Building Memory Efficient Java Programs

Nick Mitchell Edith Schonberg Gary Sevitsky

Preface

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

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

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

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

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

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

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

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

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

The book is divided into four parts:

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

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

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

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

Contents

List of Figures

List of Tables

Part I

Memory Costs in Java

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, a compiler that runs as your program is running and responds to its operation, and a growing list of standard, open source, and proprietary libraries, such as collections, all written by experts. All you have to do is piece together the parts and let the Java runtime system do the rest for you.

The reality, of course, is very different. If you just assemble the parts, take the defaults, and follow all the otherwise good advice to make your program flexible and maintainable, you may soon find that your memory costs are *much* higher than imagined. What's more, keeping your data structures alive for exactly as long as you want can take much more effort than planned.

Languages like C give you a lot of control over storage, and so it is clear what the consequences of your choices are. In Java much is done for you, both by the runtime and by the many layers of frameworks, so it is easy to lose sight of the space your data designs require. In fact, it can be downright difficult to find out. This is especially true early in the development cycle, when it is easier to make design changes. Java also gives you fewer storage options than languages like C, and some of these options are surprisingly expensive. ¡ In fact, many of the basic building blocks in Java require more space than C++. ¿ So careful attention to the cost of design decisions is especially important in Java. This book's main purpose is to help you make informed design tradeoffs based on a solid knowledge of common data design options and their space costs. ¡ And the sooner the better, since once memory problems are discovered they can require extensive refactoring ¿.

There are some common beliefs and there are some common beliefs that can get in the way Before getting into too much technical detail, ... The next section dispels some common misconceptions about memory in Java. These misconceptions can get in the way of making The remaining sections in this chapter introduce some terminology and conventions used in the rest of the book.

Thanks to years of software engineering advances, there is now a large pro-

grammer toolkit for producing well-designed and maintainable programs, including a plethora of modeling techniques, design patterns, and coding advice. This is a success story for software engineering. Productivity has increased at a remarkable pace to meet the demands of an exploding software-driven world.

Along the way, programming has changed in some fundamental ways. Rather than building applications from scratch, programmers assemble applications from a collection of off-the-shelf, reusable frameworks and libraries. These reusable frameworks and libraries provide suitable abstractions, while under the covers they do the heavy lifting, performing a myriad of low-level tasks like connecting to a database, managing transactions, providing security, or building user interface widgets.

Layers of abstraction make it easy to construct applications rapidly, but also make it easy to introduce huge inefficiencies without the programmer being aware of it. Consuming too much memory is a particularly common kind of inefficiency. When a program stops with an out-of-memory exception, it's often hard to tell whether there's a memory leak or the program is just too big. In either case, the next more difficult question is how to fix it. This book is about to how to avoid memory problems in the first place. Before getting into specifics, we first look at the bigger picture, to motivate why you should care about how much memory your program is using.

1.1 The Big Pileup

A Java application is like an iceberg, and you, the programmer, typically see only the very tip. Coding at the tip, you select which libraries and frameworks to use for various purposes. Abstractions provided by these libraries and frameworks insulate you from the actual implemention. At the tip of the iceberg, you don't see (and you don't need to see) what's under water, and this what reusability is all about. Abstractions improve understandabiliy, quality, and pruductivity.

Unfortunately, abstractions also hide memory costs, and you are unaware whether a framework method call creates 5 objects or hundreds of objects. If a prorgram runs out of memory, then abstractions suddenly get in the way. To fix a memory problem, assuming you can't just increase the heap size, you need to get inside what's been carefully hidden from you, the rest of the iceberg. Typically, once you dive in, you will see that the frameworks you are using call other frameworks, which call other frameworks, and so on, eventually calling core libraries. The iceberg is really a big layering of abstractions, large enough to overwhelm anyone.

Often a memory problem doesn't have a single cause, which is another reason why it can be hard to fix. Like developers at the tip of the iceberg, framework developers also use abstractions without understanding their memory costs, so that each abstraction layer introduces inefficiencies. Added together, these result in a systemic bloat problem, which needs extensive refactoring to fix. Limiting systemic bloat requires paying attention to memory at every step of the development process and at every level of abstraction.

1.2 Some Common Misconceptions About Memory

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

Misconception. Memory is cheap, hardware is always improving, so things are fine. **Reality.** With gigabytes of memory available, you never need to worry about running out. However, resources are still finite, and it is surprising how easy it is to saturate them. Furthermore, while processor speeds have been doubling every two years following Moore's law, a physical limit has now been reached preventing further processor speedup. Instead, the number of processor cores on a chip is expected double every two years. However, memory bandwidth and cache sizes will not grow proportionally. Larger heaps combined with relatively less bandwidth is a recipe for a big performance hit going forward, and the rapidly growing number of small embedded processors will require more efficient use of memory.

Misconception. The JIT optimizer and garbage collector are so good that they will clean up all efficiencies. things are fine, everything is cheap; there was an article on developer works, called "Go ahead, make a mess", and this idea that objects are free, at least temporaries are, and everything will be taken care of for you. The JIT is going to clean things up, the garbage collector will elp, all of this great research on garbage collection and jit optimization, and you shouldn't have to worry about that; just code whatever the best design is from a maintainability standpoint, or whatever other standpoint, and the performance will magically be taken care of.

In fact, in the memory space, the JIT is doing, all of the commercial JITs that we know of are doing 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. We'll look into the detail of what that means. And similarly the garbage collector, yes it is cleaning up temporary objects. But even for temporaries there are other costs. General belief is that JRE, JIT, garbage collection take care of things.

The construction of the objects in the first place; garbage collectors are only dealing with shortlived objects, and footprint problems are a problem of long-lived objects, which garbage collectors are not addressing at all.

Myth: If you are using a library or framework, in particular if it is popular, it is natural to think that it was developers by experts, and therefore it is highly efficient. Therefore, you do not have to worry about its memory footprint. In reality, framework developers may not know the costs of the frameworks that they themselves are using. Additionally, they can't predict how their framework will be used, and you cannot assume that it will be optimized for your particular situation.

Myth 4: Performance or memory – tradeoff – would sacrifice good design. Goal of the book: teach how do good design, while taking memory costs into account.

Many developers know things are bad, but not how had, and we see this over and over again. In fact, I just came back, about a month ago, working with a group of IBM rational developers in Ottowa, a very strong group of people, very good engineers, who were quite aware of how memory contrained they were, and even they were surprised still when we looked at the actual cost, to see just how costly things were. They were even more than they thought.

