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 applications 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

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.

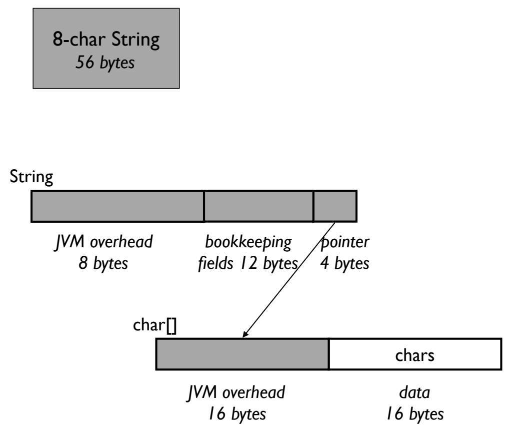


Figure 2.1. An eight-character string in Java 6.

- *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.

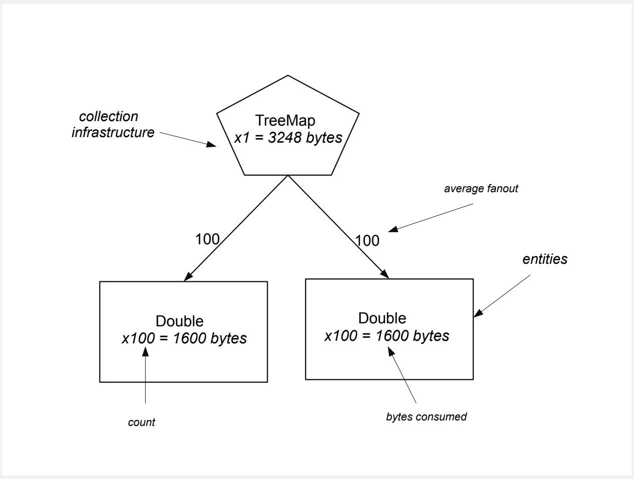
Strings, in particular short strings, are pervasive. It is important to be aware of the overhead of strings when incorporating them into your data design. Given the overhead of Java strings, the choices you make in Java need to be different than in a language like C, where the string overhead is minimal.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

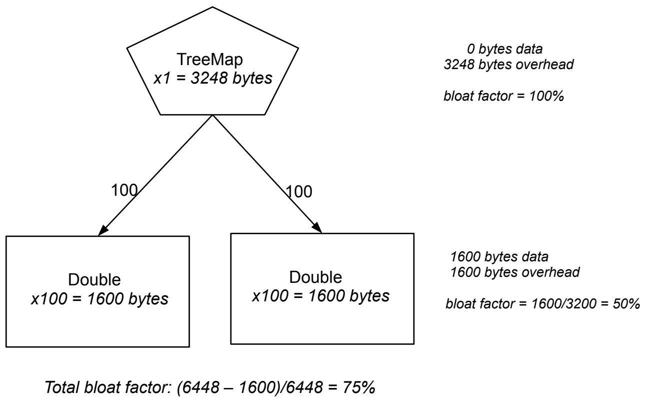


Figure 2.2. EC Diagram for 100 samples stored in a `TreeMap`

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) {
```

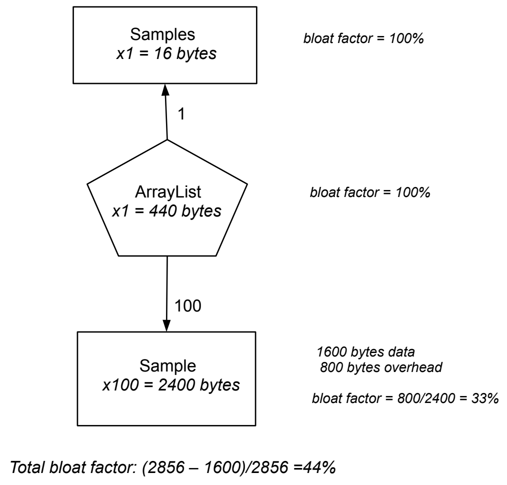


Figure 2.3. EC Diagram for 100 samples stored in an `ArrayList` of `Samples`

```
                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?

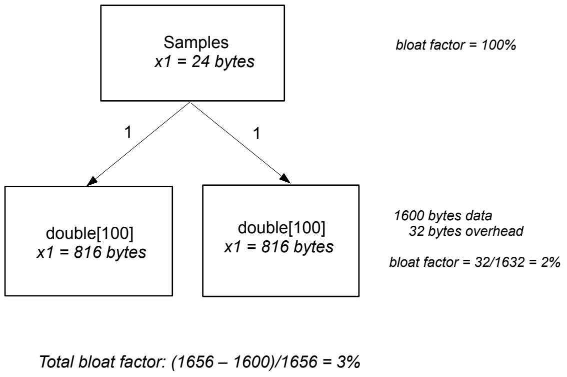


Figure 2.4. EC Diagram for 100 samples stored in two parallel arrays

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.

