some instances are actually shared. As always, using `equals` is a better practice.

The Java library provides a `valueOf` method for each of the boxed scalars. `Character` and `Short` pools work in a similar fashion to `Integer`, sharing only values within a range. For some types, such as `Float`, there is no sharing actually implemented. For `Boolean` and `Byte`, a shared constant is returned for every possible value. In general, there is no penalty for always using `valueOf`.

The exception to this rule is if you expect to have many copies of values outside the range of what the built-in pool actually shares. In this case you will not get the benefit of the built-in mechanism, and it may be worth implementing your own sharing instead. When sharing something as small as a boxed scalar, however, there must be a very high degree of duplication to make up for the overhead of your sharing mechanism.

---

[2]This is an issue only for the Oracle JRE. The IBM JRE places no fixed limit on interned `Strings`.

6.5   Sharing Your Own Structures

Beyond strings and boxed scalars, there can be other kinds of duplicated data structures that consume large portions of the heap. There is no built-in Java mechanism to share data in general, so you have to implement a sharing pool from scratch. All of the sharing pool issues from Section 6.2 need to be addressed. The shared 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 not cause a memory leak.

There are two common styles of implementing your own sharing pool, depending on the complexity of the data being shared. We illustrate both in this section. For the purposes of this discussion we assume that the set of shared data is always needed throughout the run, so there is no need to worry about garbage collection. In Section 12.5.1 we revisit these two examples, and show how to implement sharing pools with garbage collection. We leave the addition of concurrent access as an exercise.

Sharing simple data.   To illustrate one common style of user-written sharing pool, consider a graph where every node has an annotation, and many of these annotations are duplicates. Each annotation is a single, immutable object, containing just a few scalar fields. The graph is modified dynamically, and a new annotation may be assigned to a node at any time. Assume for now that the same universe of annotations is needed for the duration of the run. The main requirement is the ability to find and retrieve existing annotations quickly to share them.

Interestingly, none of the common collection classes meet this requirement out of the box. A `HashSet` can store `Annotations` uniquely, but retrieving an existing `Annotation` is not easy. The

```
HashMap<Annotation, Annotation>
    canonicalizingMap;
```

`HashSet` can let you know whether an equivalent item already exists in the set, but can not return that item quickly. To get the preexisting item, you would need to iterate over the entire set. The most common approach is to use a `HashMap` that maps the `Annotation` to itself, as shown on the right. This design assumes that an `Annotation` can serve as its own key. In other words, that the `hashcode` and `equals` methods are defined on `Annotation` so that they ensure uniqueness. Note that, unless overridden, `equals` is implemented as `==`, so sharing data structures typically requires writing a new `equals` method.

Callers must create an instance of `Annotation` in order to find out whether it's already in the shared pool. This is similar to the pattern of using `String.intern`, where you create a new `String` in order to see if a matching shared `String` exists. Therefore, this type of canonicalizing map only makes sense when sharing relatively simple data that is inexpensive to create.

Sharing more complex data. Suppose that we would like to share more complex data, such as the type information from Figure 6.1. In this example, the

```
HashMap<String, Type>
    canonicalizingMap;
```

type is uniquely identified by a `String` type name. Each shared structure consists of three objects. Rather than asking the programmer to create a new `Type` structure only to discover that it exists in the pool, we can instead use the type name as the key, as shown on the right. That way callers of the sharing pool only have to create a `String` in order to find or retrieve the `Type`. In this example we save the creation of one object for each new instance of the structure created. For more complex structures the savings will be much greater.

Whichever of these two approaches you use, it's a good idea to hide the use of the sharing pool behind a factory method. That will make it easier to make changes later, should you find that sharing is not worthwhile.

6.6   Quantifying the Savings

Before implementing a sharing scheme, we can estimate the space savings to make sure it's worth the effort, and to make sure it doesn't cause a net gain in memory usage. Here is a sample analysis of sharing `Strings`. This style of analysis can be applied to other types of shared data.

First, we'll need to estimate a few properties of the data we are considering sharing. For this example, we'll start with a set of one million `Strings`, and make the simplifying assumption that they are all the same size, ten characters each. Without any sharing, this data would require 56 bytes per `String`, or 53 Mbytes in total. Next, we'll need the overhead of the sharing mechanism, in this case Java's built-in string interning. The JRE stores its map of shared strings in native memory. We'll assume that the native map costs 20 bytes on average for each distinct value[3].

Finally, we'll look at the degree of duplication in the data set. One useful measure is the ratio of distinct values to the total number of items in the set. A lower fraction, closer to 0, means there are more duplicates, since there are fewer distinct values in the set. A higher value, closer to 1, means there are more distinct values, so less duplication.

Figure 6.2 shows how the cost of sharing strings varies based on this ratio. Each data point compares not sharing (the left bar) against interning (the right bar). There is considerable savings with even a moderate amount of duplication. For example, when distinct values are 50% of the total number of `Strings`, on average 2 identical `Strings` per value, 17 Mbytes are saved. When the percent is above 74%, however, memory is wasted, since there isn't enough duplication to justify the extra overhead.

---

[3]This is an approximation, for illustration purposes, of what a native weak hash map might cost on a 32-bit JRE.
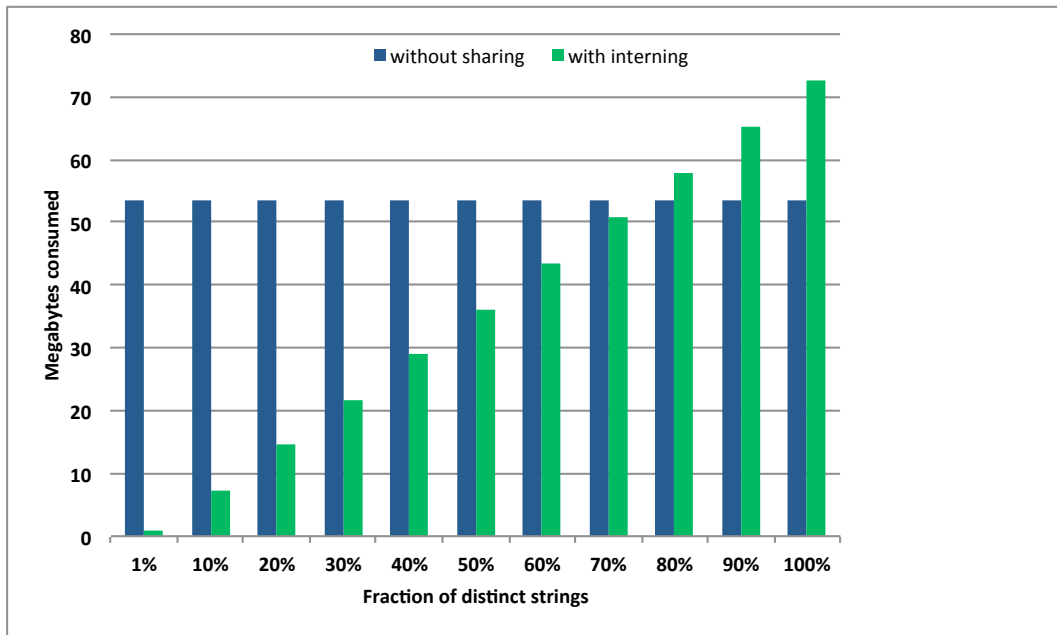
Figure 6.2. Comparing the memory consumed by one million ten-character `Strings`, stored individually vs. with interning. The chart shows how the savings (or waste) varies with the degree of distinctness of the data. 10% means there are only 100,000 distinct strings.

Estimating the Savings from Sharing

Let:

N = *the number of items in the data set*
S = *the average size in bytes of a data item*
D = *the ratio of distinct values to total number of items*
E = *the average per entry overhead of the sharing mechanism*

In a shared implementation, a fraction D of the total items incur the overhead of the sharing pool. The savings is the cost difference between the non-shared and shared versions:

$$N \cdot S \ - \ N \cdot D \cdot (S + E)$$

The savings will be positive whenever:

$$D < \frac{S}{S + E}$$

The callout shows a general formula for approximating the savings of sharing any kind of data. If you use a standard Java `Map` class as your sharing mechanism, the overhead will typically be higher than the overhead of `String` interning. Therefore, your data must have either more duplicates or larger duplicate items, compared to our example, in order to achieve comparable savings. If the map needs to provide for garbage collection, the overhead will be higher still. See Section 9.1 for estimating the per-entry cost of a map.

6.7 Summary

Many Java heaps are filled not only with high-overhead data structures, but with many identical copies of them. Sharing read-only data can provide big memory savings with little downside. There are many techniques available:

- To share a small number of values that are known at compile time, use `String` literals, `enum` constants or `final statics`.

- The JRE maintains a sharing pool for `Strings`. Use `String` interning to make use of this sharing pool.

- The standard library maintains fixed-size pools for a range of `Integers` and some of the other boxed scalars. You can take advantage of these by using `valueOf` factory methods, instead of constructors, to create boxed scalars.

- You can implement a sharing pool for your own datatypes using `Map` classes. To keep down the cost of creating new instances, choose the right pattern for the data: `Map<value, value>` for simple data, and `Map<key, value>` for sharing larger structures.

When sharing data, remember these four rules:

- Shared structures must be immutable.

- The result of `equals` testing should be the same whether or not objects are shared. In application code, use `equals` rather than `==` to test for equality.

- Sharing pools should not be used if there is limited sharing, and the overhead of the pool will outweigh the benefit.

- The pool must provide for garbage collection of shared objects that are no longer needed, unless: 1. the set of shared objects is small, or 2. the set of shared objects is stable for the lifetime of the pool.

The techniques described in this chapter for sharing immutable data can lead to substantial savings for many applications. All of these techniques are within the bounds of standard, object-oriented programming practice. Later in the book, in Chapter 15, we look at bulk storage and sharing techniques that stretch beyond the normal Java box in order to achieve even greater space savings.

Chapter 7

COLLECTIONS: AN INTRODUCTION

Collections are the glue that bind together your data. Whether providing random or sequential access, or enabling lookup 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. Collection overhead typically accounts for 10-15% of the Java heap, and it is not uncommon to see much higher numbers in individual heaps. The way collections are employed can make or break a system's ability to scale up.

This and the next three chapters are about using collections in a space-efficient way. We'll look at typical patterns of collection usage, their costs, and solutions for saving space. We'll see techniques for analyzing how local implementation decisions play out at a larger scale. We will also look at the internal design of a few collection classes.

Chapter 7. Collections: An Introduction Collections serve a number of very different purposes in your application, each with its own best practices as well as traps. The current chapter is a short introduction to issues that are common to any use of collections. It includes a summary of collection resources that are available in the standard and some open source alternative frameworks.

Chapter 8. One-to-many Relationships An important use of collections is to implement one-to-many relationships. These enable quick navigation from an object to related objects via references. This chapter covers the patterns and pitfalls of implementing these 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, and which will continue to increase. 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 commonly-used 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 cost could easily be multiplied by hundreds of thousands or millions of instances.

There is much in the standard collections that is not under your control. 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. Some collection classes let you specify a few options that can help, such as the amount of excess capacity to allocate initially. On the whole, though, they provide few levers for tuning to different situations. They were designed mostly for speed rather than space. They also 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. In addition, the standard APIs can force you into expensive decisions in other parts of your design, such as requiring you to box the scalars that you place in collections.

Fortunately, there are some easy choices you can make that can save a lot of space. Costs vary greatly among different collection classes. The best way to save space is to carefully choose which collection class to use in each situation. For example, a 5-element `ArrayList` with room for growth takes 80 bytes. An equivalent `HashSet` costs 256 bytes, or more than 3 times as much. Like other kinds of in-

frastructure, 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 69% space savings can be achieved if the application can do without features such as uniqueness checking that `HashSet` provides. In Section 2.3 we saw a similar example, achieving a large improvement when real-time maintenance of sort order wasn't needed. In general, understanding your system's requirements and the cost of various collection choices will help you make large reductions in memory.

When looking at the cost of collections, it's important to understand how a particular collection class will work *in your design.* This is because the same collection class will scale differently in different situations. Some collections were only designed to be used at a certain scale. For example, a given map class may work well 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 are pure overhead, a combination of fixed and variable overheads, as discussed in Section 2.4. These determine the way that a given collection class will scale in each situation. The *fixed overhead* is the minimum space needed with or without any elements. The *variable overhead* is the increment of space needed to store each additional element. The way these costs add up depends on the context — whether there are many small collections or a few large ones — as well as on the number of elements stored. High fixed costs take on more significance in small or empty collections, and in particular, when there are many of them. High variable costs matter 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. Some collection classes allocate excess capacity for growth, both at the time the collection is first allocated and later on as the collection grows. This overhead is included in the fixed and variable costs.

We'll analyze the space needs of very small collections (where the number of elements is roughly in the single digits) a little differently from those of larger collections. Tables 8.1 through 8.4 in Chapter 8 show the overhead of small and empty collections for the most commonly-used collection classes. Chapter 9 shows how to compute the overhead of larger collections. Appendix A gives more comprehensive information for additional classes and platforms.