Then there are the people who just give up - I know Java is expensive, it's always expensive, there is nothing I can do about it, it's just a cost of object oriente programming in Java, and if I try to do anything, it's going to break my good design. And so, hopefully, one of the things we want to achieve is to raise awareness of cost. So that it's not a lost cause, there is some hope. It's possible to make informed tradeoffs, can't fix all problems. Identify places where good engineering can help by the developer, and then where people are going to hit a wall.

Finally, this is an issue for performance, not just for memory. There's a false dichotomy in a lot of people's minds that say well, if you have a lot of footprint 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, even something as simple as I am using so much of my heap for long-lived objects, then I have very little headroom for temporaries. And so my garbage collector has to run much more often.

Or I don't have enough room to size the caches for things I get from the database as large as I like, so I go back to the database more than I like, so this can have a huge performance cost.

Fortunately, bloated designs are not an inevitable consequence of object-oriented development.

1.3 Quiz

Understanding memory costs requires counting bytes, which may seem like a strange activity for a Java programmer, accustomed to rapid assembly of applications from assorted libraries. 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. To start you thinking about bytes, here is a quiz to test how good you are at estimating sizes of Java objects. Assume a 32-bit JVM.

```
Question 1: What is the size ratio in bytes Integer to int?

   a. 1:1
   b. 1.33:1
```

```
      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: Which statement is true about a HashSet compared to
               a HashMap with the same number of entries?

      a. It has fewer data fields and less overhead.
      b. It has the same number of data fields and more overhead.
      c. It has the same number of data fields and the same overhead.
      d. It has more data fields and it has less overhead.


   Question 4: Arrange the following 2-element collections in size order:

      ArrayList, HashSet, LinkedList, HashMap

   Question 5: How many collections are there in a typical heap?

      a. between five and ten
      b. tens
      c. hundreds
      d. thousands
      e. order(s) of magnitude more than any of the above

   Question 6: What is the size of an empty ConcurrentHashMap?
   (Extra Credit)
      a. 17 bytes
      b. 170 bytes
      c. 1700 bytes
      d. 17000 bytes
      e. 500 bytes


ANSWERS: 1d, 2e, 3b,
         4 ArrayList LinkedList HashMap HashSet,
         5e, 6c
```

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. 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.

Strings often consume half of the heap and 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 th 16-bit Unicode character set, but this accounts for only 16 of the 56 bytes. The rest is various kinds of overhead.

After strings, collections are the most common types of objects in the heap. In typical real programs, having 100's of thousands and 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 20 bytes more than another, which may seem insignificant, but in a production execution, the wrong choice can add 20 megabytes to the heap.

`ConcurrentHashMap`, compared to the more common collections in the standard library, is surprisingly expensive. If you are used to creating hundreds of `HashMaps`, then you might think that it is not a problem to create hundreds 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. The quiz gives some sense of how surprising the sizes are for the Java basic objects, like `Integers`, strings, and collections. When code is layered with multiple abstractions, memory costs become more and more difficult to predict.

In fact, as we will see in some of our examples, inside the Java library themselves, the low level libraries are calling other low level libraries. String buffer uses ;; Hashset interms of hashmap, and so forth. This is true for memory and performance.

Compared to systems languages like C, Java space costs are high, even for the most basic building blocks. This makes it all the more important for developers to be aware of memory costs.

1.4 Notation and Conventions

1.4.1 Entity-Collection Diagrams

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

A data model implementation begins with a conceptual understanding of the entities and relationships in the model. This may be an informal understanding,

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

While the Java

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

1.4.2 Defining Terms

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

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

- A *class* is a Java class. A class name, for example `String`, always appears in type-writer font.
- A *data model* is a set of classes that represents one or more logical concepts.
- Finally, an *object* is an instance of a class, that exists at runtime occupying a contiguous section of memory.

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:

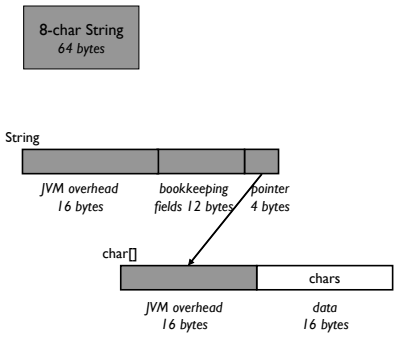


Figure 2.1: An eight character string in Java 6.

- *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.

- *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.

An 8-Character String: You learned in the quiz in Chapter 2 that an 8-character string occupies 64 bytes, which seems surprisingly high. The actual character data takes up 16 bytes, since Java supports the international character set, which uses 2-bytes per character. So it costs 16 bytes to represent the eight characters themselves. The other 48 bytes are pure overhead. This structure has a *bloat factor* of 75%. The actual data occupies only 25%. These numbers vary from one JVM to another, but the overhead is always very high. (Unless otherwise noted, all measurements in this book are from the 32-bit IBM Java 6 JVM.)
Figure 2.1 shows why the bloat factor is so high. Strings have all three kinds of overhead – JVM overhead, pointers, and bookkeeping fields. Every Java string is really two objects: a `String` that serves as a wrapper, and a `char` array with the actual characters. Two objects means two object headers, plus the pointer glueing the two objects together. The `String` object is entirely overhead, containing no real data. Part of its cost is three bookkeeping fields: a length and an offset, which have meaning only when this string is the result of a substring operation, and a saved hashcode. Adding all of this together, the overhead is 48 bytes.

If you were to design a string from scratch, 20% overhead might seem a reasonable target. For such a common data type, it is important to obtain a low overhead. Unfortunately, for a string to have only 20% overhead with its current implementation, it would need to be 96 characters long. As bad as this seems, this string representation does at least have the benefit that it amortizes away its overhead. The overhead cost is always 48 bytes, so overhead of a string approaches 0% for larger and larger strings. For some data structures it is not possible to amortize away overhead costs, as discussed in Section 2.4.