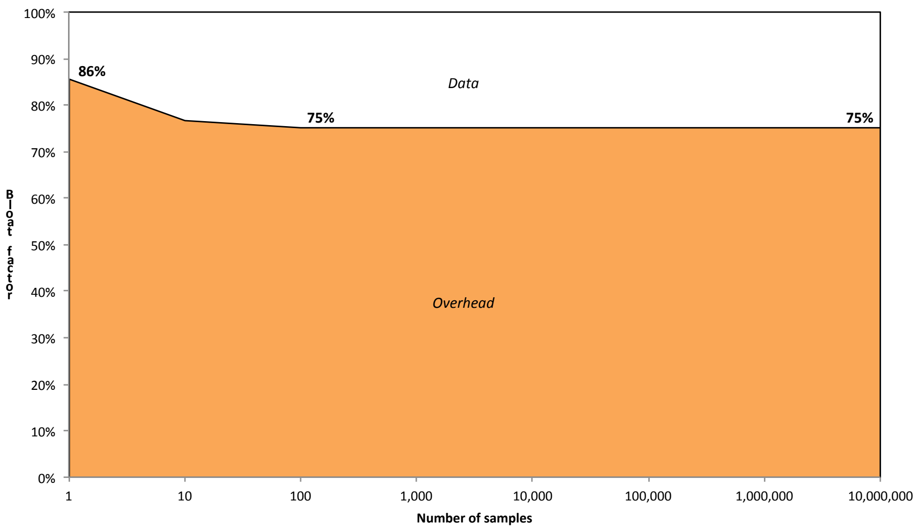


Figure 2.5. Health Measure for the `TreeMap` Design Shows Poor Scalability

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

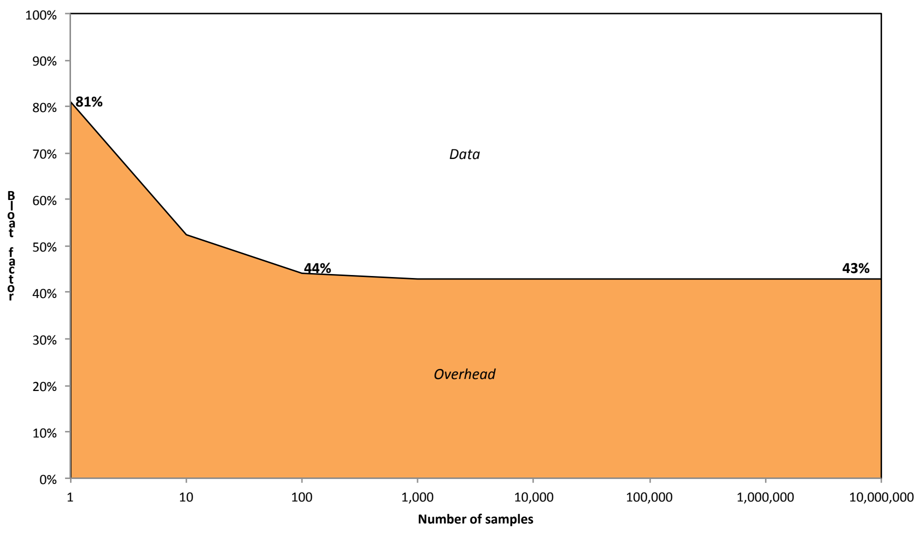


Figure 2.6. Health Measure for the `ArrayList` Design

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 *variable*. 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.

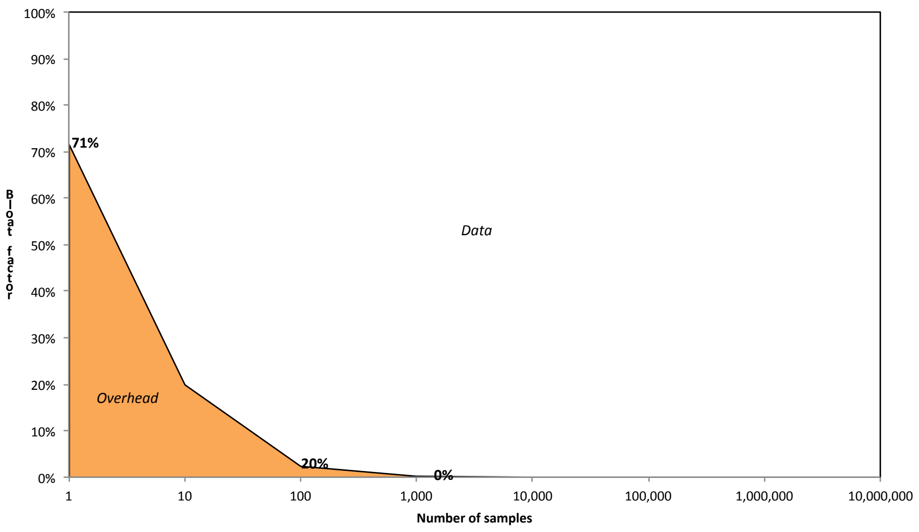


Figure 2.7. Health Measure for the Array-Based Design Shows Perfect Scalability

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.