It is helpful to look at collection overhead together with the actual data stored in the collections' elements. If a data structure uses an expensive collection class to store a small amount 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 a design. This analysis will help you choose the right collection for your design, or restructure your data into a more efficient design if necessary.

Finally, we do not recommend implementing your own collection classes, or at least not unless you have exhausted all other possibilities. We recommend first

considering all of the techniques in the next few chapters, as well as some of the more space-efficient open source collection frameworks. Note that the space efficiency of any implementation will be limited by Java's modeling constraints and delegation costs.

7.2   Collections Resources

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

The Standard Collections   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 7.1 is a guide to the resources we discuss in this book.

Alternative Collections Frameworks   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 lifetimes. Home-grown lifetime management mechanisms are a common source of errors in many applications. 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. Many of them actually take up more space than a similar design using the standard collections. As always, there is no substitute for analyzing space usage yourself.

We'll look briefly at four of the most relevant open source collections frameworks. In the next few chapters we'll see a few examples of using these classes to solve specific problems. Table 7.2 gives a summary of the capabilities that we discuss (sometimes only briefly)[1]. This is just a sampling of what's available. We encourage you to further explore these and other frameworks on your own.

Guava   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 a few special-purpose classes, for example some of the immutable collections, that are more space-efficient than their general-purpose equivalents. Guava's `MapMaker` class provides very general support for

---

[1]We use the following versions of these frameworks: Guava r14.0.1, Apache Commons Collections r3.2.1, GNU Trove r3.0.3, and fastutil r6.5.2

Resource	Description	Discussed in
`Collections` statics	Memory-efficient implementations of empty and singleton collections.	Sections 8.4, 8.5.1
	Unmodifiable and synchronized behaviors are added via collection wrappers. Can be costly if used at too fine a granularity.	Sections 8.6.2, 8.6.3
`Arrays` statics	Provides static methods if you need to create simple collection functionality from arrays	Section 2.3 shows one example
`IdentityHashMap`	Lower-cost map when using an object reference as key	Section 9.2
`EnumMap`	Compact map when keys are `Enums`	Section 10.3
`EnumSet`	Compact representation of a set of flags	Section 5.2
`WeakHashMap`	Supports one common scenario for managing object lifetime using weak references.	Section 12.5.2
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 8.6.3
`ConcurrentHashMap`	Hash map when contention is a concern. Avoid use at too fine a granularity.	Section 9.5

Table 7.1. Some useful collection resources in the Java standard library

building caches, sharing pools, and other lifetime management mechanisms, 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.

Apache Commons Collections The Apache Commons Collections framework is an older framework with similar objectives to the Guava framework. Its focus is 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 Collections has not been under active development for a few years. Its API has not been updated to take advantage of generics.

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

fastutil The fastutil framework has similar goals to Trove, primarily time and space efficiency. Like Trove, fastutil provides collections of primitives, 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.

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. When working with collections, keep in mind:

- 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, Java collections use a lot of memory.

- Collections vary widely in their memory usage. Sometimes there is a less expensive choice of collection class available, either from the standard library or from an open source framework. Analyzing your application's requirements,

Feature	Supported by	Discussed in
Primitive collections	fastutil, Trove	Section 9.3
Lighter-weight maps and sets	fastutil, Trove	Section 9.1
Immutable collections	Guava	Section 8.6.1
Small collections	fastutil	Section 8.5.2
Customized storage in entry-based collections	Trove, Commons	Section 9.1
Maps with weak/soft references	Guava, Commons	Section 12.5
Caches and pools	Guava	Section 12.5

Table 7.2. A sampling of memory-related resources available in open source frameworks

specifically which features of a collection you really need, can suggest less expensive choices.

- The same collection class will scale differently depending on its context. Ensuring scalability means analyzing how a collection's fixed and variable overhead 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.

Chapter 8

ONE‑TO‑MANY RELATIONSHIPS

One-to-many relationships are typically implemented in Java using the standard library collection classes. Each object maintains a collection of the objects related to it along a given relationship. Since one-to-many relationships are such an important part of most data models, it is not uncommon for Java applications to need hundreds of thousands, or even millions, of collections. Therefore, simple decisions, like which collection class to choose, when to create collections, and how to initialize them, can make a big difference in memory cost. This chapter shows how to lower memory costs when implementing relationships with collections.

8.1   Choosing the Right Collection for the Task

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.

Using collections for relationships often results in many small or empty collections. That's because for a given relationship there are usually lots of objects that are related to either just a few other objects or to none at all. 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[1]

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

---

[1]We'll use the shorthand *relationship* to mean a one-to-many relationship, unless otherwise noted. We'll also use *small collection* to mean a very small collection, where the number of elements is roughly in the single digits.
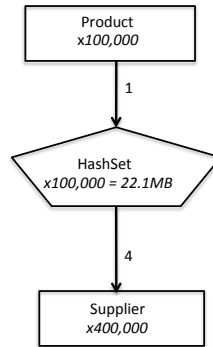
Figure 8.1. A relationship between products and alternate suppliers. Stored as one `HashSet` of alternate `Suppliers` per `Product`.

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;
    ..
    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 22.1MB. This cost is all overhead. It's hard to
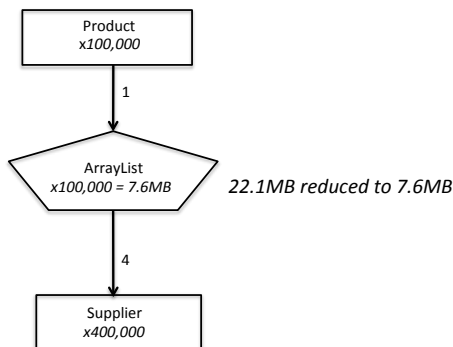
Figure 8.2. A relationship between 100,000 products and alternate suppliers, where the alternate `Suppliers` associated with each `Product` are stored in an `ArrayList`.

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` does maintain uniqueness, but enforcing uniqueness in the data model is not always needed. Many applications perform this check in their loading code. If it is important to guarantee uniqueness in the data model, it can be enforced for an `ArrayList` with little extra checking code, 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 14.5MB.

8.2   Inside Small Collections

Let's look inside a `HashSet` to see why it is so much bigger than an `ArrayList`. The structure of a `HashSet` is shown in Figure 8.3. All collections have a similar basic structure: a wrapper which remains stable, and an internal structure that changes as entries are added and removed.

The standard library designers implemented `HashSet` by delegating its work to a degenerate `HashMap`, that is, one with keys but no values. The `HashSet` object is therefore just a wrapper, and it points to a `HashMap` wrapper. All collections have wrapper objects, but a `HashSet` has two of them.
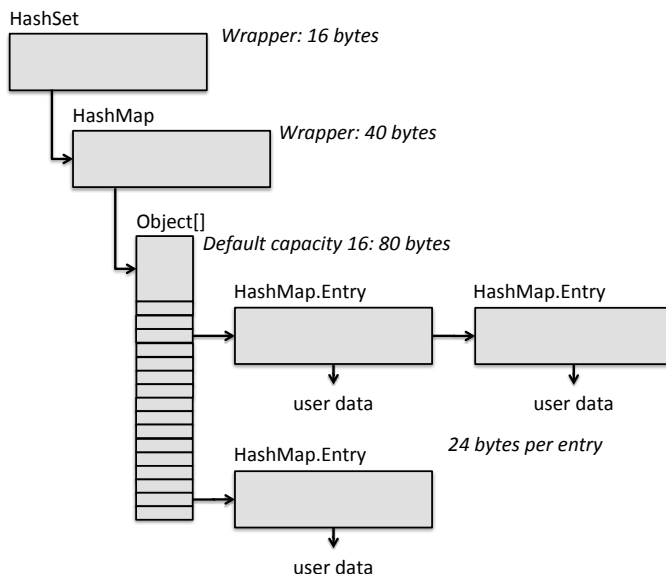
Figure 8.3. A look inside a `HashSet`. Shown with 3 entries.

`HashMap` itself uses a *chaining* design. Its internal structure is an array of hash buckets, with initial size of 16 by default. Each bucket is a linked list of `HashMap.Entry` objects. Each entry object points to an element of the user's data, in other words, to a key and value.

In contrast, `ArrayList` is a simpler structure, as shown in Figure 8.4. It's an expandable array, consisting of a wrapper object and an array. The array points directly to the user data. The array has an initial size of 10 by default. As a result of its simpler design `ArrayList` has a smaller fixed cost and a smaller variable cost than `HashSet`.

The fixed cost is the memory needed before any entries are added. For a `HashSet` the fixed cost consists of two wrapper objects, taking 56 bytes, plus the array's JRE overhead and 16 slots, bringing the total to 136 bytes. We are including the array's empty slots as a fixed cost since they are allocated right from the start [2] [3]. The fixed cost of an `ArrayList` is considerably lower, a total of 80 bytes. That includes its wrapper object and default 10-element array. Fixed costs matter most in small collections. As collections grow they become less significant.

A `HashSet` maintains a `HashMap.Entry` object for each entry, at 24 bytes each.

---

[2]Once a collection grows beyond its initial size, we'll treat excess capacity as a variable cost, as we discuss in the next chapter.

[3]`HashSet` often has another fixed cost, not shown in our figures. The first time you iterate over the set, a 16-byte `HashMap.KeySet` is created and retained for the lifetime of the set.
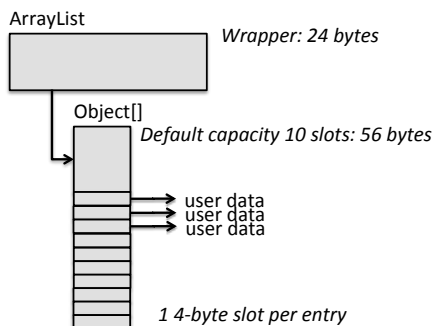
Figure 8.4.   Inside an `ArrayList`.   Shown with three entries and default capacity.
`ArrayList` has a relatively low fixed overhead, and is scalable.

This is its variable cost, that is, the incremental cost of storing an entry. It is much higher than that of an `ArrayList`, which use a 4-byte array slot to point to each entry. A lower variable cost means that `ArrayList` scales much better than `HashSet` for large collections. The variable cost also adds up for small collections, whenever you have a lot of instances of them.

In summary, why does `HashSet` take more space than `ArrayList`? We can see a number of reasons. Some of the extra cost is because of additional functionality, such as providing uniqueness checking and entry removal, in constant time. `HashSet` is also optimized for performance, and sacrifices memory under the assumption that `HashSets` will contain a large number of elements. The decision to reuse the more general `HashMap` code adds to the memory needed, especially the fixed cost. Other extra costs are due to unavoidable Java overhead. 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`. This mismatch between collection implementation and usage is a leading cause of memory bloat.

Table 8.1 compares the memory costs of four common classes from the standard libraries. The table shows bytes needed when each collection contains just a few entries, plus fixed and variable costs for computing the memory needed for small sizes in general. All have been allocated with default capacity. These costs have been calculated based on the Oracle JRE, using the techniques described in Chapter 3. You can calculate costs for similar classes using the same methodology. Appendix A

Collection	with 1 entry	with 4 entries	with n entries		
			fixed	variable	comments
ArrayList	80	80	80	0	for n in 0..10
HashMap	144	216	120	24	for n in 0..12
HashSet	160	232	136	24	for n in 0..12
LinkedList	72	144	48	24	for any n

Table 8.1. Cost of some common collections when they contain very few entries and are allocated with default capacity. Variable cost is the cost per entry, above the fixed cost of the collection. Costs apply only within the specified range.
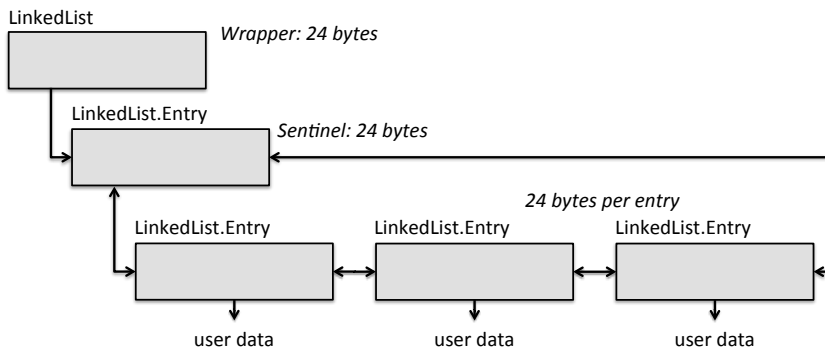
Figure 8.5. A look inside a `LinkedList`. Shown with 3 entries.

has more information about additional classes and JRE platforms.

Let's now look at one more choice, `LinkedList`, useful when ordering is needed and there are frequent insertions or deletions. Figure 8.5 shows its internal structure. A `LinkedList` always has one wrapper object, plus a dummy entry object that acts as an end-of-list sentinel, presumably for performance reasons. The total fixed cost is 48 bytes. `LinkedList`, just like `HashSet`, `HashMap`, and `TreeMap`, is an *entry-based* collection, where an internal entry object is allocated for each element in the collection. Each `LinkedList.Entry` requires 24 bytes, the variable cost of the collection. Entry-based collections have higher variable costs than *array-based* collections, such as `ArrayList`, which point directly to the user data from array slots.