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

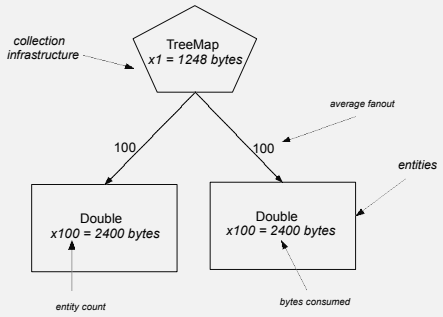
2.2 Entities and Collections

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

To help understand the health of more complex data structures, it is useful

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

The Entity-Collection (EC) Diagram



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

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

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

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

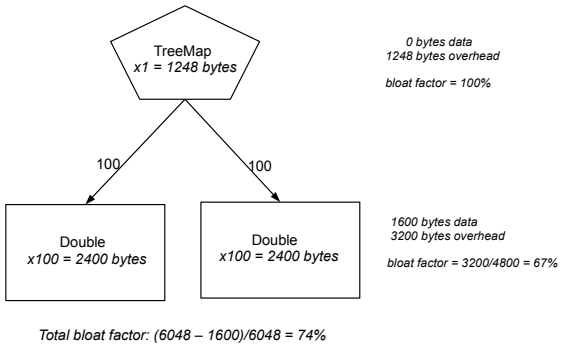


Figure 2.2: EC Diagram for 100 samples stored in a `TreeMap`

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 collected. The solution requires a data structure to store the samples.

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

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

Looking at the individual parts of the EC diagram can provide some insight into the source of this overhead. First, the sample timestamps and values are stored as `Double` objects. This is because the standard Java collection APIs take only `Object`s as parameters, not scalars, forcing you to box your scalars before putting them in a collection. Even with Java's autoboxing, the compiler still generates a boxed scalar in order to store a value in a standard collection.

A single instance of a `Double` is 24 bytes, so 200 `Doubles` occupy 4,800 bytes. Since the data is only 1,600 bytes, 33% of the `Double` objects is actual data, and 67% is overhead. This is a high price for a basic data type.

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

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

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

2.3 Two Memory-Efficient Designs

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

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

`Collections` each can take an `ArrayList` and a `Comparable` object as parameters. To take advantage of these methods, the new `Sample` class has to implement the `Comparable` interface, so that two sample timestamps can be compared:

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

    private final double timestamp;
    private final double value;

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

    public double getTimestamp() {
        return timestamp;
    }

    public double getValue() {
        return value;
    }

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

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

```
public class Samples {

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

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

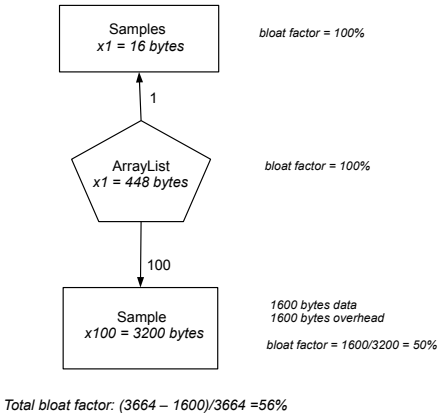


Figure 2.3: EC Diagram for 100 samples stored in an `ArrayList` of `Samples`

```
        }

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

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

While this is a big improvement, 56% overhead still seems high. Over half the memory is being wasted. How hard is it to get rid of this overhead completely? Eliminating overhead means eliminating objects altogether. This is not a recommended practice, unless memory is extremely tight, and there is no other choice. It is none the less an interesting exercise to compare a best case design with the other designs. The most space-efficient design uses two parallel arrays of doubles. One array stores all of the sample timestamps, and

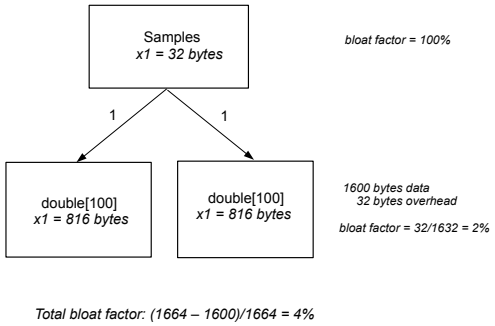


Figure 2.4: EC Diagram for 100 samples stored in two parallel arrays

the second stores all of the values. These arrays can be stored in a wrapper class that implements a map interface.

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

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

2.4 Scalability

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

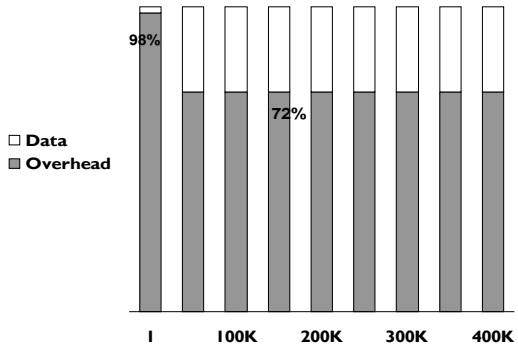


Figure 2.5: Health Measure for the `TreeMap` Design Shows Poor Scalability

design will scale much earlier.

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

To understand why this happens, recall that the infrastructure of `TreeMap` is made up of nodes, with two 64-element arrays hanging off of each node. As samples are added, the infrastructure grows, since new nodes and arrays are being created. Also, each additional sample has its own overhead, namely the JVM overhead in each `Double` object. When the `TreeMap` becomes large enough, the *per-entry overhead* dominates and hovers around 72%. The bloat factor is larger when the `TreeMap` is small. In contrast, for small `TreeMap`s, the fixed cost of the initial `TreeMap` infrastructure is relatively big. The `TreeMap` wrapper object alone is 48 bytes. This initial fixed cost is quickly amortized away as samples are added.

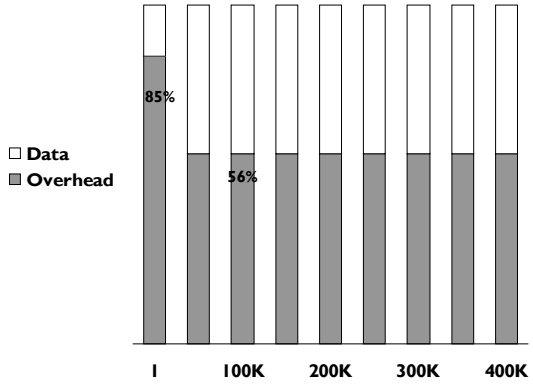


Figure 2.6: Health Measure for the `ArrayList` Design

Fixed vs Per-Entry Overhead

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

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

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

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

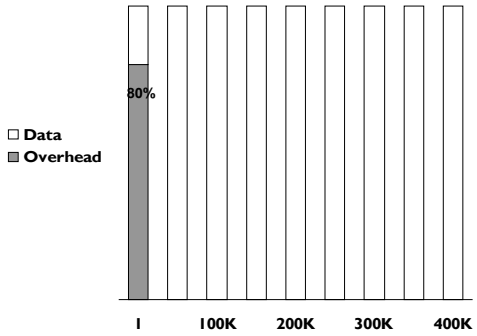


Figure 2.7: Health Measure for the Array-Based Design Shows Perfect Scalability

difference.