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

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

[1] Unless otherwise noted, all of the numbers throughout the book are based on the Oracle Java 6 JRE for 32-bit architectures. Appendix A gives information needed to estimate sizes in various environments. N.B. The current draft is based on Oracle release sr31, and, where mentioned, IBM release sr10-fp1. The final book will be updated with later versions.

```
        int id;            // 4 bytes
        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.

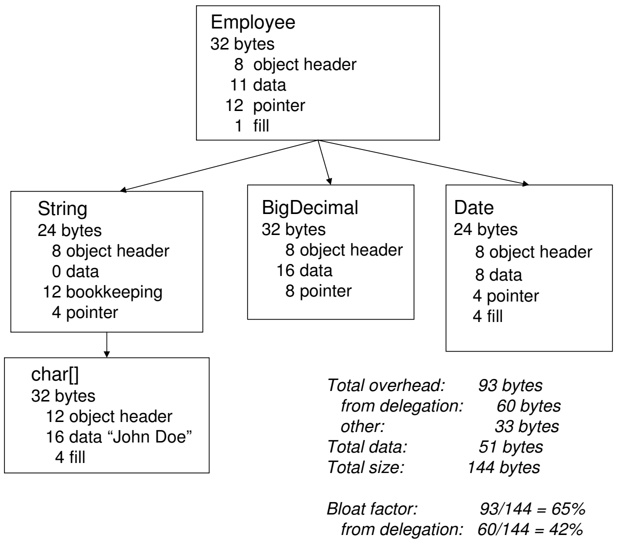


Figure 3.1. The memory layout for an employee, with some field data delegated to additional objects.

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:

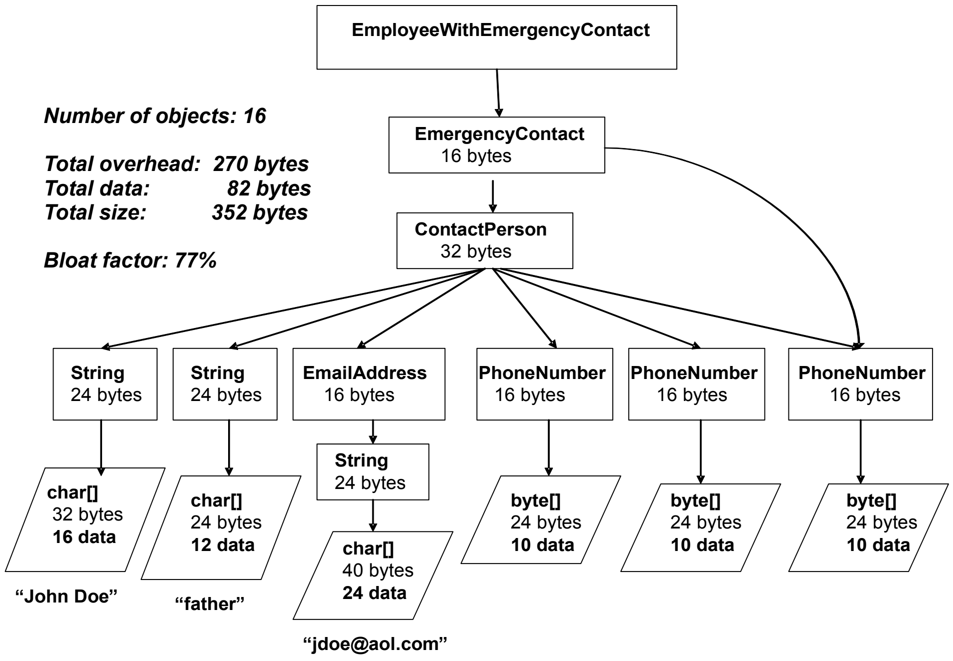


Figure 3.2. The memory layout for an employee with an emergency contact, using a fine-grained design.