Collection	with 1 entry		with 4 entries	
	default	minimum	default	minimum
ArrayList	80	40	80	56
HashMap	144	80	216	184
HashSet	160	96	232	200

Table 8.2. The effect of capacity on the cost of some small collections, comparing the default capacity with the minimum to accommodate the number of entries. For `HashMap` and `HashSet`, 1 entry requires a capacity of 2, and 4 entries requires a capacity of 8.

Collection	with n entries		
	fixed	variable	comments
ArrayList	36	4	for any n [4]
HashMap	64	24	capacity = 2, holds up to 1
	72	24	capacity = 4, holds up to 3
	88	24	capacity = 8, holds up to 6
HashSet	80	24	capacity = 2, holds up to 1
	88	24	capacity = 4, holds up to 3
	104	24	capacity = 8, holds up to 6

Table 8.3. Fixed and variable costs of some common collection classes when capacity is set to the minimum to accommodate the number of entries.

The combination of fixed and variable costs can make `LinkedList` an expensive choice even at small sizes. From Table 8.1 we can see that a 4-element `LinkedList` is much larger than the equivalent `ArrayList`. Again, for such small collections it's worth asking what the performance gains are in practice, compared to an array-based representation. In the next chapter we'll look at a few examples of array-based maps and sets from some open source frameworks, in the context of larger collections. A few of these are appropriate for small collections as well.

8.3   Properly Sizing Collections

Many collection classes, such as `ArrayList`, `HashMap` and `HashSet`, use arrays in their implementations. When the array becomes full, a larger array is allocated and the contents are copied into the new array. Since allocation and copying can be expensive, these arrays are allocated with extra capacity, to avoid paying these growth costs too often. By default, the initial capacity of an `ArrayList` is 10, and the capacity increases by 50% whenever the array is reallocated. The capacity of a `HashMap` or `HashSet` starts at 16 by default, and grows by a factor of 2 when the

---

[4]Round total cost up to nearest 8 bytes.

collection becomes more than 75% full.

These policies trade space for time, on the assumption that collections always grow. However, many applications have relationships with hundreds of thousands of collections that do not grow once the data has been loaded. Most may never contain more than a few elements. In these cases, the empty array slots can add up to a significant bloat problem, with nothing gained in performance. The same holds true for larger collections that stop growing.

Fortunately, it is often possible to right-size collections at creation time, by specifying an initial capacity. For example, if you know that an `ArrayList` has a maximum size of x, which is less than the default size, then you can set its initial capacity to x when calling its constructor. On the other hand, the standard collections do not give you much control over their growth policies. So if you are wrong and the `ArrayList` grows bigger than x, extra capacity will be allocated, which may be worse than just taking the default.

For an `ArrayList`, another approach is to call its `trimToSize` method, which shrinks the array by eliminating the extra growth space. Since trimming reallocates and copies the array, it is expensive to call `trimToSize` while a collection is still growing. Trimming is appropriate after a collection is fully populated. In applications where the data has a build phase followed by a use phase, the `ArrayLists` can be trimmed between these two phases.

Returning to the example of the relationship between products and alternate suppliers, the `ArrayLists` in Figure 8.2 have been initialized with 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 2.3MB, or another 30%, as shown in Figure 8.6.

`HashSets` and `HashMaps` do not have `trimToSize` methods, but it is possible to set their initial capacity at construction time. The capacity is actually not the number of elements the collection is expected to hold. It specifies instead the number of hash buckets, that is, the number of slots in the array. It is automatically rounded up to the nearest power of two. Hash-based collections always require some excess capacity to reduce the likelihood of collisions. In addition, if the collection does grow and the array needs to be reallocated, there will be the expense of rehashing the entries. The load factor, by default .75, determines the maximum number of elements in the collection, relative to the size of the array, before a larger array needs to be allocated. Therefore, it is important to take the load factor into account when setting capacity. For example, a default `HashMap` with capacity of 16 can hold a maximum of 12 elements before needing to allocate a larger array.

Table 8.2 gives a sense of the savings you can achieve for some common small collections by setting the capacity to the minimum. Table 8.3 shows the breakdown into fixed and variable costs for some minimally-sized collections. You can see,
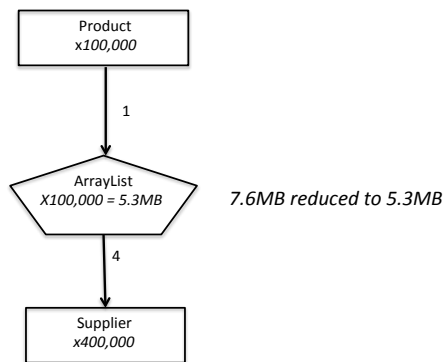
Figure 8.6. The relationship between `Products` and `Suppliers` after all of the `ArrayLists` have been trimmed by calling the `trimToSize` method.
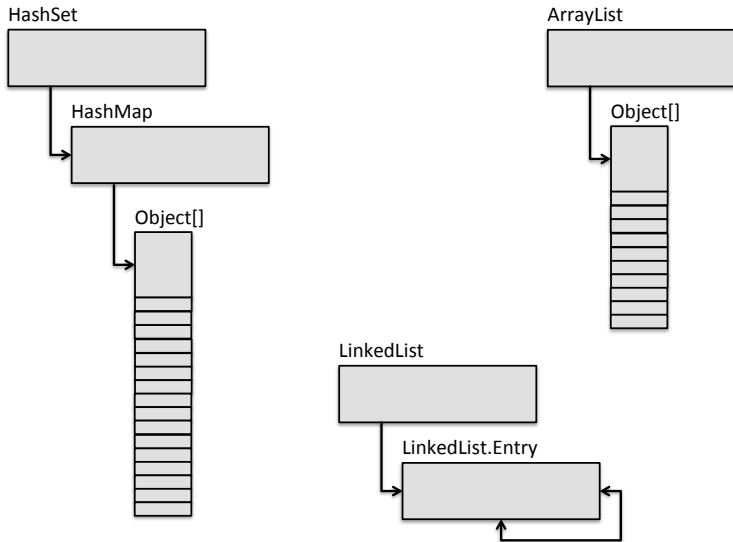
Figure 8.7. The internal structure of three empty collections. Each requires at least two objects, before any entries are added.

compared to Table 8.1, how minimal sizing reduces the fixed cost.

An `ArrayList`, `HashMap`, or `HashSet` will not automatically reduce the size of its array when elements are removed, or when the collection is cleared.

8.4   Avoiding Empty Collections

Maintaining a large number of empty collections is another common problem that leads to memory bloat. Empty collection problems are generally caused by eager initialization, that is, by allocating collections before they are actually needed. You might think that eager initialization would not be a big problem, since entries will be added eventually. However, it's common to find large numbers of collections that remain empty throughout an execution.

Making matters worse, the standard collections allocate their internal objects in an eager fashion. For example, `ArrayList` allocates its internal array before any entries are inserted, and `LinkedList` always allocates a sentinel entry. As a result, every empty collection takes two or more objects, as shown in Figure 8.7. This is true even if you allocate the collection with the minimum possible capacity. Therefore, the smallest empty collections are still quite large. For example, a zero-capacity `ArrayList` requires 40 bytes. Table 8.4 shows the sizes for some common collection classes when empty.

Collection	Size in bytes	
	default capacity	minimum capacity
ArrayList	80	40
LinkedList	48	48
HashMap	120	56
HashSet	136	72

Table 8.4. Size of empty collections, for some common collection classes. Empty collections require a lot of space, even when initialized to the smallest possible capacity.

Example   Suppose the relationship in Figure 8.6 is initialized by the code:

```
class Product {
    ..
    ArrayList<Supplier> alternateSuppliers;
    ..
    public Product() {
        ..
        // Allocate the collection in advance
        alternateSuppliers = new ArrayList<Suppliers>();
        ..
    }
}
```

Initially, each product allocates 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 have no alternate suppliers, there will be 25,000 empty `ArrayLists`, which consume about 1MB even after calling `trimToSize`. Figure 8.8 shows the entity-collection diagram after removing 25,000 empty alternate supplier `ArrayLists`. The diagram now shows only 75,000 alternate supplier `ArrayLists`, since there are no more empty `ArrayLists`. We therefore adjust the average fanout in and out of `ArrayList` to .75 and 5.33, respectively.

Lazy allocation   Delaying allocation is the way to avoid creating lots of empty collections. That is, instead of initializing all of the collections that you think you may need, allocate them on demand. One approach is simply to leave collection fields null until needed. However, this requires extra checking code whenever you access these fields, to avoid `NullPointerExceptions`.

For many applications, the abstract `Collections` class provides a better solution. You can initialize collection fields to point to shared, immutable empty collections, using the static methods `emptySet()`, `emptyList()`, and `emptyMap()` [5]. The following code maps all products to a single, immutable, empty list of suppliers, so that no empty `ArrayLists` are created:

```
class Product {
    ..
    List<Supplier> alternateSuppliers;
    ..
    public Product() {
```

---

[5] Static constants EMPTY_SET, EMPTY_LIST, and EMPTY_MAP provide a similar capability, without the type safety of generics.
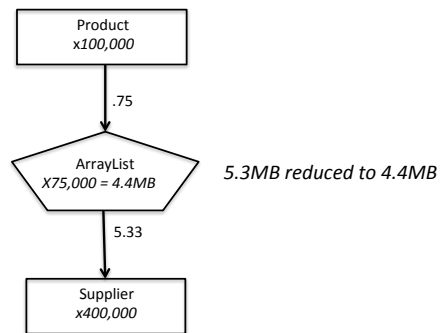
Figure 8.8.   The relationship between `Products` and `Suppliers`, with no empty `ArrayLists`.

```
            ..
            // Initialize to a shared, static collection
            alternateSuppliers = Collections.emptyList();
            ..
        }
    }
```

This initialization avoids the need to check whether an `ArrayList` exists at every use. The size method, iterators, and other access functions work as in any other collection. However, you have to be careful not to let any references to these static empty collections escape their immediate context. If you do give out a reference, then there is no way to update this reference once an actual collection is allocated. Instead, you can provide access and update methods to the relationship, so that the implementation remains hidden. You will also need to code to interfaces, delaying the use of a concrete class until a collection is actually allocated. In this example, `Product` declares a `List` of suppliers, so that it can point to either the shared empty list, or to an `ArrayList` once it's populated. This is, of course, a good practice in general, so that the implementation can be easily changed later.

If lazy allocation is not a good option for your application, then right-sizing collections, at initialization time or after loading is completed, will still make a difference for empty collections.

8.5   Hybrid Representations

8.5.1   Mostly-small collections

It is often the case that the collections used in a relationship are not of uniform size. There often are many collections that are small and fewer that 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 `ArrayLists` for smaller collections, and `HashSets` for larger collections.

One catch is that you will not usually know in advance which collections in the relationship will end up being small and which will grow to be large. Therefore one or more conversion operations will be necessary at some point if a collection grows large enough.

Returning to our original example, let's suppose that our average of 4 alternate suppliers per product is distributed as follows: 25% of products have no alternate suppliers, 25% have one alternate supplier, the next 25% have between 2 and 6, averaging 3 each, and the remaining 25% have an average of 12 each. Let's also assume that we do in fact need to guarantee uniqueness in the data model. The code for the `Product` class is shown below. As in the case of lazy allocation, we'll need to hide access to the collections behind accessor and update methods.

```java
public class Product {

    // Threshold for switching to HashSet
    private static final int arrayListMax = 6;
    ..
    protected Collection<Supplier> alternateSuppliers;
    ..
    public Product() {
        ..
        // Initialize to a shared, empty collection
        alternateSuppliers = Collections.emptyList();
        ..
    }

    public void addAlternateSupplier(Supplier supplier) {
        int numSuppliers = alternateSuppliers.size();
        if (numSuppliers == 0) {
            // Create a singleton list
            alternateSuppliers =
                Collections.singletonList(supplier);
            return;
        }
        if (!(alternateSuppliers instanceof HashSet)) {
            // Uniqueness check for non-HashSet cases
            if (alternateSuppliers.contains(supplier)) {
                return;
            }
            if (numSuppliers == 1 &&
                !(alternateSuppliers instanceof ArrayList)) {
                // Convert to ArrayList
                alternateSuppliers =
                    new ArrayList<Supplier>(
                        alternateSuppliers);
            }
            else if (numSuppliers == arrayListMax) {
                // Convert to HashSet
                alternateSuppliers =
                    new HashSet<Supplier>(alternateSuppliers)
                        ;
            }
        }
        // Add supplier
```

```
        alternateSuppliers.add(supplier);
    }
    ..
}
```

Singleton collections   The standard library class `Collections` provides *singleton collections* that hold one element each. These use much less memory than their more general counterparts. A singleton list and set take only 16 bytes each. Singleton collections can be created via factory methods in `Collections`. In our example, we first initialize the relationship with a shared reference to a static empty set, in case there are no alternate suppliers. The method `addAlternateSupplier` creates a singleton list to hold the first alternate supplier, if one is added. The singleton collections are immutable, in other words, they cannot be modified once they are initialized. In your application, though, if you expect there to be deletions and they are infrequent, it's easy enough to write code that simply deletes the singleton collection until it's needed again.