2.5 Summary

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

- The *memory bloat factor* measures how much of your the memory usage of your design is real data, and how much is an artifact of the way you have chosen to represent it. This metric tells you how much room there is for improvement, and if in fact you are paying a reasonable price for what you are getting.

- Complex data structures are built up from data entities and collections of these entities. The *Entity-Collection (EC) Diagram* shows the costs associated with the different parts of a complex data structure. These diagrams help you to compare the memory efficiency of different representation choices.

- By classifying the overhead of a collection as either *fixed* or *per-entry*, you can predict how much memory you will need to store very large collections. Being able to predict scalability is critical to meeting the requirements of larges applications.

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

Part II

Modeling Data Types

Chapter 3

Delegation

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

3.1 The Cost of Objects

There is no Java library method that returns the size of an object. This is by design. Unlike systems languages like C, a Java programmer is not supposed to know either the size of an object or how it is laid out in memory. The JRE, not the programmer, manages storage, and the JRE has the freedom to implement its own storage policy. However, knowing the sizes of objects is necessary for understanding memory requirements and scalability. Fortunately, it is not hard to estimate the size of an object, and a good estimate of the most prevelant classes is usually sufficient for making intelligent design choices. You need to know just a few basics, starting with the number of bytes needed to store primitive data types. These sizes are defined in the Java language specification [1], and are given in Table 3.1.

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, ad-

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

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

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

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

dresses that are multiples of 8. To show how implementations differ, Table 3.2 gives object header and alignment costs imposed by two JREs, SUN Java 6 (update 14) and IBM Java 6 (SR4), both for 32-bit architectures.

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

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

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

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

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

Class	Data size	Sun Java 6 (u14) Header	Sun Java 6 (u14) Alignment fill	Sun Java 6 (u14) Total bytes	IBM Java 6 (SR4) Header	IBM Java 6 (SR4) Alignment fill	IBM Java 6 (SR4) 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.

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

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

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

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

The size of a `SimpleEmployee` is 40 bytes:

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

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

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

Estimating Object Sizes

The size of an object can be estimated as follows:

1. Add together the sizes of all of the fields and the object header. Fields include those in the object's class and all of its superclasses.
2. Round up the result to the next multiple of the object alignment.

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

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

3.2 The Cost of Delegation

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

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

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

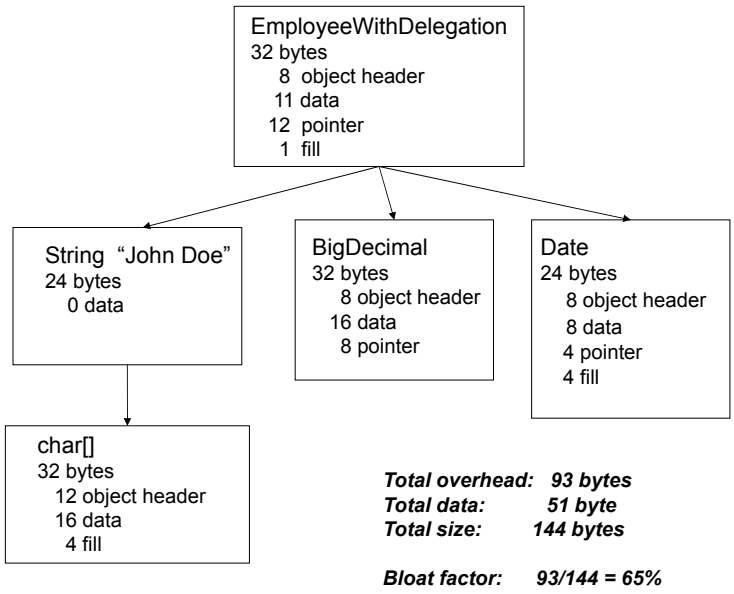


Figure 3.1: The memory layout for an employee "John Doe".

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

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

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

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

In the spirit of keeping things simple, Java does not allow you to nest objects inside other objects, to build a single object out of other objects. You cannot

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

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

3.3 Fine-Grained Data Models

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

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

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

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

class EmergencyContact {
    ContactPerson contact;
    ContactMethod preferredContact;
```

```
}

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

class ContactMethod {
    ContactPerson owner;
}

class PhoneNumber extends ContactMethod {
    byte[] phone;
}

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

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

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

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

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

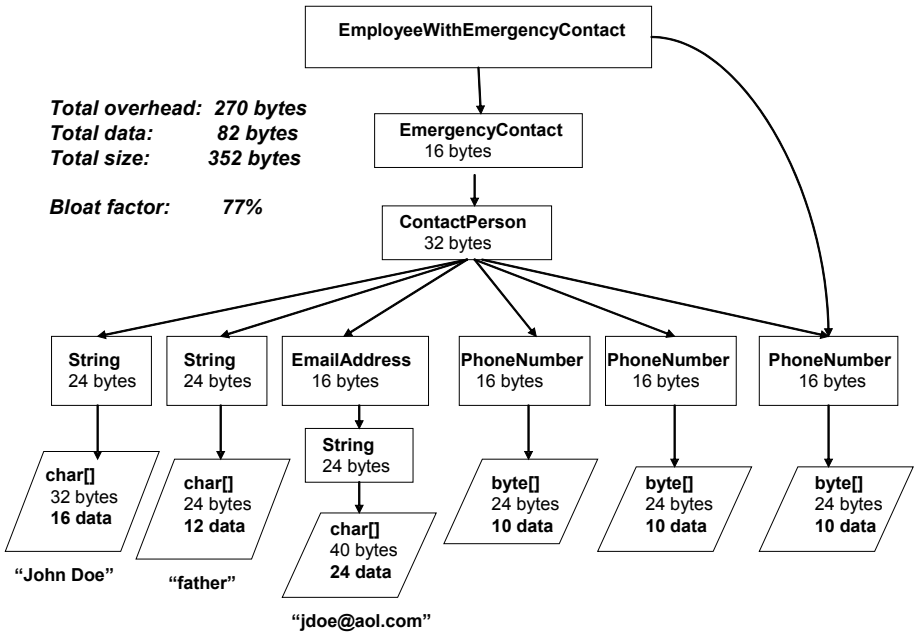


Figure 3.2: The memory layout for an employee with an emergency contact.

```
    PhoneNumber work;
    ContactMethod preferredContact;
}
```