```
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

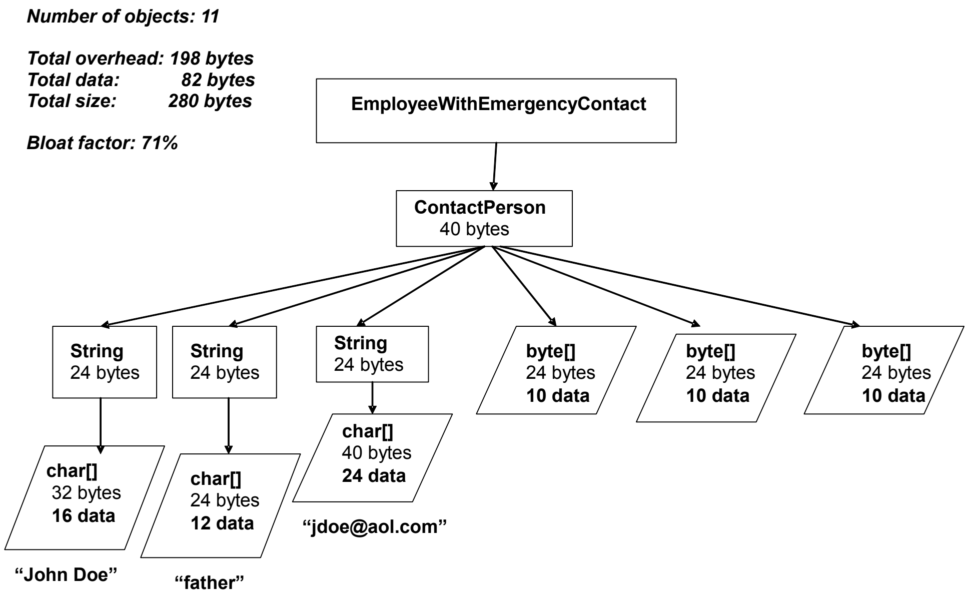


Figure 3.3. The memory layout for an emergency contact, with a more streamlined design.

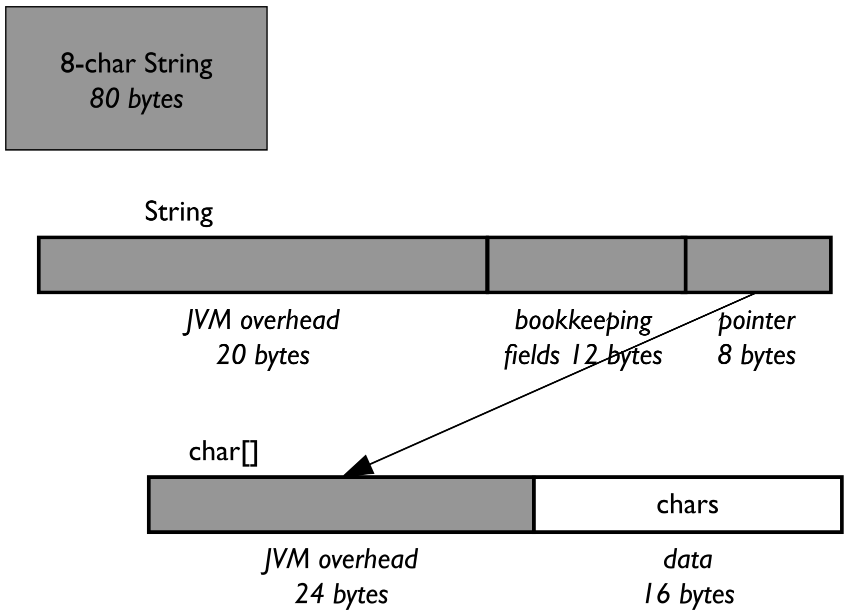


Figure 3.4. The memory layout for an 8-character string on a 64-bit JRE.

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.