Converting to larger representations   If another alternate supplier is added, the code replaces the singleton list with an `ArrayList`. We can choose a threshold, say six entries, to be the maximum number of alternate suppliers the `ArrayList` representation should hold. If more alternate suppliers are added beyond that, the representation is switched to the more costly `HashSet`. For the singleton and `ArrayList` representations, we need to check uniqueness, and we use the `contains` method to do so. For `HashSet`, we can rely instead on `HashSet`'s built-in uniqueness checking.

   Implementing hybrid representations is more complicated than just using one type of collection for a relationship. However, it can save significant space in some cases. In our example, an implementation that uses a `HashSet` (minimally sized) and shares a single empty collection would spend 18.9MB on collections overhead. The hybrid representation would reduce that to 11.6MB, a 38% savings.

8.5.2   Load vs. use scenarios

Another scenario where you can benefit from multiple representations is when you have a distinct load phase followed by a phase where you only need read access to the data. If your application requires more expensive functionality at load time, such as uniqueness checking, frequent deletions, or maintenance of insertion order, and you can afford a higher footprint during that phase, a representation such as `HashSet` or `LinkedList` can be a simple solution. Once loading is complete, you can make a second pass over the data and replace the relationship with a more compact collection, such as `ArrayList`. Since the cardinality of each collection is known in advance, it is easy to allocate each `ArrayList` to the right size.

   Depending on how much extra coding and maintenance you are willing to do, you can further optimize by using arrays rather than collections, since you know

the size of each array in advance. This can save you the cost of the `ArrayList` wrapper, or 24 bytes per collection. In general we do not encourage coding your own collection functionality unless you absolutely have to. Most of the complexity of collection behavior is in the update functionality, however. These collections are readonly, which should make coding simpler.

Another alternative, if you would like to guarantee that the use-time collections are readonly, is to use an immutable collection class, such as one from the Guava open source framework. See Section 8.6.1 for details.

The fastutil open source framework provides one more solution for certain cases. Its `ObjectArraySet` class is one of the few collection classes from any framework that uses less memory than the standard `ArrayList`. Its fixed cost is 32 bytes vs. the `ArrayList`'s 40. It provides set behavior backed by a simple array. Its only caveat is that its `contains` function uses a linear search, which will be slow for a large set.

8.6   Special-purpose Collections

Relationships sometimes have additional requirements, such as synchronization or readonly behavior. In some cases, choosing a special-purpose collection for the relationship will also result in a space savings. In other instances, however, collections with specialized functionality, even those that restrict functionality, can have hidden costs compared to their more general counterparts. They may work fine as large collections, but do not scale well when there are many small collections, due to higher fixed costs. In this section we look at a sampling of special-purpose collections, and see how they work in the context of relationships. Table 8.5 shows the memory costs of the special-purpose collections discussed in this chapter.

8.6.1   Immutable behavior

The Guava open source framework includes a set of *immutable* collection classes. They supply the same functionality as some of the common standard collections, but prevent update operations from being performed. This can be useful if you have a relationship that has distinct load and use phases, and you want to guarantee at run time that there are no inadvertent updates, once loading is complete. At the end of loading, simply switch the representation to an immutable collection, as discussed in the previous section. In some applications, the contents of each relationship instance will even be known at load time, and remain constant thereafter. In these cases you can allocate an immutable collection right from the start.

The class `ImmutableList` has the same fixed and variable costs as the standard `ArrayList`, providing the added protection at no additional cost. If you require set functionality, Guava provides an `ImmutableSet` class. It costs more than an `ArrayList`, but considerably less than a `HashSet`. For example, with 4 elements, its total overhead is 96 bytes, compared with a `HashSet`'s 200. If you are willing to trade

a hash-based element lookup for a binary search, Guava's `ImmutableSortedSet` provides further savings, giving you set functionality for the same memory cost as an `ArrayList`.

8.6.2   Unmodifiable behavior

Suppose our developer would like to allow users of the data model to pass around the list of alternate suppliers for each product, but in a way that prevents inadvertent updates. A seemingly simple approach is to implement the relationship with a collection that has *unmodifiable* behavior, and expose this reference as part of the data model's API.

In the standard Java libraries, certain features such as unmodifiable are provided by wrapping an existing collection with a view that augments or restricts the behavior of the underlying collection. The `Collections` class provides static factory methods for this purpose. In contrast to an immutable collection, whose contents will not change, an unmodifiable collection is a view over a collection which may continue to change. The view may be passed to selected users to give them restricted access to the underlying collection. Below is the code to create unmodifiable `ArrayLists` in the data model:

```
public class Product {
    ..
    List<Supplier> alternateSuppliers;
    ..
    public Product() {
        alternateSuppliers = Collections.unmodifiableList(
            new ArrayList());
        ..
    }

}
```

Figure 8.9 shows the E-C diagram when we add the unmodifiable behavior to our solution from Figure 8.6 in Section 8.3, using minimally-sized `ArrayLists`. The cost of the collections increases from 5.3MB to 6.9 MB, or a 28% increase in collection overhead. The inset shows why: each instance of the relationship now incurs an additional fixed cost, that of the view wrapper.

It bears asking whether this feature, which serves a development-time safety-checking purpose, is worth the relatively high run-time cost that results when applying it at such a fine scale. Alternative solutions are to encapsulate access to the relationship, or to turn on the unmodifiable behavior only during testing.
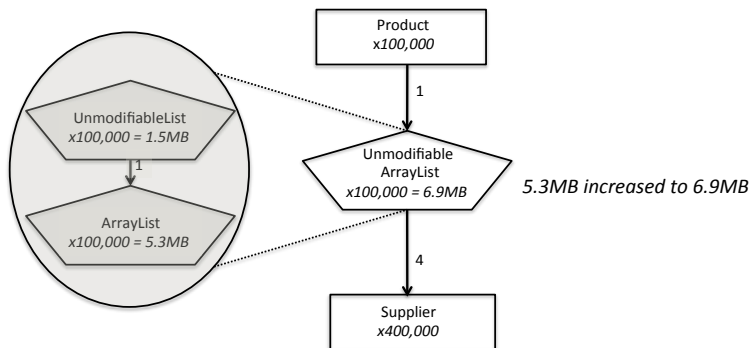
Figure 8.9. A relationship between 100,000 products and alternate suppliers, where the alternate `Suppliers` associated with each `Product` are stored in an `Unmodifiable ArrayList`. The inset shows how each collection now incurs the fixed cost of an additional view layer.

8.6.3 Synchronized behavior

The standard Java collections, such as `ArrayList` and `HashSet`, do not provide for synchronization. Java takes the same approach here as with unmodifiable behavior. To guarantee safe usage of a collection from concurrent threads, you may create a synchronized view over it. Static factory methods from the `Collections` class are provided for this purpose.

As with unmodifiable views, when using this feature for a relationship, the view layer adds a fixed cost (see Table 8.5) that will be multiplied by the number of collection instances. The memory cost of using synchronized behavior at this scale can quickly add up. For a relationship that will have infrequent accesses and updates from concurrent threads, where latency is not an issue, adding synchronized behavior is a good choice. It is a much less expensive choice than using concurrent collections in this context[6].

The original Java 1 collections were written with synchronized behavior built in. The Java 2 framework moved this behavior out of the collections proper. The older collections are still available, and are a less expensive solution in some cases. There are some slight differences in functionality. At the same time, they have been updated to be compatible with the Java 2 collections interfaces and with generics. The `Vector` class provides for synchronization with fixed and variable costs identical to those of the newer `ArrayList`. `Hashtable` has similar costs to `HashMap`, but with a smaller default capacity and more flexibility in setting the initial capacity. There is no Java 1 analogue to `HashSet` or `LinkedList`.

---

[6]In the next chapter we'll look at a more extreme example, of using concurrent collections at a fine granularity, at a huge memory cost.

[7]There is an additional fixed cost if you iterate over the keys or values (not the entries). An unmodifiable view is created over the key set or values collection, and will persist for the lifetime of the original collection.

[8]There is an additional fixed cost if you iterate over the keys, entries, or values. A synchronized view is created over the key set, entry set, or values collection, and will persist for the lifetime of the original collection.

[9]Round total cost up to nearest 8 bytes

[10]Round total cost up to nearest 16 bytes

[11]Round total cost up to nearest 8 bytes

[12]Round total cost up to nearest 8 bytes

[13]Round total cost up to nearest 8 bytes

Collection	with 1 entry	with 4 entries	with n entries		
			fixed	variable	comments
Collections statics:					
SingletonSet	16	-	16	-	
SingletonList	16	-	16	-	
SingletonMap	40	-	40	-	
UnmodifiableList	-	-	16	-	add'l cost
UnmodifiableSet	-	-	16	-	add'l cost
UnmodifiableMap	-	-	24	-	add'l cost [7]
SynchronizedList	-	-	24	-	add'l cost
SynchronizedSet	-	-	16	-	add'l cost
SynchronizedMap	-	-	32	-	add'l cost [8]
Guava:					
ImmutableList	40	56	36	4	for any n [9]
ImmutableSet	64	96	64	8	for any n [10]
ImmutableSortedSet	40	56	36	4	for any n [11]
fastutil:					
ObjectArraySet	32	48	28	4	for any n [12]
Java 1 collections:					
Vector	40	56	36	4	for any n [13]

Table 8.5. A sampling of special-purpose collections and their costs. Assumes minimally-sized collections, when applicable.

8.7   Summary

When collections are used to represent relationships, they often result in many small collection instances. Their cost is dominated by fixed-size overhead. Variable overhead matters as well. To mitigate high memory costs for relationships:

- Choose the most memory-efficient collection for the job at hand. For example, when collections have at most a few elements in them, you don't need expensive functionality like hashing. In order to choose, first ask some questions about the relationship:

  - What operations will be performed? Will there be deletions, insertions, uniqueness checks?

  - What's the expected cardinality? Is it uniformly distributed, or will there be mostly small collections and a few large ones? Will there be many empty collections?

  - Is there a distinct load phase, after which the relationship no longer changes?

- 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. You can postpone creating collections until they are needed.

- When the size distribution is not uniform, it is sometimes reasonable to use a hybrid representation that adapts to the data. If there are distinct load vs. use phases, a different representation for each phase can sometimes be a good solution.

- Some special-purpose collections can help save space, such as `SingletonSet`. Others are not designed for use at a small scale. Make sure the behavior is worth the added cost.

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

INDEXES AND OTHER LARGE COLLECTION STRUCTURES

In a relational database system, data is neatly organized for you into tables. While you may suggest a few fields to index, the structures that allow you to access your data 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 the structures that are the entry points into various groupings of these objects. In this chapter we look at the memory considerations when designing large structures for accessing your data. We'll look briefly at large collections 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, also known as maps, that let you look up data by value. We'll first look at the costs of large collections, and at ways to keep them to a minimum for the task at hand. In the second half of the chapter we'll look at the cost tradeoffs when you need to design more complex access structures made of multiple levels of collections.

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 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 its fixed overhead, such as wrapper objects and array headers, becomes insignificant. For example, in an `ArrayList` with 100 elements, the fixed cost is only 9% of the total. With 1000 elements, it falls to 0.1%. The difference in size in the above example

is pretty dramatic, more than 7:1. That reflects the difference in the variable over-heads of the two collection classes. Like much else in Java, variable overheads in large collections can take up a surprising amount of memory if you are not careful. As a general rule, the variable overhead of array-based collections, like `ArrayList` is much lower than that of entry-based collections where a new entry object is al-located for each element in the collection. You may have noticed that most of the collections in the standard collection library are entry-based, including `HashMap`, `HashSet`, `LinkedList`, and `TreeMap`. Table **??** shows a comparison of variable costs for the most commonly used classes from the standard collections library, along with a sampling from open source libraries.

Excess Capacity   As we saw in the previous chapter, many collection classes allocate excess capacity for performance reasons, mainly to accomodate growth. One differ-ence between small and large collections is the way excess capacity is computed. In the very smallest collections, excess capacity comes from the initial capacity being too large. In contrast, as a collection grows larger, the amount of spare capac-ity allocated is proportional to the number of elements. For example, whenever an `ArrayList` needs more space, it allocates an array that is 50% larger than its current size; `HashMap` doubles the number of buckets when the number of elements reaches a user-specified load factor. So for any collection that's grown beyond its initial size, you can think of spare capacity as part of the variable cost. If an `ArrayList` is 1/3 spare capacity, then every element really costs 6 bytes, rather than the 4 bytes for the array slot that holds the pointer.