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

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

class ContactPerson {
    PreferredContactMethod preferred;
    String name;
    String relation;
    String email;
```

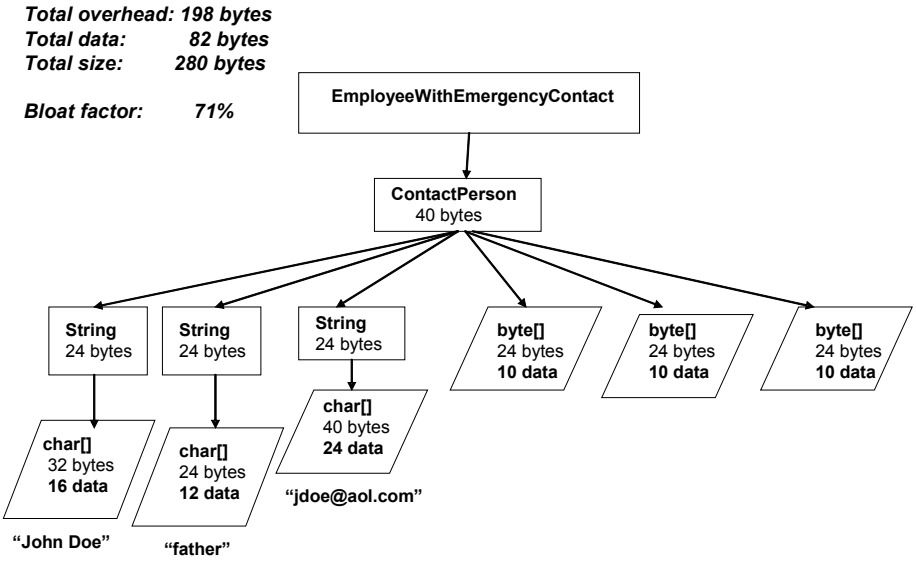


Figure 3.3: Memory layout for refactored emergency contact.