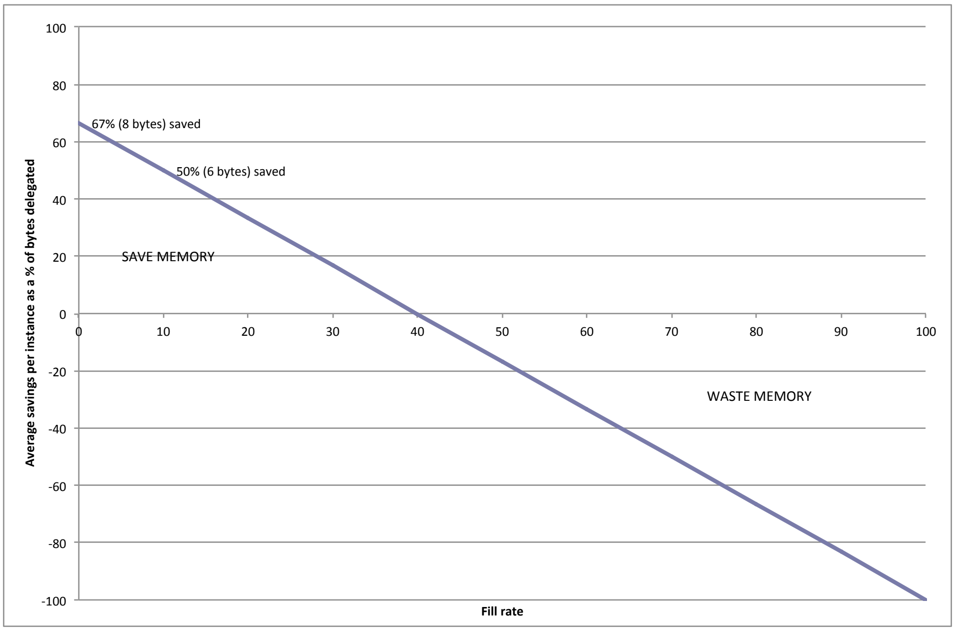


Figure 4.1. This plot shows how much memory is saved or wasted by delegating 12 bytes of memory to a side object. The x-axis is the fill rate. The y-axis is the average memory saved per instance, as a percentage of the bytes delegated.

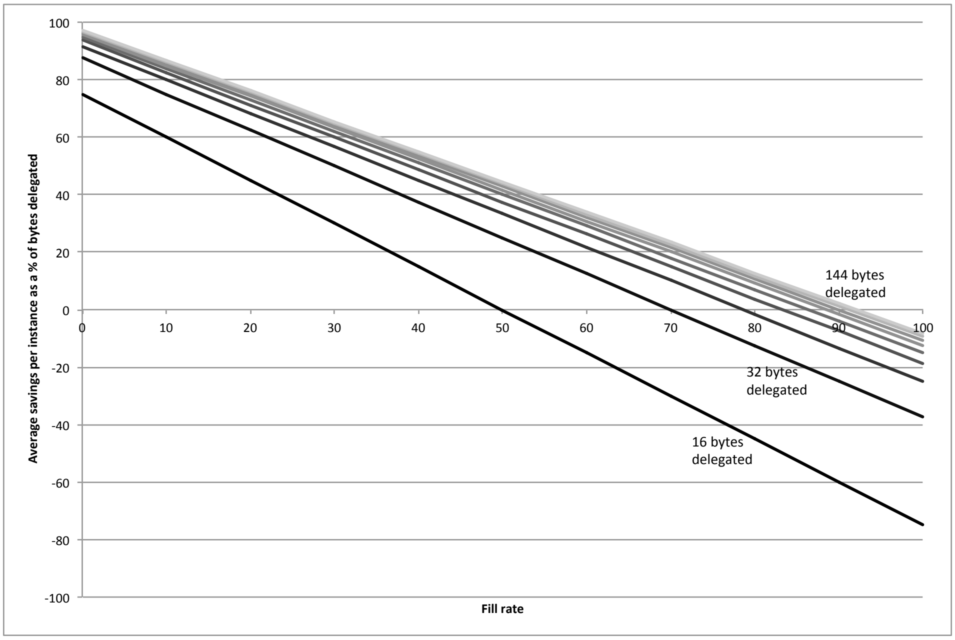


Figure 4.2. This plot shows how much memory is saved or wasted depending on how many bytes are delegated to a side object. The x-axis is the fill rate. The y-axis is the average memory saved per instance, as a percentage of the bytes delegated. Each line represents a different delegated-fields size, 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:

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

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

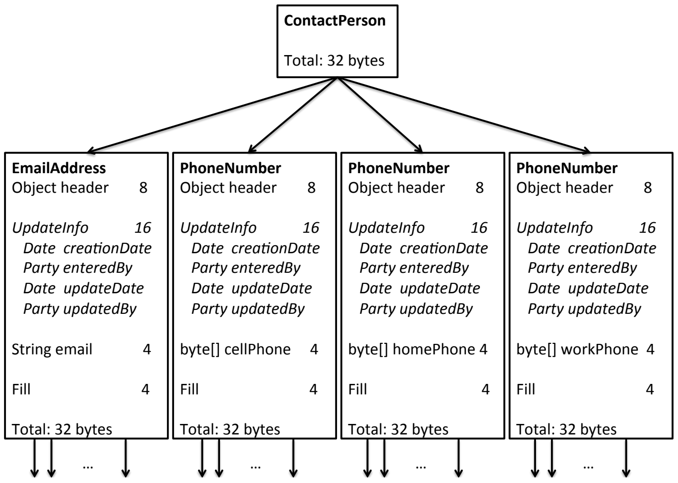


Figure 4.3. The cost of associating `UpdateInfo` with every `ContactMethod`.

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.