Collections grow in jumps, which makes predicting the size of a collection a little tricky. For that reason, Table**??** shows a range of variable costs for each collection. You can use the minimum number if you expect no spare capacity, or in the case of hashtable-based maps and sets, no spare capacity beyond what is required for reasonable operation. The maximum number gives you a worst case, if you added just that one extra element that caused the jump. Slightly less than the midpoint of the range will give you an estimate of the variable cost if the most recently allocated spare capacity is half occupied. Again, the table only applies to collections once they have grown beyond their initial size.

Solutions for reducing excess capacity for larger 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.

Excess capacity isn't much of an issue for collections like `HashMap` and `HashSet`, where the bulk of the overhead is from the entry objects. For larger array-based collections, excess capacity can be a more significant part of the overhead, though

it's relative to a more efficient representation in the first place.

Array-based Hash Maps and Hash 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 **??** in the previous chapter 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 may contain 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, it is also known as closed hashing). In this technique, keys, values, and other information are stored directly in arrays. These are usually parallel arrays, though some implementations use a single array and interleave different kinds of information in successive slots. Chains of entries that map to the same bucket are threaded through the arrays. 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 entry object is so high. fastutil and Trove are examples of open source frameworks that provide open addressing maps and sets that save space. The sets in these frameworks save even more space for a different reason: they are specialized for the task, rather than delegating their work to a more general 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 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 will reduce the space savings to a greater extent than in open chaining implementations. So it's important to look at the entire variable cost when making decisions on which framework to use (see Table **??**). For example, fastutil's object to object open addressing hash map maintains only 9 bytes of data for each element, compared to the standard `HashMap`'s 28. When you add in the greater cost of excess capacity that difference is reduced to approximately 1:2. ¡or example here instead, with absolute numbers¿.

Sharing the Costs in Entry-based Collections   Some alternative frameworks provide another approach to reducing space in entry-based collections. The idea is that your objects become the collection's entry objects, thus saving a delegation overhead. The

downside is that they transfer some of the work of maintaining the collection to your code. One example is in the Apache Commons framework, which lets you subclass its map and map entries. If you have a key with two fields, for example, you would normally have to wrap those fields into a key object. Now you could include those fields in a custom map entry instead, and save a delegation cost. ¡figure here?¿ This introduces a reliability and maintenance problem, however, since it relies on subclassing a fairly complex class. You must now make sure your code doesn't break anything in current and future versions of Commons's hash map implementation.

A simpler approach is taken by Trove's linked list. If you have a class whose objects are to be stored in a linked list, your class can implement the `TLinkable` interface. Your class would provide the chaining, and again, save the cost of a separate entry object. Some restrictions are necessary in order for this to work: no instance of your class can be a member of more than one linked list simultaneously, nor appear more than once in the same list. In addition, your class must now include pointer fields, which increase the cost of these instances when they are not stored in a list.

9.2   Identity Maps

The Java standard library does provide one hash map class that uses an open addressing implementation. The `IdentityMap` class can save you space if you have the need for a large map that fits its requirements. The idea is that it uses the identity of the key object, rather than its contents for comparisons. In other words, it uses `==` against the key your provide, rather than the `.equals()` method. This breaks the `Map` contract, but there are a number of applications where this doesn't matter.

Suppose I want to associate additional information with each product. Using a standard map, I could map the product's unique key, say SKU,

[Discuss trove/fastutil here too? If so, add to table in Intro chapter.]

9.3   Maps with Scalar Keys or Values

9.4   Multikey Maps

9.4.1   Example: Evaluating Three Alternative Designs

9.5   Concurrency and Multilevel Indexes

9.6   Multivalue Maps

9.7   Summary

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, which are also visualized in Figure 11.1. This chapter steps through these requirements, defining them and giving examples 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.
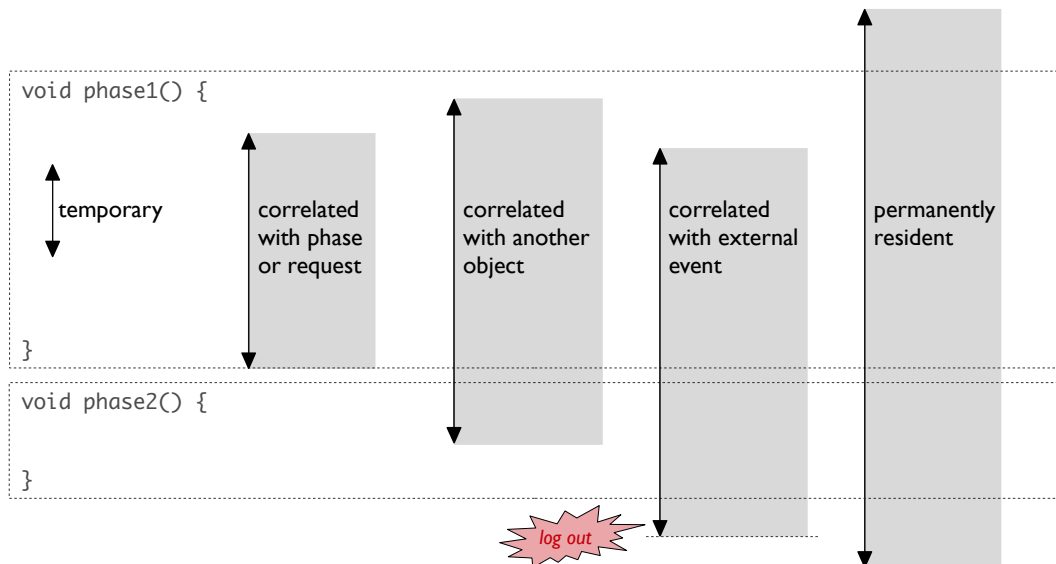
Figure 11.1. Illustration of some common lifetime requirements.

the application runs forever is an especially complex issue. This is true for more than server applications. Desktop applications such as the Eclipse integrated development environment share many of the same challenges. Improperly managing the lifetimes of objects, for short-running applications, often does not result in critical failure. Indeed, a short-running application often finishes its run before one would even notice a problem with memory consumption. Plus, you 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 possibility for mistakes to pile up, and for misconfigured or poorly implemented caches to impede performance means that special care must be taken to manage object lifetimes correctly when implementing a server application.

The heap consumption of a long-running application fluctuates over time, depending on the application phase. A timeline view of expected memory consumption, such as shown in Figure 11.2, helps to visualize these phase fluctuations. It shows memory during the lulls and peaks of activity: when the server starts up, when requests are processed, and when sessions time out. We will use timeline views 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 JRE 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 in Figure 11.2 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,
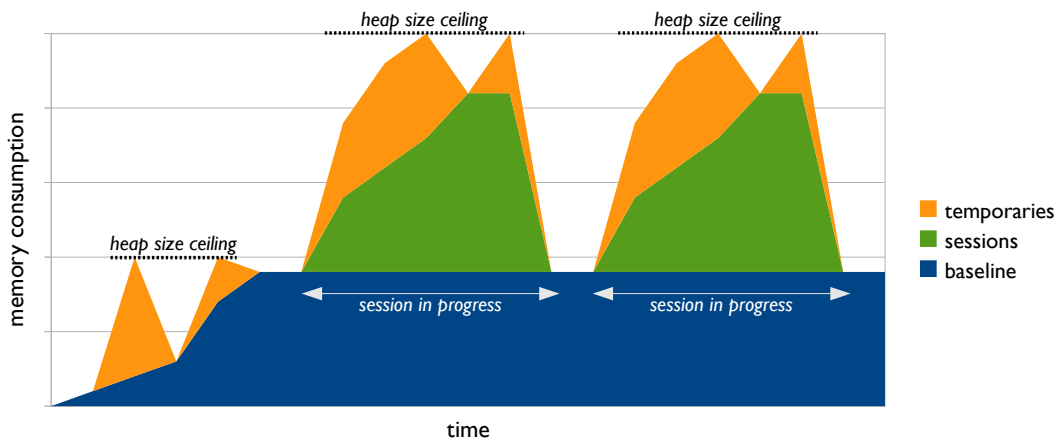
Figure 11.2. Memory consumption, over time, typical of a web application server.

once the JRE runs a garbage collection, the temporaries no longer in use will no longer be in the heap.

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 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. Shortly after they are constructed, the string builder, string, and character arrays, are no longer used:

```
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 duplication 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 and freeing up the memory for an object is free, there remains the work done in the constructor:

```java
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 every 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 DateFormat.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

Often 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.

Objects that Live and Die Together   If you need to augment the state stored in instances of a class that you are responsible for, you could 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. However, 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 short-running methods, you don't have to think about it, since local stack-allocated temporaries go away automatically when the method exits. 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 a 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 responsibility 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. **??** 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 figures, 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.

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. Like objects associated with requests, objects associated with sessions may accidentally live forever because of a bug, causing a memory leak. possible

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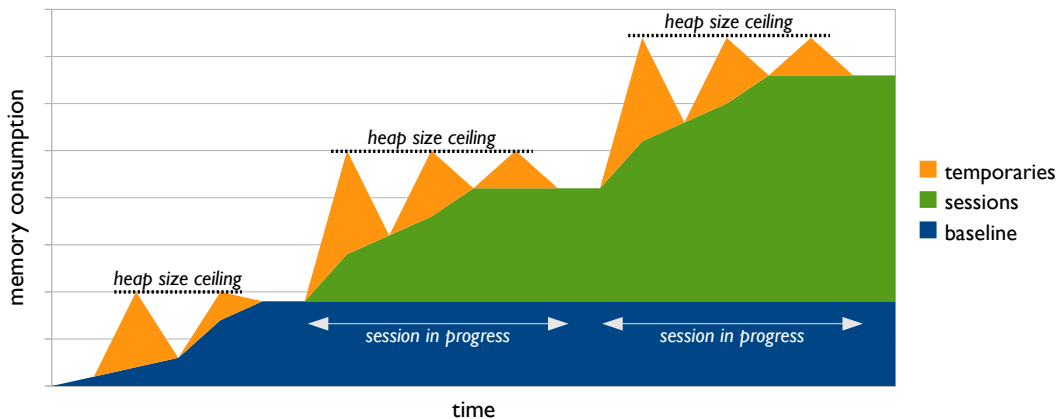
Figure 11.3. If session state is not cleaned up properly, a memory leak is the result. This means more frequent garbage collections, and ever increasing heap size.

11.4   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 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 data parsing 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 15 discusses strategies for coping with this situation.

11.5   Time-space Tradeoffs
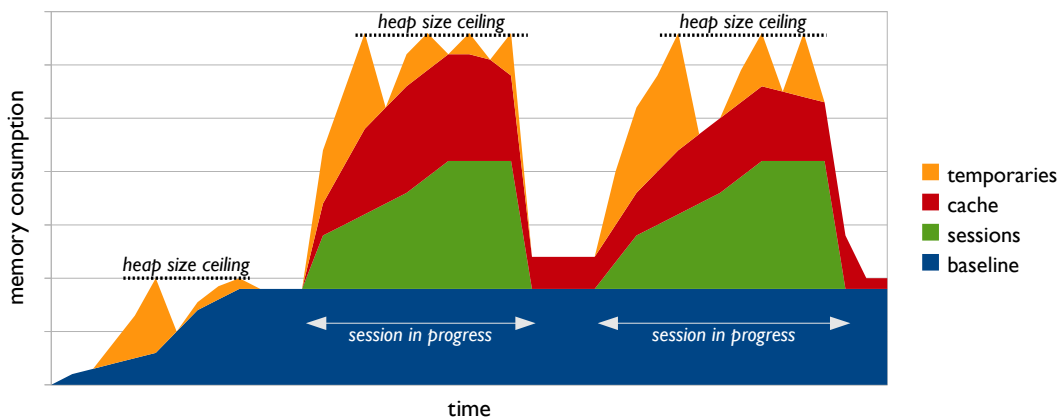
Figure 11.4. When a cache is in use, there is less headroom for temporary object allocation, often resulting in more frequent garbage collections.

It is sometimes beneficial to extend, or shorten, the lifetime of an object, depending on whether you need to optimize for time or space. 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. **??** discusses implementation strategies for these situations.

Our example server application caches data from an expensive third-party data source. Caching external data in the Java heap complicates programming and management tasks. The cache must be configured properly so that its contents live long enough to 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*. **??** 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.

11.6   Summary

This chapter sets the stage for Part 2 of this book by defining five types of object lifetime requirements that programmers need to be aware of, especially for long-lived applications like servers. If you make the effort to identify the lifetime requirements for the objects in your application, you can avoid common mistakes that result in memory leaks. Here is a high level summarization of the different kinds of lifetime requirments.

- At one extreme are short-lived temporary objects that are managed by the JRE garbage collector. You don't have to worry too much about temporaries, except that creating too many of them can cause a performance problem.

- Some objects have a lifetime requirement that is correlated with something else, such as the lifetime of another object, an execution phase, or an external event. Often, these correlated objects must be explicitly freed based on another object going away, a phase ending, or an event notification. Managing correlated lifetimes can be tricky and error-prone. There is, or course, no way to explicitly free objects in Java, so **??** describes a variety of techniques for implicitly freeing correlated objects.