```
    byte[] cellPhone;
    byte[] homePhone;
    byte[] workPhone;
}
```

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

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

3.4 Large Base Classes

As discussion in the last section, highly-delegated data models can result in too many small objects. Occasionally, you run across a highly-delegated data model where the delegated objects are large. This can happen when delegated classes inherit from a large base class. When fine grained data modeling is combined with inheriting from large base classes, memory costs multiply and can become prohibitive.

A frequent data management requirement is to track creation and update information, that is, when data is created or updated and by whom. Here

is a base class, taken from a real application, that stores create and update information.

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

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

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

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

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

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

3.5 64-bit Architectures

If your application does not fit into memory, perhaps moving to a 64-bit architecture will save you. However, to support a 64-bit address space, more memory

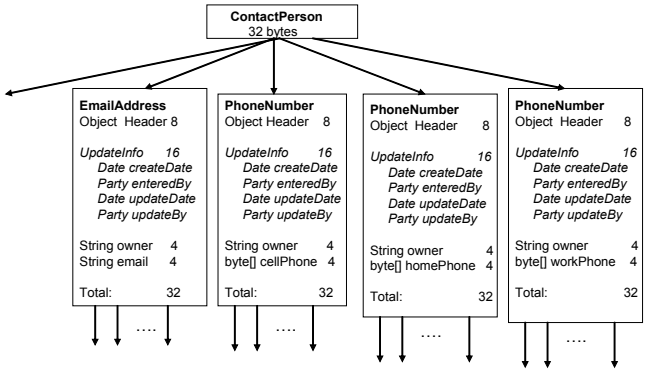


Figure 3.4: The cost of associating `UndateInfo` with every `ContactMethod`.

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 3.5. The 64-bit string is 50% bigger than the 32-bit string. All of the additional cost is overhead.

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

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

3.6 Summary

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

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

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

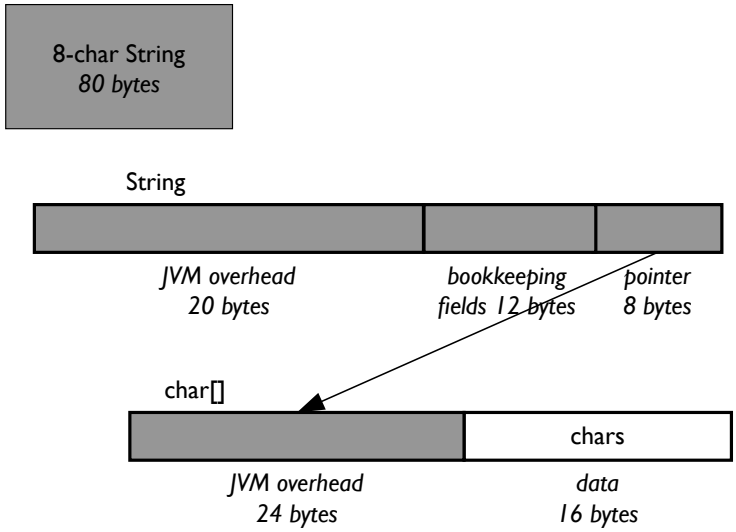


Figure 3.5: The memory layout for an 8 character string by a 64-bit JRE.

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

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

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

Chapter 4

Reducing Object Bloat

Chapter 5

Sharing Immutable Data

Part III

Modeling Relationships

Chapter 6

The Cost of Java Collections

Chapter 7

Reducing Collection Bloat

Part IV

Lifetime Management

Chapter 8

Common Lifetime Patterns

It would be great if programmers could be in charge only of hooking together and populating their data structures, leaving the JRE responsible for the details of object allocation and reclamation. The Java runtime does indeed help manage some aspects of these management tasks, but leaves a good deal of work in your hands. In Java, you needn't explicitly free objects, and in that way a managed language is a big step up from a language like C. However, the ultimate promise of automatic memory management, that you can create objects without regard for messy details of storage management, doesn't play out ideally in practice. Unless you are careful, your program will suffer from bugs such as memory leaks, and suffer from poor performance. Furthermore, if your objects don't easily fit into the limits of a single Java process, and you need to manage, explicitly, shuttling them in and out of the Java heap.

Unfortunately, then, it is necessary to plan out the lifetime of your objects. Your application needs some objects to live forever and it needs the rest to die a timely death. This task of managing object lifetime is made simpler by the fact that there aren't innumerable ways in which objects live and die. For the most part, your objects will fall into one of five common patterns of lifetime. Table 8.1 summarizes these five important patterns.

	Lifetime Property	Example
Section 8.2	Temporary	new parser for every date
Section 8.3	Needed Forever	product catalog
Section 8.4.1	Correlated with Object	object annotations
Section 8.4.2	Correlated with Phase	DOM used only for parsing
Section 8.5	Deferred Deletion	session state

Table 8.1: Five important categories of object lifetime.

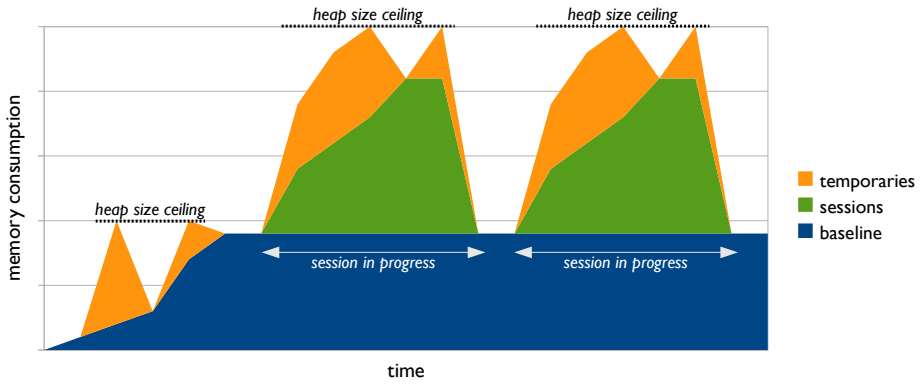


Figure 8.1: Memory consumption, over time, typical of a web application server.

8.1 Examples from a Server Application

Configuring memory settings is an iterative process. It usually involves a fair amount of trial and error, as one tunes the various knobs to balance memory consumption and application performance. These knobs affect things like the size of the Java heap, how many entries a cache should hold, and the timeout value for these caches. This is usually a process of black box tuning: twist a knob, and see how overall performance changes. In addition to being hit and miss, it is also quite prone to bugs. If you set the size of a cache too high, you risk poor performance due to excessive garbage collection, and even possibly process failures, due to running out of heap space.

Tuning memory consumption in long-running applications, such as servers or integrated development environments, is a particularly thorny issue. Improperly managing memory in a short-running application may not be the end of the world. In an application that runs more or less forever, mistakes can pile up over time. In addition, they synthesize information from a diverse array of sources, each with its own performance trade-offs. As such, caching plays a large role in these applications. Consider an example from a server aplication.

Object Lifetimes in a Server Application: A web application commerce server preloads catalog data into memory to allow for quick access to this commonly used data. It also maintains data for users as they interact with the system, browsing and buying products. Finally, it caches the response data that comes from a remote service provider that charges per request. How does Java heap consumption vary over time? Which heap size fluctuations indicate a problem, and which are expected behavior?

The heap consumption of this application will fluctuate over time. A timeline view expected memory consumption helps to illustrate the five main kinds of

object lifetime. It visualizes memory during the lulls and peaks of activity, as requests are processed and when sessions time out, and as the server starts up. Figure 8.1 shows an example timeline for a server that, as it starts up begins to load catalog data into the Java heap. Then, it responds to one request at a time. The total height of the area under the curves represents the memory consumption at that point in time. In this example, as the server starts up, it begins to load catalog data into the heap. This data will be used for the entire duration of the server process. The Java objects that represent this catalog are objects that are needed forever. In the timeline picture, this data is respresented by the lowest area, labeled *baseline*. Notice how it ramps up quickly, and then, after the server has reached a "warmed up" state, memory consumption of this baseline data evens out on a plateau for the remainder of the run.

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

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

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

The catalog data should last forever, while the session data lives for some bounded period of time. It is possible that session state will live beyond the end of your session, but nonetheless it has a lifetime that is bounded. If, due to an

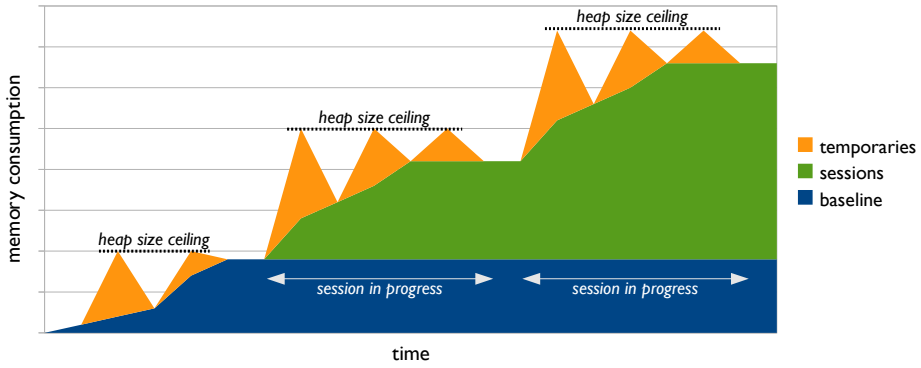


Figure 8.2: If session state is not cleaned up properly, a memory leak is the result. This means more frequent garbage collections, and ever increasing heap size.

Memory Leaks are still possible, even with automatic garbage collection!

bug, part of this session state is not reclaimed, the application will leak memory. Though it is supposed to have a bounded lifetime, it accidentally lives forever. In this case, over time, the amount of heap required for the application to run will increase without bound. Figure 8.2 illustrates this situation, in the extreme case when all of session state leaks. Over time, the area under the curve steps higher and higher.

Finally, this example server caches data from some expensive third-party data source. When caching data inside of Java objects, there is a fourth effect on the timeline landscape. The cached data must be configured properly to live long enough to be useful. It also must not occupy so much of the heap so as to leave little headroom for temporaries. Figure 8.3 shows an example where the cache has probably been configured to occupy too much heap space. Observe how, compared to the other timeline figures, there is little headroom for temporaries. In this case, the result is more frequent garbage collections. If the cache were sized to occupy an even greater amount of heap space, it is possible that there would no longer be room to fit session data. The result in this case would be failures in client requests. So, as you can see, sizing caches is important. As discussed later, it is very tricky to properly size caches, and is something best left in the hands of the JRE.

8.2 Temporary Objects

If your application is like most Java applications, it creates a large number of temporary objects. They hold data that will only be used for a very short interval of time. It is often the case that the objects in these transient data structures are only ever referenced by local variables. For example, this is the case when you populate a `StringBuffer`, turn it into a `String`, and then ultimately (and only) print the string to a log file. The point at which all

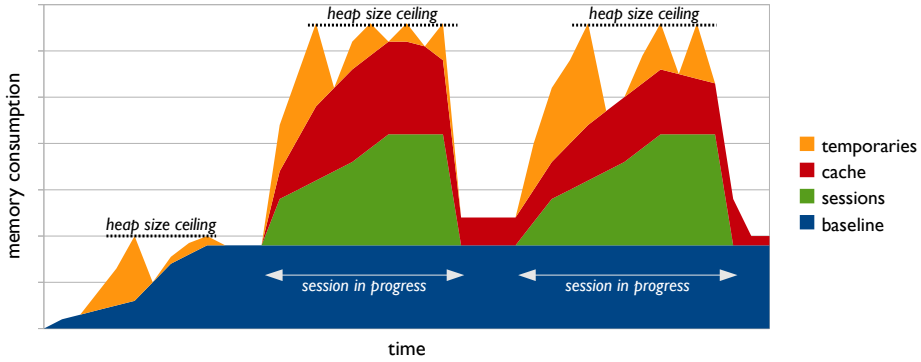


Figure 8.3: When a cache is in use, this leaves less headroom for temporary object allocation, often resulting in more frequent garbage collections.

of these objects, the strings and character arrays, are no longer used is only shortly after they are constructed. These objects serve as transient homes for your data, as it makes its way through the frameworks and libraries you depend on. Temporaries are often necessary to bridge separately developed code and enable code reuse: as long as you can convert your data layout into a form that an API requires, then you can reuse the functionality it provides.

In many cases, you need do nothing special to manage the temporary objects your application creates. After all, generational garbage collectors these days do a very good job digesting a large volume of temporary objects. In a generational garbage collector, the JRE places temporary objects in a separate heap, and thus it need only process the newly created objects during its routine scan.

Unfortunately, in Java it is pretty easy to create a high volume of temporary objects. Say your application fills up the temporary heap ever second. In this case, based on the common speeds of garbage collectors, your application could easily spend over 20% of its time collecting garbage. Is it difficult to fill up the temporary heap once per second? Typical temporary heap sizes run around 128 megabytes. Say your application is a serves a peak of 1000 requests per second, and creates objects of around 50 bytes each. If it creates around 2500 temporaries per request, then this application will spend 20% of its time collecting garbage.

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

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

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

8.3 Objects Needed Forever

A `SimpleDateFormat` object in the previous example was created in every loop iteration, and never used again. An improvement would be to create and use a single formatter for the remaining duration of the run. Though the Java 6 documentation does not say so explicitly, it is safe to reuse a single instance of this object multiple times. You must be careful to remember that it is not safe to do so in multiple threads. The next chapter will discuss remedies to this problem. The updated code for the parse method would be:

```
static final DateFormat fmt = new SimpleDateFormat();
```

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

A static field of an object is one way that Java gives you to indicate that you want an object to live forever.

8.4 Objects with Correlated Lifetimes

Many objects are needed for a very specific interval of time. This interval is usually defined either by the lifetime of another object, or by the duration of a method call. Once that other object is not needed, or once that method returns, then these objects are also no longer needed. These are the two many cases of objects with correlated lifetime.

8.4.1 Objects that Live and Die Together

Normally, if you need to augment the state stored an object, you modify the source code of some existing classes. For example, to add a secondary mailing address to a `Person` object, you can add a field to that class, and update the initialization and marshalling logic accordingly. Sometimes, you will find it necessary to associate information with an object that is, for one reason or the other, locked down.

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

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

You could store the annotations in a map that is keyed by the original object, say of type `T`:

```
Map<T, Date> timestamps = new HashMap<T, Date>();

void addTimestamp(T t) {
    timestamps.put(t, new Date());
}
Date getTimestamp(T t) {
    return timestamps.get(t);
}
```

This solution will function correctly, but suffers from a *memory leak*. As the application runs, it will consume greater amounts of Java heap, up until the point when the JRE runs out of heap space to allocate any more objects. This solution leaks memory, because the `timestamps` map introduces a reference to the main objects. When the garbage collector scans the heap to see which objects are still alive, the references in this map will be among those that keep the objects alive. The next chapter discusses these issues in more detail. An improved solution would use the `WeakHashMap` from the Java standard libraries. By replacing the initialization of the `timestamps` map, we have the same functionality as before, but no memory leak.