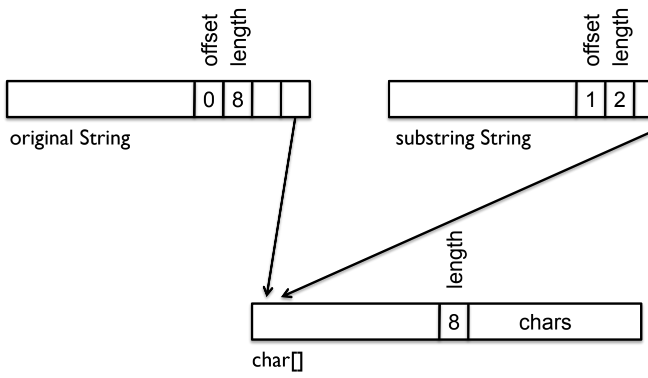


Figure 4.4. A string and a substring share the same character array. The length and offset fields are needed only in substrings. In all other strings, the offset is 0 and the length is redundant.

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:

```
    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`.

```
    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

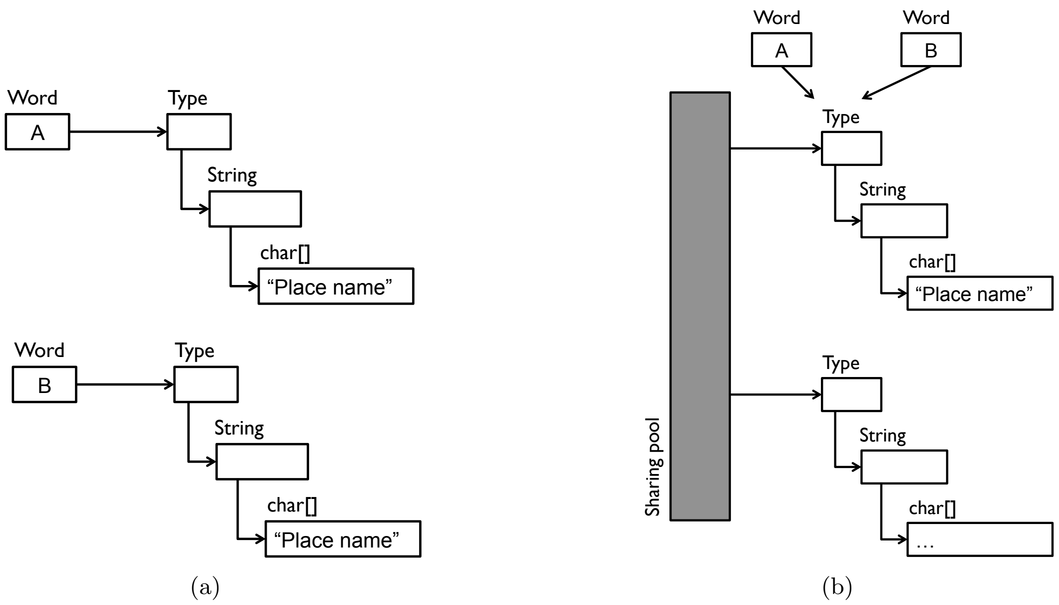


Figure 6.1. (a) Words A and B point to duplicated data. (b) Words A and B share the same data, stored in a sharing pool.