- At the other extreme, many objects are needed throughout the execution, and must be permanently resident in the heap.

The moral is that even though Java has a garbage collector, you still have to understand and implement object lifetime requirements.

Chapter 12

LIFETIME MANAGEMENT

[NOTE: Section 12.1 has way too much detail. I think the whole section could be deleted, but if you want to keep some of the points, should be much shorter, maybe one page. It's certainly out of place after Chapter 11. The remaining sections are more appropriate, especially Basic Ways of Keeping an Object Alive – I like that — Edith]

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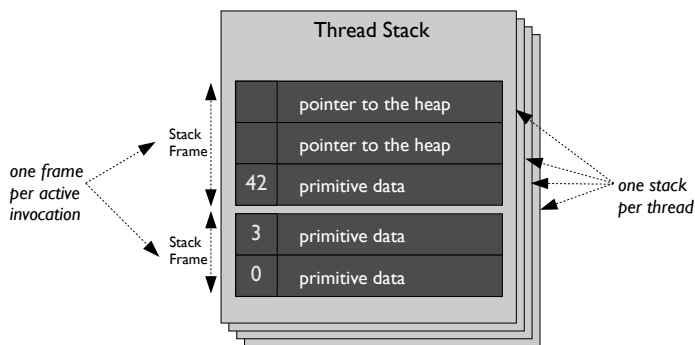
Figure 12.1. The compiler takes care managing the stack, by pushing and popping the storage (called *stack frames*) that hold your local variables and method parameters.

pects of the managed runtime will consume resources even within your process. This practice of continual experimentation is important, given the disparity between the 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. **??** goes into more detail on the tuning parameters at your disposal.

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`.

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

---

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

thread, these stack limits are exceeded, you may see a `java.lang.StackOverflowError` 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 Oracle 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 `struct`s. 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 `OutOf-MemoryError`. 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?

The Permspace Heap   The JRE stores information about classes, including executable code and string constants, 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 Oracle JRE is in the former camp, while the IBM and JRockit JREs are in the latter camp. The Oracle JREs call this heap the *permspace* heap. This name came about because the heap contains objects and other data that are, at least for the most part, immortal. On Oracle JREs, you can size the permspace heap via the command-line parameter `-XX:MaxPermSize`.

---

[2]If you are curious, the best you can do is to analyze performance both with and without the underlying anlaysis. On Oracle 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.

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 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

---

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

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.

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.

When to Collect: GC Safe Points   The garbage collector typically only runs when it has decided that heap memory has become highly constrained. When this happens, it quite often needs to stop the application threads, so that it can look for objects to reclaim.[4] It requests that the threads stop, and then waits until each thread has reached a *safe point* 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. At this juncture, it begins to look for objects to reclaim.

What to Collect: 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

---

[4]A *concurrent* collector only performs part of the garbage collection work concurrently with the application threads. Most concurrent collectors still need to pause the threads in order to perform reclamation.

(a) A live data structure.   (b) Then, a reference is clipped.
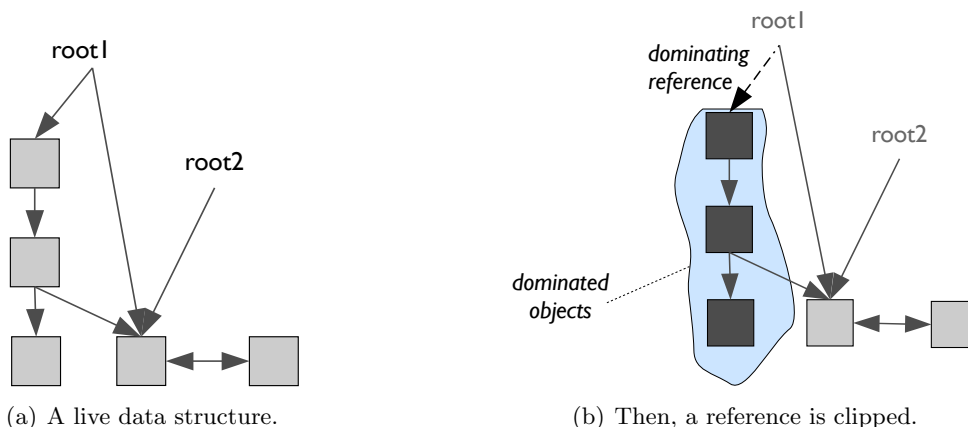
Figure 12.3. The garbage collector uses the references between objects to determine what objects are collectable. The data structure on the left is filled entirely with live objects. The one on the right, after a link is clipped, now contains some collectable objects; all objects reachable only by traversing a *dominating reference* will be collectable, as well.

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. Java applications are frequently plagued by memory leaks, and cases where memory allocations *drag* out their lifetime, beyond when
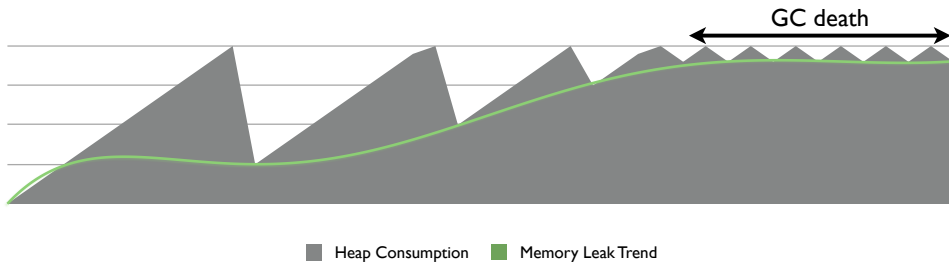
Figure 12.4. When your application suffers from a memory leak, you will likely observe a trend of heap consumption that looks something like this. As memory grows more constrained, garbage collections are run with increasing frequency. Eventually, either the application will fail, or enter a period of super-frequent collections. In this period of "GC death", your application neither fails, nor makes any forward progress.

your code needs them. Avoiding these problems, especially in more complex cases such as those involving caching or the use of native resources, is a perennial challenge of writing in Java. For these cases, you need to be aware of the how the managed runtime treats an object in various stages of its life.

Memory Leaks, and Running Out of Java Heap   If your application seems to have periods where it runs very slowly, and then either magically recovers or dies, then it may be suffering from a memory leak. Figure 12.4 shows a typical trend of heap consumption that you may observe, if it indeed does have a leak. This chart represents how much Java heap memory is free *after* each garbage collection. The characteristic aspect of a leak is an inexorable decrease in the amount of free memory after each garbage collection. Sometimes, the amount of free memory after a garbage collection sawtooth up and down, for example because caches fill up and are cleaned out as memory grows tight, or due to the natural ebb and flow of load on your application over time. Nevertheless, the upwards climb is stll there. As heap memory becomes less and less available, garbage collections become increasingly frequent.

    If your application exhausts the Java heap, you will observe an `OutOfMemoryError` exception thrown. On Oracle 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.
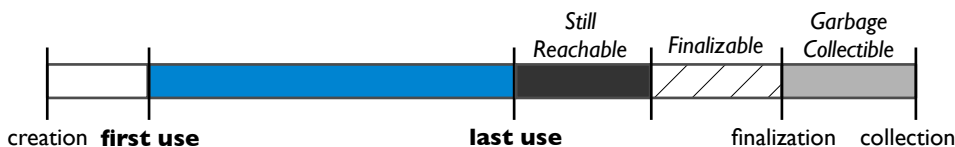
Figure 12.5. Timeline of the life of a typical object.

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 nicknamed *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.

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 Oracle JDK build of the JRE, then you can use the `jstat` tool.[5] 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 12.5 illustrates the lifecycle of a typical object in a well behaved application.

---

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

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);
  ...
}
```

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 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

> 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

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 [4], 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;
    ...
  }
}
```

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 state-

Figure 12.6.   When `f` returns, `obj` ownership automatically ends, but `static_obj` ownership persists.

ment 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.

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`

```
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");
}
```

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.

(a) Diamond sharing.   (b) Root sharing.
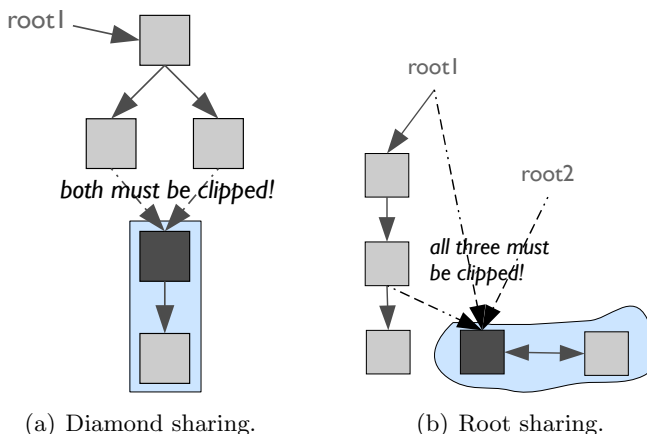
Figure 12.7. When an object is shared, such as the shaded ones shown here, care must be taken to clip all edges from emanating from outside of the region you wish to reclaim.

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 12.7 illustrates a situation where three references must be clipped before an object, the darkly shaded one, becomes a candidate for garbage collection. There are two other important things to note in this example. First, just 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

[(GSS) These next 3 section headers are for the table of contents to show what we are likely to have in a final version. The placement of these headers isn't right relative to the rest of the section. These are just section names that I'd like to stand out in the TOC for the reviewers to see. They're common cases, and we reference some of them in the first half of the book.]

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

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

12.5.1   Example. Correlated Lifetime: Sharing Pools

12.5.2   Example. Correlated Lifetime: Annotations

12.5.3   Example. Time-space Tradeoffs: Caches

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.

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*.

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 make a partial effort to clip soft references in a least-recently used (LRU) 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. In reality, no contemporary JRE uses a true LRU policy for clipping soft references. For example, with Oracle JREs, it is possible that all soft references that haven't been used in 6 minutes will be automatically cleared, even though the heap has 500MB of space free, out of a 600MB heap. **??** has more details on this subject. Soft references are still useful, despite this limitation, but you will need to design around this limitation.

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. **??** 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 directly referenced ownly 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.

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

Figure 12.8. 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 12.8.

Reference Queues   Normally, when you create a soft reference, and the JRE decides to clip it, then the associated `Reference` object becomes immediately garbage col-

lectable. 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 sounds tedious and error prone, it is! For some use cases, the standard library provides mechanisms that hide some of these details. For example, `WeakHashMap` does a good job of completely hiding reference queues and weak references from you.

If you do find a need to use reference queues directly, there are a few points of caution you should incorporate into your design. First, if you don't poll your reference queue at a rate that at least equals the worst case rate at which they will be queued, then the size of your reference queue will grow unboundedly. The worst case rate of objects queueing up is the rate at which they are created. Therefore, you should make sure to poll the queue in all code contexts that create any of the special kinds of references.

Reference Queue Polling Rule

A reference queue must be polled at least as often as `Reference` objects are created, that are destined for that queue. Otherwise the queue will leak memory, growing unboundedly.

Second, since you need to poll wherever a queueable item is created, you may run into a severe degree of lock contention. The `poll` method of `ReferenceQueue` does synchronize access to the queue, so you don't need to worry about race conditions, but this can result in a major impediment to multithreaded scalability. **??** gives an example of how to solve this problem, when implementing a concurrent cache.

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 cleanup action with the reclamation of an object. For example, 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 many resources, possibly scattered across several machines. There are operating system resources, for managing connections, 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.

Java provides *finalization* and *phantom references* as two ways around this problem. With the mechanisms, you can install cleanup hooks. Finalization lets you associate a cleanup hook with a class. Phantom references let you associate a cleanup hook at a finer granularity, on an object-by-object basis. Invocation of a finalization cleanup hooks is managed by the JRE, so your cleanup code operates at the whim of the JREs scheduling decisions. If it does not run the finalization cleanup hooks in a timely enough manner, there's pretty much nothing you can do. In contrast, you are in charge of invoking the phantom reference-based cleanup hooks, primarily via the reference queue mechanism.

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. The threads need to sleep or spin-lock before they can enter the contended critical section.[6]

Java provides an alternative way to implement a permanently resident pattern called *thread-local storage*. Thread-local storage provides a easy way to store cloned 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 explicitly `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: `ThreadLocal<DateFormat>`.

---

[6]Spin-locking, optimistically assuming that the lock will be available in the near future, retries the lock acquisition a few times in a tight loop. This can help, in the case of limited lock contention, but can result in wasted CPU cycles in the case of severe contention.

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.

Part III

Scalability

Chapter 13

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?   **??** 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 15 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 14

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.

14.1 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 14.1 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
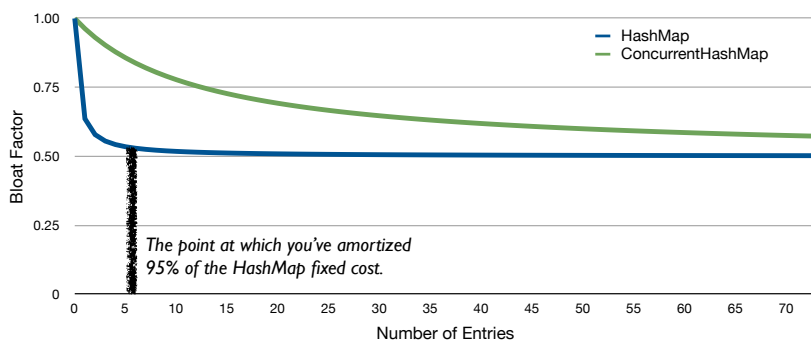
Figure 14.1. An example of the asymptotic behavior of bloat factor.

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:

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

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.
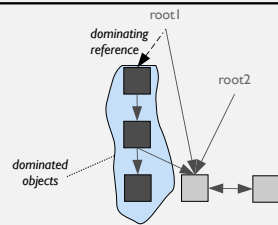
Every class has a *unit cost*, the cost for every additional instance. You can determine the unit cost of a class by counting up the size of its fields, taking into account JRE-imposed costs. Similarly, every interconnected group of instances has a unit cost, the sum of the sizes of the classes involved in that group. Section 3.1 introduced this style of accounting. Figure 14.2 gives an example EC diagram of a HashMap that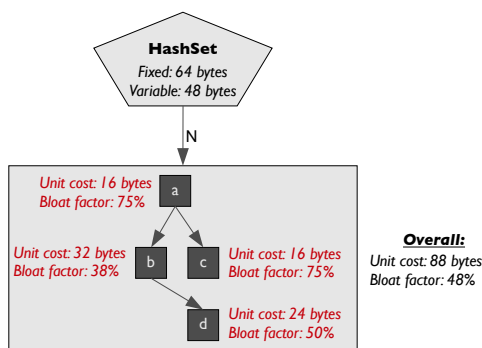 contains an interconnected group of four objects. The unit cost of each entry in this structure is given by the sum of the sizes of those classes: 88 bytes, in this case.

Figure 14.2. 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 14.2 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.

14.2   Quickly Estimating the Scalability 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 14.3(a) and Figure 14.3(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 14.2.

14.3   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 14.4. 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.
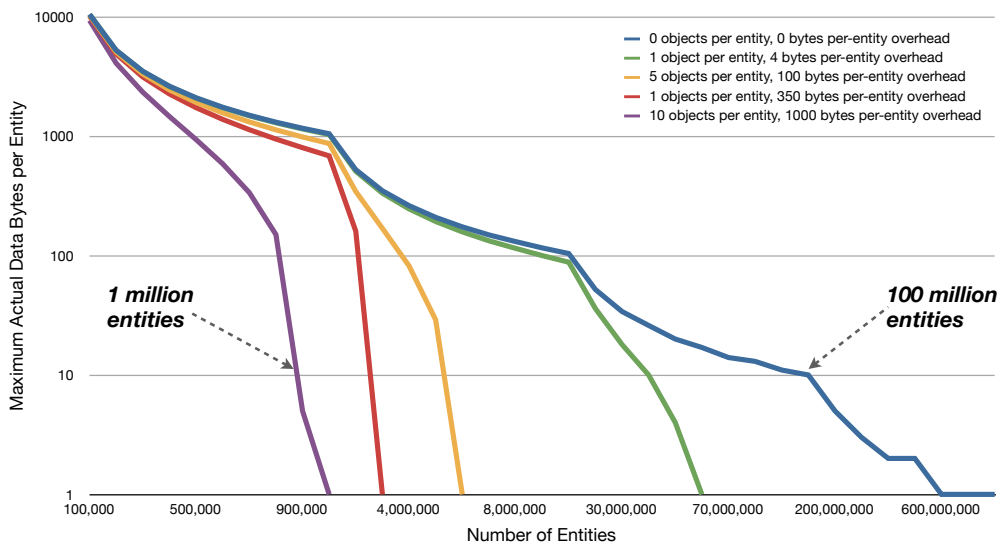
(a) The amount of actual data you can store in your entities depends on the degree of delegation in your entities, and the per-entry overhead of the collection in which these entities reside.
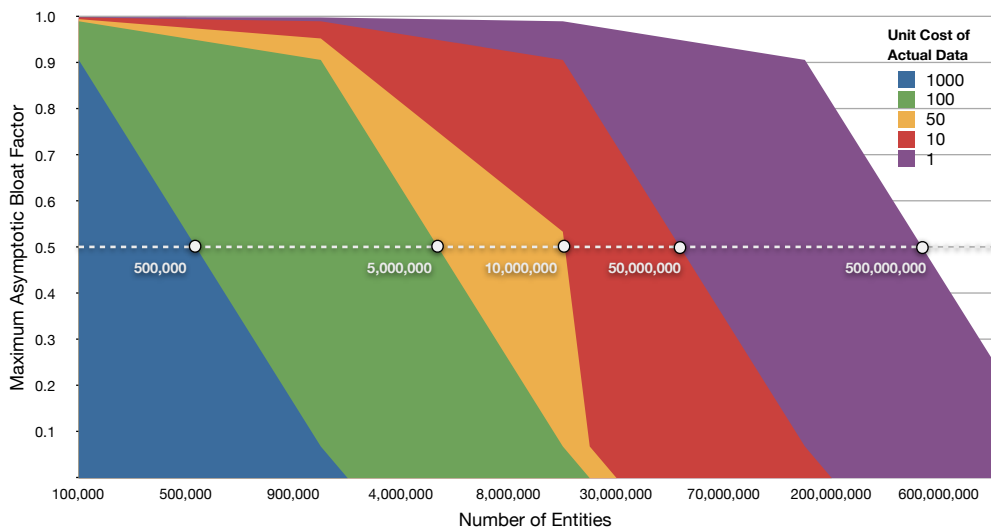
(b) The area under each curve shows the bloat factor you can afford, for various amounts of actual data that you need to store.

Figure 14.3. 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 14.3(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.

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

```
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();
}
```
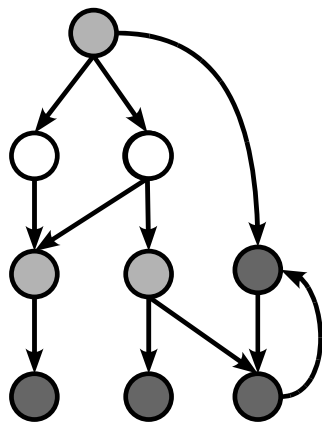
(b) If you do!

Figure 14.5. 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.

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?

Figure 14.4. Example graph.

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 14.5 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.

14.3.1   The Straightforward Implementation, and Some Tweaks

A reasonable place to start is with a straightforward mapping of interfaces to concrete classes. Figure 14.6 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>();
}
```

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

(a) Node implementation.

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

Figure 14.6. 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[5] 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 (**??** 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[6]. Under the covers, though, the same mismatch exists, and now you are faced with the added complexities of interacting with these APIs.

improvement:

$$V = 4 \qquad\qquad \text{(variable cost of node array)}$$
$$U = 16 \qquad\qquad \text{(actual data per node)}$$
$$B = 1 - 16/380 \qquad \text{(bloat factor of node)}$$

maximum room for improvement $= \dfrac{V}{U} + \dfrac{1}{1-B} = 24x$

The maximum room for improvement is a factor of 24x, which is very high. Table 14.1 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).

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

Table 14.1. 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.

14.3.2 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 14.7a 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 14.7. 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 l;
    }
```

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.

14.3.3   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.

14.3.4   Supporting Growable Singleton Collections

Section 14.3.2 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.

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

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

Figure 14.8. The Singleton Collection pattern.

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 14.8: `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 14.3.1: 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]

14.4   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 14.5), 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.

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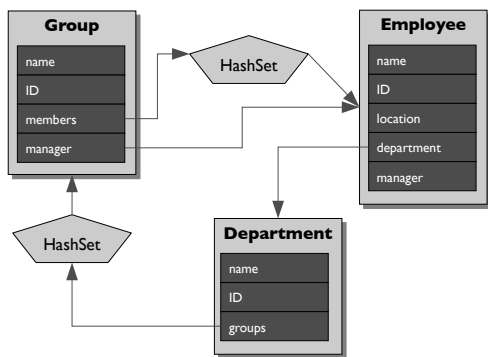
In this way, a conventional storage strategy, one which maps entities to objects and relations to collections, suffers from two problems that impact scaling up the size and complexity of a design.

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

Figure 15.2. 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 15.2 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 15.3. In C, `nodes` is an array of 20 integers, not pointers to 20 separate heap allocations.

15.1 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 15.4 gives an example which takes the graph of objects from Figure 15.2 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 15.4. Storing attributes in parallel arrays.

15.2   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 15.5(a) repeats the example graph from Figure 14.4, 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.
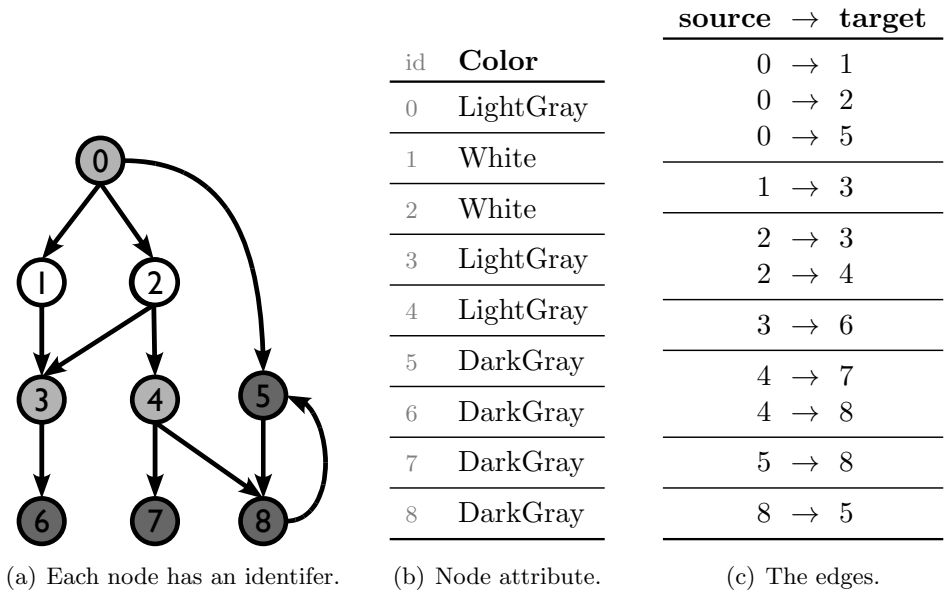
For the most part, these issues are much simpler when using column-oriented storage. Adding new attributes to nodes or edges is a trivial operation. Adding a new node attribute requires an extra array. You needn't reach a concensus among developers as to whether this is a good idea.

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.

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`

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

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

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

source	$\rightarrow$	target
0	$\rightarrow$	1
0	$\rightarrow$	2
0	$\rightarrow$	5
1	$\rightarrow$	3
2	$\rightarrow$	3
2	$\rightarrow$	4
3	$\rightarrow$	6
4	$\rightarrow$	7
4	$\rightarrow$	8
5	$\rightarrow$	8
8	$\rightarrow$	5

(a) Each node has an identifer.   (b) Node attribute.   (c) The edges.

Figure 15.5. 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.

15.3   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 15.5c 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 15.5c.

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 15.5c 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 15.6 shows an update to Figure 15.5, 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 15.5c by the **to** attribute. Figure 15.6a 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 15.6b, which shows the two

| node id | Color | Children | | Parents | |
		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 15.6. In order to support efficient traversal of the graph edges, you will need to index them. These tables update the example of Figure 15.5 to do so. For example, node 2 has two children and one parent. The children start at index 4 in Figure 15.6b, which shows the two children to be nodes 3 and 4.

children to be nodes 3 and 4.

15.4   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 15.7(a) and Figure 15.7(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 nonstring 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 15.7(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 15.7(d). This last step requires the most work

(a) Three normal `Strings`.

(b) Interning.

(c) Bulk sharing pool.