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.

```
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.

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

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 `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.

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

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 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.

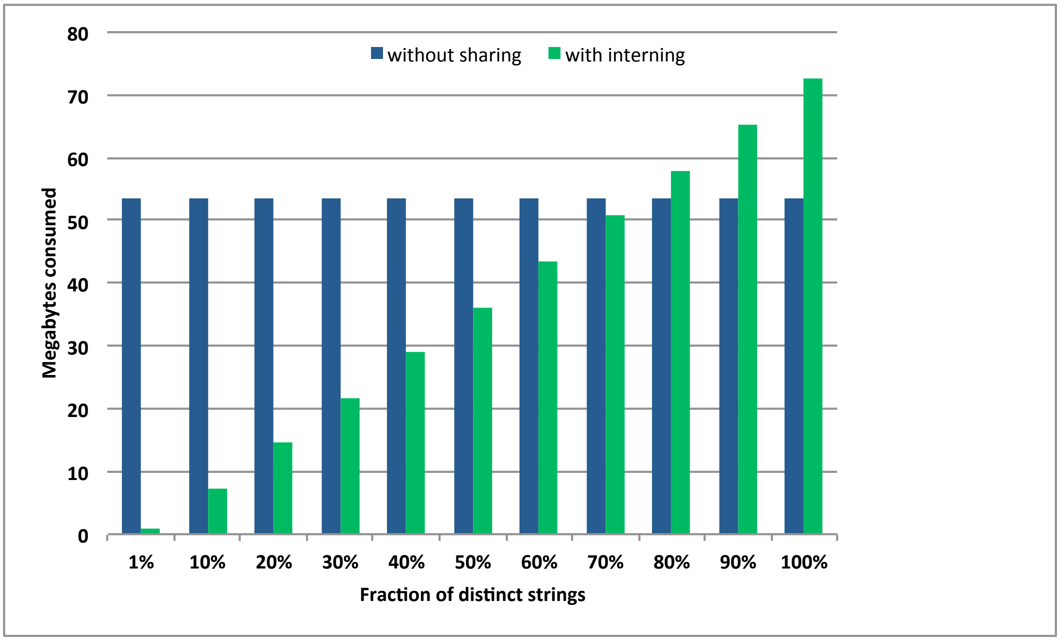


Figure 6.2. Comparing the memory consumed by one million ten-character `Strings`, stored individually vs. with interning. The chart shows how the savings (or waste) varies with the degree of distinctness of the data. 10% means there are only 100,000 distinct strings.

Estimating the Savings from Sharing

Let:

$$\begin{aligned} N &= \textit{the number of items in the data set} \\ S &= \textit{the average size in bytes of a data item} \\ D &= \textit{the ratio of distinct values to total number of items} \\ E &= \textit{the average per entry overhead of the sharing mechanism} \end{aligned}$$

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:

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

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 infrastructure, collections serve a necessary function, and paying for overhead can be worthwhile. That is, as long as you are not paying for features you don't need. In the above example, a 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.

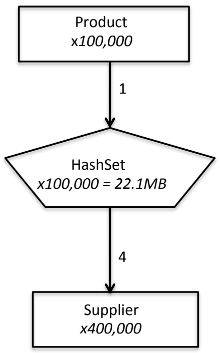


Figure 8.1. A relationship between products and alternate suppliers. Stored as one `HashSet` of alternate `Suppliers` per `Product`.

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

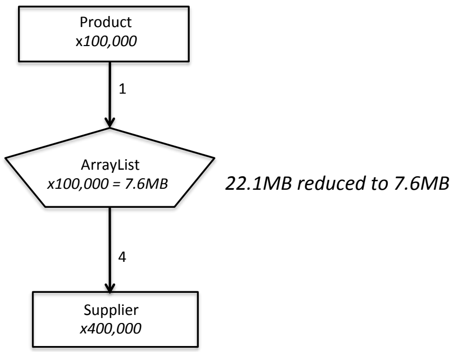


Figure 8.2. A relationship between 100,000 products and alternate suppliers, where the alternate `Suppliers` associated with each `Product` are stored in an `ArrayList`.