(d) Bulk sharing pool, no wrappers.
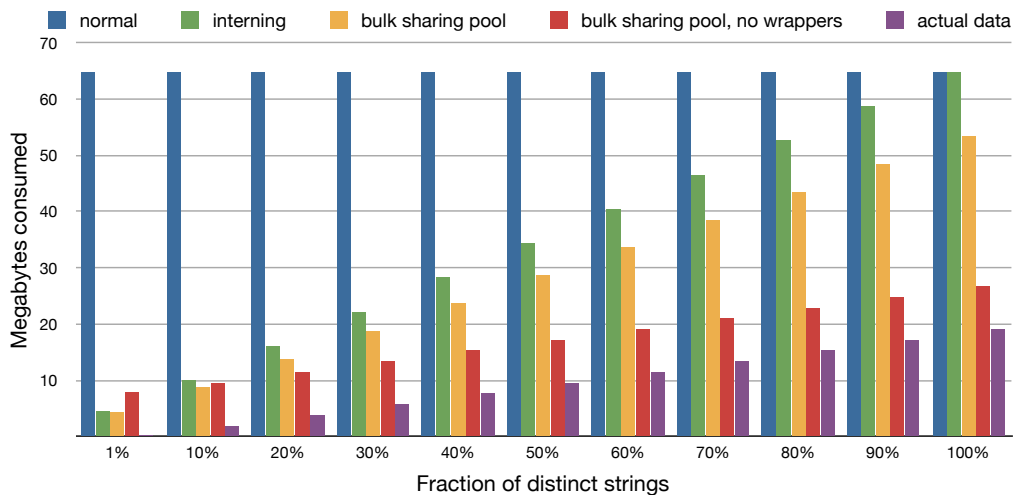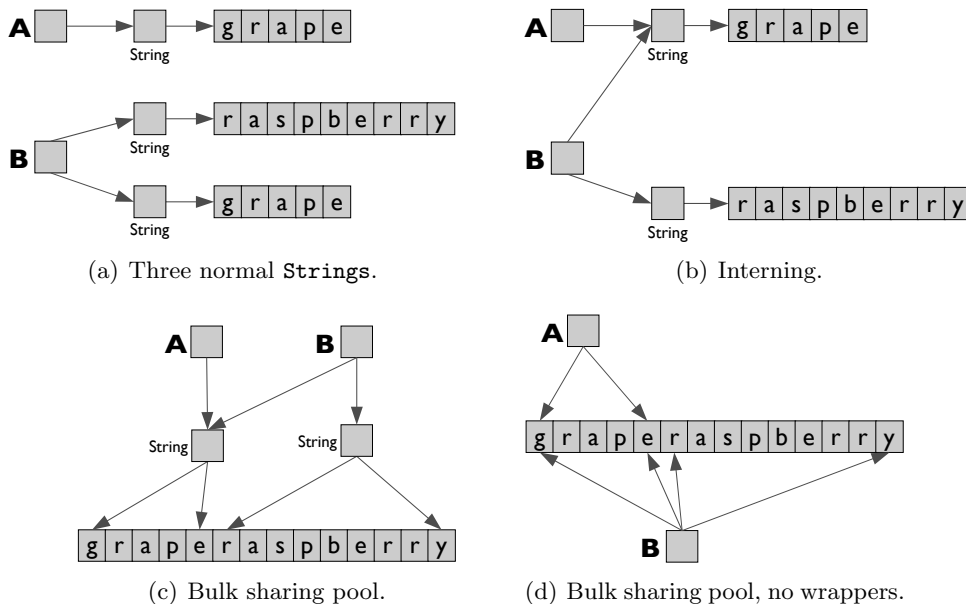
(e) The memory consumed by one million strings, each of length 10 bytes, for varying degrees of distinctness; e.g. 10% means that there are only 100,000 distinct strings.

Figure 15.7. 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 15.7(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 15.7(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.

15.5　　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.

15.6   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.

A COMPARISON OF SIZINGS ON JREs

[GSS: The following is the section structure we want to appear in the TOC. It doesn't match the text right now but will eventually.]

A.1   Sizing Criteria

A.1.1   Object-level costs

A.1.2   Field-level costs

A.1.3   Address space

A.2   Memory Costs of Commonly-used Classes

TODO: (Sizing Criteria) Let's make a clear separation of information readers need to estimate the size of their applications vs. the maximum address space issues. Otherwise the tables will be too large. That will also make the text more problem-oriented. We need three tables:

- Object-level overheads: (32 vs. 64 vs. compressed) x (Oracle vs. IBM), showing (object header size, array header size, object alignment size, bytes per reference)

- Field-level overheads, in two parts.

    - a. (type) x (arch) show field alignment
    - b. Show the field-packing rules for each each JRE (may be a bullet list rather than a table)

- Addressable bytes: (Oracle vs. IBM) x (OS), showing bytes. Some slots will be blank.

The majority of the body of this book focuses on the Oracle 32-bit JRE. This appendix provides supplementary material regarding the sizing criteria used by other JREs, and how various configuration choices affect these parameters.

Platform	Bytes per Reference	Bytes per Header	Alignment	Addressible Bytes
IBM zOS 31-bit	4	12	8	1.3GB
Windows 32-bit	4	8–12	8	1.8GB
AIX 32-bit	4	12	8	2.5-3.25GB
Solaris 32-bit	4	8	8	3.5GB
Linux 64-bit	8	24	16	128TB
AIX 64-bit	8	24	16	> 1PB
compressed refs	4	12	8	28-32GB
extended compressed refs	4	12	16	64GB

Table A.1. Sizing information for various platforms.

Table A.1 summarizes the important sizes that JREs use. These include the number of bytes each reference consumes, the number of bytes each object header consumes, the boundary (in bytes) to which each object allocation is aligned, and the maximum number of bytes a Java application can address. These values vary fairly widely, from one JRE to another.

The value that varies most is the maximum heap size a Java application can use. If you aren't running on a 64-bit JRE, you should exercise extreme caution in setting the heap size too high. For example, on a 32-bit Windows platform, if you request a heap size of 1.8GB, then your application will only have a few hundred megabytes of address space left over for native resources. All of your byte code, compiled code, and all of the JREs metadata about classes and the heap, need to fit into this small window. On AIX, if you request 3.25GB of Java heap, the story is similar.

[GSS: Compressed References discussion below. Will be moved elsewhere].

TODO: This duplicates discussion we have in the Objects and Delagation Chapter. - let's move info that we really need to that chapter, unless it's very detailed. Let's keep the Appendix mainly for reference details.

Provider	Command-line Argument
IBM	-Xcompressedrefs
Oracle	-XX:+UseCompressedOops
JRockit	-XXcompressedrefs

Table A.2. Options for specifying that you wish the JRE to use compressed references. This is only relevant for 64-bit JREs.

One potential downside to switching to a 64-bit JRE is an increase in memory consumption. Each reference will consume 8 bytes, and each object header will consume as much as 24 bytes. Luckily, most JREs support *compressed references*. When using compressed references, references and object headers consume as much as they would on a 32-bit platform, which is nice. There are two downsides. One is that the maximum heap size is far lower than it would be with full 64-bit pointers, though still higher than with a 32-bit JRE. For example, the Oracle and IBM Java 6 JREs support heap sizes of only up to 32GB; in some cases, the limit is 25 or 28GB. The other downside is a potential, though small, decrease in performance. THe hardware expects 64-bit pointers, but the heap only stores 32 bits. Therefore, each memory operation may require an extra shift operation. Some JREs are clever enough to avoid this expense, if the requested maximum heap size is less than 4GB.

For some JREs, if you are willing to pay a higher memory overhead, then you can support up to 64GB of memory, while still using only 32-bit references. When JREs operate in this mode, each object is aligned to a 16 byte boundary. This may result in an increase alignment overhead, compared to the normal 8-byte alignment. For example, say a `Double` boxed scalar has 8 bytes of data and a 12-byte header. This adds up to 20 bytes. With 8-byte alignment, you will pay 4 bytes for alignment cost. With 16-byte alignment, the alignment overhead is 12 bytes. On the other

hand, for a `String` or `Integer`, the alignment costs remain unchanged, whether the JRE aligns to 8- or 16-byte boundaries.

Many JREs will automatically enable this support for compressed references. You should consult the documentation of your specific JRE provider to know whether this is the case.[1]

[GSS: Identity Hashcode discussion below. This should be more of a footnote than a whole section.]

Depending on the JRE, the object header may actually be 4 bytes smaller than indicated in Table A.1. Some JREs leave room for the identity hashcode in the header of every object. The identity hashcode is usually the address of the object. If the garbage collector decides to move the object to another address at some point, then it has to store the original address somewhere. This is necessary so that the `identityHashcode` method always returns the same value, for the lifetme of a given object. The values in the table assume that the JRE preallocates space for this identity hashcode, and so every object pays the expense, even if the object's identity hashcode is never used. Some JREs are clever enough to store the identity hashcode only when it is needed: if it is used, and, even then, only if the object is moved from its original location. The only way you can know whether this is the case is to write a small program that allocates a large number of `Integer` objects. You know how much space you expect the object to consume, based on the 4 bytes of actual data, and the expected header size. Try it, and see how many you can fit, in a given size of heap.

---

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

JRE OPTIONS RELATED TO MEMORY

Perm space (Oracle only) . 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.

Sharing pool for `Integer`s (Oracle and IBM JREs) . You can change the upper range of the built-in `Integer` sharing pool by setting `-XX:AutoBoxCacheMax=size`. You can only extend the range upward; the upper limit is always a minimum of 127.

BIBLIOGRAPHY

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, 3rd Edition*. Addison-Wesley, 2005.

[2] K. Venstermans, L. Eeckhout, and K. DeBosschere, "64-bit versus 32-bit virtual machines for java," *Software-Practice and Experience*, vol. 36, no. 1, 2006.

[3] J. Bloch, *Effective Java, Second Edition*. Addison-Wesley, 2008.

[4] OSGi Alliance, "Osgi service platform release 4." [Online]. Available: http://www.osgi.org/Main/HomePage. [Accessed: Jun. 17, 2009], 2007.

[5] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, p. 5, 2004.

[6] C. Bauer and G. King, *Java Persistence with Hibernate*. Greenwich, CT, USA: Manning Publications Co., 2006.

[7] Apache, "The apache software foundation. apache daytrader benchmark sample." [Onlne]. Available `https://cwiki.apache.org/GMOxDOC20/daytrader.html`.

[8] Google, "Protocol buffers: Google's data interchange format." [Online]. Available: `http://code.google.com/apis/protocolbuffers`.

[9] Google, "Guava: Google core libraries for java." [Online]. Available: `http://code.google.com/p/guava-libraries/`.

[10] D. F. Bacon, S. J. Fink, and D. Grove, "Space- and time-efficient implementation of the java object model," in *The European Conference on Object-Oriented Programming*, 2002.

[11] D. J. Pearce and J. Noble, "Relationship aspects," in *Aspect-oriented software development*, 2006.

[12] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Object-oriented Programming, Systems, Languages, and Applications*, 2006.

[13] I. Dillig, T. Dillig, E. Yahav, and S. Chandra, "The CLOSER: Automating resource management in Java," in *International Symposium on Memory Management*, 2008.

[14] N. Mitchell and G. Sevitsky, "Building memory-efficient java applications," in *International Conference on Software Engineering*, 2008.

[15] A. Shankar, M. Arnold, and R. Bodik, "JOLT: Lightweight dynamic analysis and removal of object churn," in *Object-oriented Programming, Systems, Languages, and Applications*, 2008.

[16] M. Harren *et al.*, "XJ: Facilitating XML processing in Java," in *World Wide Web*, 2005.

[17] M. Braux and J. Noyé, "Towards partially evaluating reflection in Java," in *Workshop on Partial Evaluation and Semantics-based Program Manipulation*, 2000.

[18] B. Dufour, B. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *Foundations of Software Engineering*, 2008.

[19] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *Programming Language Design and Implementation*, 2005.

[20] G. Kiczales *et al.*, "Open implementation design guidelines," in *International Conference on Software Engineering*, 1997.

[21] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *International Conference on Software Engineering*, 2008.

[22] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *International Conference on Software Engineering*, 2007.

[23] Apache XML Project, "Xerces Java parser." `http://xerces.apache.org/xerces-j`, 1999.

[24] H. Maruyama, K. Tamura, N. Uramoto, M. Murata, A. Clark, Y. Nakamura, R. Neyama, K. Kosaka, and S. Hada, *XML and Java: Developing Web Applications.* Addison-Wesley, 2002.

[25] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delega-
     tion in a class-based object-oriented programming language," in *International
     Conference on Software Engineering*, 2008.

[26] B. Smaalders, "Performance anti-patterns," *ACM Queue*, vol. 4, no. 1, 2006.

[27] N. Mitchell and G. Sevitsky, "The causes of bloat, the limits of health," *Object-
     oriented Programming, Systems, Languages, and Applications*, 2007.

[28] N. Mitchell, G. Sevitsky, and H. Srinivasan, "Modeling runtime behavior in
     framework-based applications," *The European Conference on Object-Oriented
     Programming*, 2006.

[29] D. Spinellis, "Another level of indirection," in *Beautiful Code: Leading Program-
     mers Explain How They Think* (A. Oram and G. Wilson, eds.), ch. 17, O'Reilly
     and Associates, 2007.

[30] N. Mitchell, G. Sevitsky, and H. Srinivasan, "The diary of a datum: An approach
     to analyzing runtime complexity in framework-based applications," *Workshop
     on Library-Centric Software Design*, 2005.

[31] B. Goetz, "Java theory and practice: Urban performance legends, revisited,"
     IBM developerWorks, 2005.

[32] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Mod-
     eling Framework*. Addison-Wesley, 2003.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements
     of Reusable Software*. Addison-Wesley, 1995.

[34] Oracle Corporation, "Troubleshooting guide for java se 6 with hotspot vm,"
     2008.

SUBJECT INDEX