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.

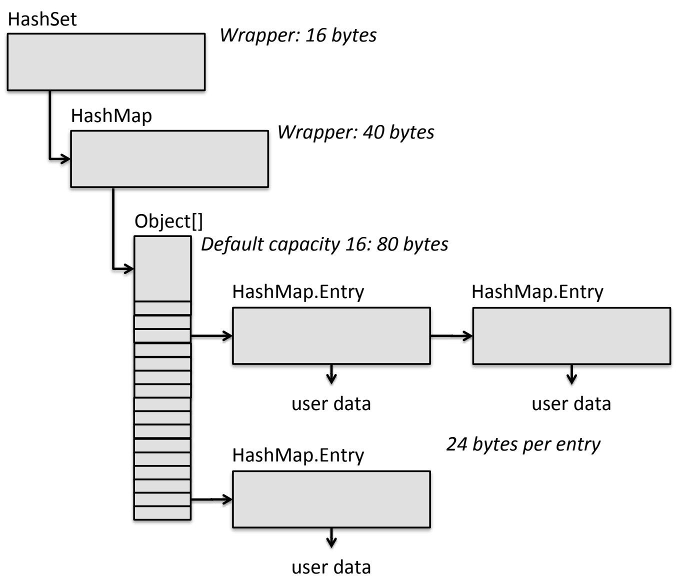


Figure 8.3. A look inside a `HashSet`. Shown with 3 entries.

`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.

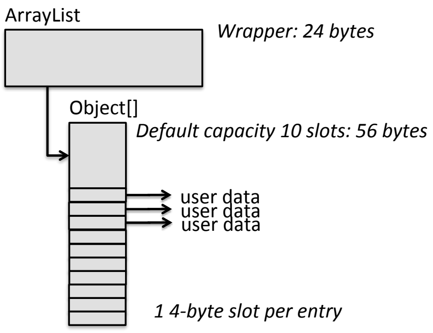


Figure 8.4. Inside an `ArrayList`. Shown with three entries and default capacity. `ArrayList` has a relatively low fixed overhead, and is scalable.

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.

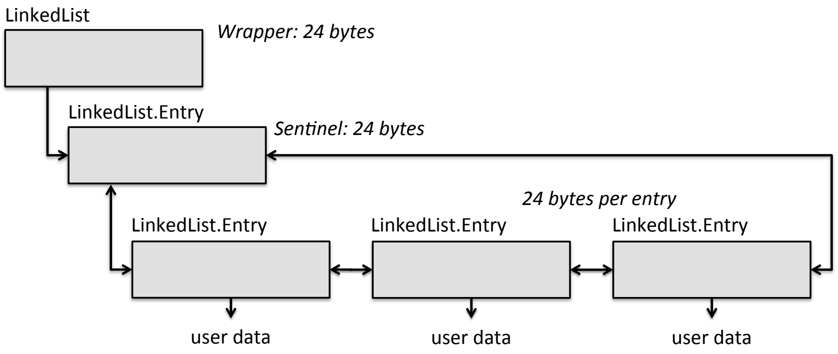


Figure 8.5. A look inside a `LinkedList`. Shown with 3 entries.

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,

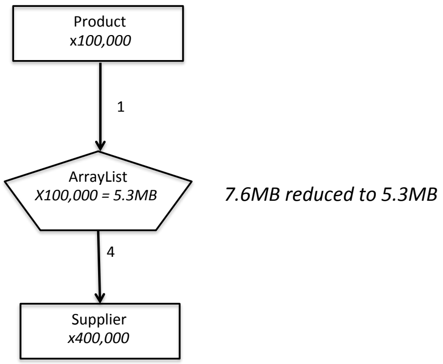


Figure 8.6. The relationship between `Products` and `Suppliers` after all of the `ArrayLists` have been trimmed by calling the `trimToSize` method.

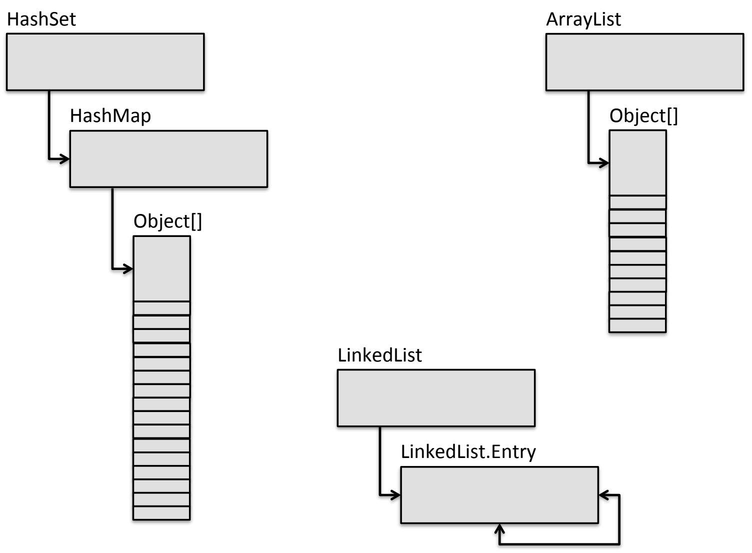


Figure 8.7. The internal structure of three empty collections. Each requires at least two objects, before any entries are added.

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.

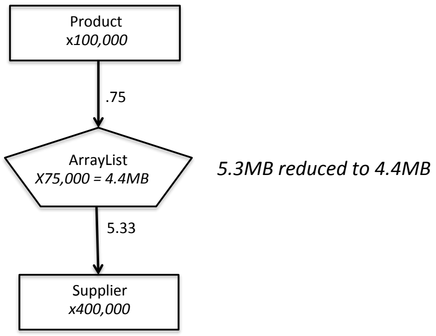


Figure 8.8. The relationship between `Products` and `Suppliers`, with no empty `ArrayLists`.

```
            ..
            // 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.

```
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.

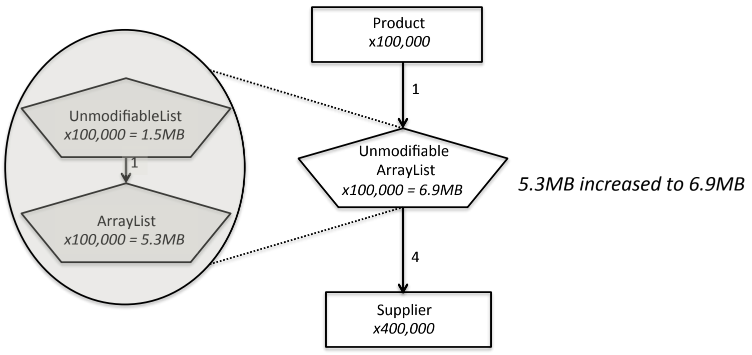


Figure 8.9. A relationship between 100,000 products and alternate suppliers, where the alternate `Suppliers` associated with each `Product` are stored in an `Unmodifiable ArrayList`. The inset shows how each collection now incurs the fixed cost of an additional view layer.

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.

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 delegation 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 Programmers 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 Modeling 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.