```
Map<T, Date> timestamps = new WeakHashMap<T, Date>();
```

Note that this same situation can hold even if you are able to modify the class definition. A common scenario requires annotations on only a subset of all instances of a class. In this case, is it not worth paying the memory cost to have the ability to annotate every single instance. Therefore, this is another case where a solution of side annotations, stored in a `WeakHashMap`, shines.

8.4.2 Objects that Live and Die with Program Phases

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

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

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

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

In server applications, most objects created within the scope of a request should not survive the request. Most of these *request-scoped* objects are not

used by the application after the request has completed. In the absence of application or framework bugs, they will be collected as soon as is convenient for the runtime. In this example, the lifetime of objects during a request are *correlated* with a method invocation: when the servlet `doGet` or `doPut` (etc.) invocations return, those correlated objects had better be garbage collectible.

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

- Registrars, where objects are registered as listeners to some service, but not deregistered at the end of a request.
- Doubly-indexed registrars. Here the outer map provides a key to index into the inner map. A leak occurs when the outer key is mistakenly overwritten mid-request. This can happen if the namespace of keys isn't canonical and two development groups use keys that collide. It can also happen if there is a mistaken notion, between two development groups, of who owns respnsibility of populating this registrar.
- Misimplemented hashcode or equals, which foils the retrieval of an object from a hash-based collection. If developers checked the return value of the `r`emove method, which for the standard collections would indicate a failure to remove, then this bug could be easily detected early; but developers tend not to do this.

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

8.5 Objects with Deferred Deletion

The last important facet of object lifetime comes from those objects that stay around beyond the scope of any one method or object. These objects must survive for some indeterminant amount of time. In some cases, this period based on the profitability of keeping them around. In other cases, objects need to be kept around for operations that span several independent operations across multiple threads. There are three important cases of objects that need to be reclaimed in some deferred fashion: caches, sharing pools, and resource pools.

8.5.1 Caches: Buying Time with Space

If the data stored in an object is cheap to recompute or refetch from an external

A **Cache** is a map that holds expensive data values, each accessed by a unique key.

data source, then a good policy would be to treat the object as a temporary. Performing a few dozen machine instructions is very likely to be a worthwhile trade-off, if memory consumption is the limiting resource for scaling up. You do have to be careful, though. Recall our earlier example that creates a new instance of the date parser `SimpleDateFormat` for each iteration of a loop. Here, as is quite common, something expensive to compute, that `SimpleDateFormat`, is treated as a temporary. If the data stored in an object is expensive to recompute or refetch, and there is some chance it might be used in the future, then it is worthwhile to keep it around.

Finding the right balance of time and space is the goal of a good cache implementation. The expense of re-fetching data from external data sources and recomputing the in-memory structure can often be amortized, at the expense of stretching the lifetime of these data structures. By increasing the actual lifetime on an object you will very likely increase peak memory consumption. A good cache defers the time that an object will be reclaimed, as long as there is sufficient space to handle the flux of temporary objects your application creates. It holds on to a data structure after the current operation is finished with it, in the hope that other operations in the near future will reuse it.

8.5.2 Sharing Pools: Avoiding Data Replication

A **Sharing Pool** stores canonical instances of data values that would otherwise be replicated in many objects.

A cache amortizes the time cost of fetching or initializing data. An orthogonal issue lies in the memory expense of storing many copies of the same data throughout the heap. This is especially a problem with strings. Heaps can often store the same string a dozen times.

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

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

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

	cache	sharing pool	resource pool
Addressing the Contents	by key	by index	by key
Elements Interchangeable?	no	no	yes
Multiple Users per Element?	yes	yes	no
Data Persists Across Uses?	yes	yes	no

Table 8.2: Comparing the charaacteristics of three mechanisms for keeping data or objects around for indefinite periods of time.

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

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

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

```
enum PropertyName = {...};
void handleNextEntry() {
    PropertyName key = getNextPropertyName();
    Object value = getNextString().intern();
    map.put(key, value);
}
```

There is an important variant of a sharing pool called the Bulk Sharing Pool. Like a normal sharing pool, the goal of a bulk sharing pool is to amortize the memor costs of storing data. However, rather than mitigate the costs of data duplication, a bulk sharing pool aims to amortize the costs of Java object headers across the elements in a pool. This is a topic that stretches notions of how to store data beyond the normal Java box, and so will be discussed, along with many similar matters, in Chapter 10.

8.5.3 Resource Pools: Amortizing Allocation Costs

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

A **Resource Pool** is a set of interchangeable storage or external connections that are expensive to construct.

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

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

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

Per-thread Singleton versus Resource Pool: Your application accesses a remote resource. Why not keep one connection persistent per thread? What's the point of a resource pool in this case?

Chapter 9

Managing Object Lifetime

Designing a lifetime management strategy requires that you take the tools that Java provides, and combine them with other strategies implemented on top of Java. The built-in mechanisms, nominally, handle some of the common patterns of object lifetime. Unfortunately, they often appear in the form of low-level JVM hooks, and so require careful coding to make correct use of them.

In a *well-behaved* application, an object's lifetime spans its allocation, use, and the short period during which the JRE takes control and reclaims the space. For some subset of an object's actual lifetime, that is the time from creation to reclamation, your application will make use of the data stored in its fields. Figure 9.1 illustrates the lifecycle of a typical object in a well behaved application.

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

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

For each iteration of this loop, this code takes a date that is represented

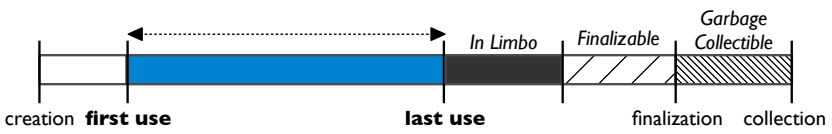


Figure 9.1: Timeline of the life of an object.

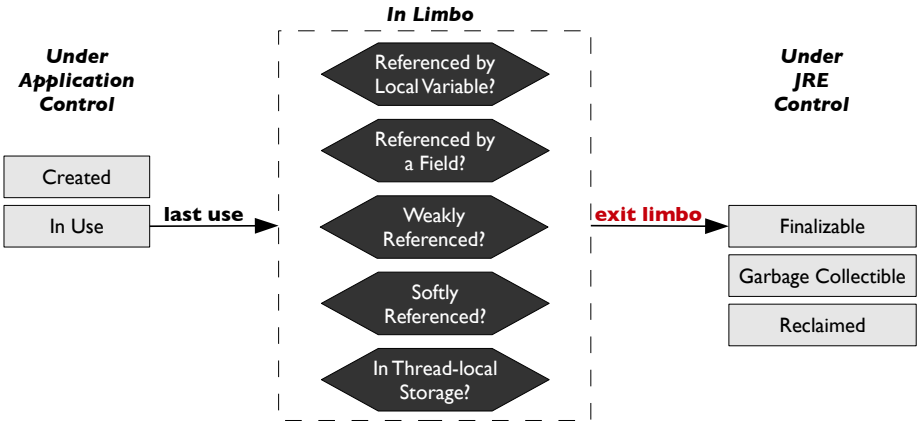


Figure 9.2: After its last use, an object enters a kind of limbo: the application is done with it, but the JRE hasn't yet inferred this to be the case. When an object exits limbo depends on the way it is referenced.

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

Objects in Limbo

In limbo, an object will never be used again, or at least not for long time, but the JRE doesn't yet know that this is the case. The object hangs around, taking up space in the Java heap until the point when it exits limbo.

The `pos` object represents to the parser the position within the input string to begin parsing. The implementation of the `parse` method uses it early on in the process of parsing. Despite being unused for the remainder of the parsing, the JRE does not know this until the current iteration of the loop has finished. This time in limbo also includes the entirety of the call to `System.out.println`, an operation entirely unrelated to the creation or use of the parse position object. Once the current loop iteration finishes, these two objects will exit limbo, and become garbage collectible.

reachable only from	moment when object exits limbo
nothing	immediately
local variable	after scope exits
instance field of an object	when that object exits limbo
static field of an object	when that object's class is unloaded
field of `WeakReference`	immediately
field of `SoftReference`	approximately LRU
... with `ReferenceQueue`	... then, after removed from queue
entry in thread local storage	when that thread dies

Table 9.1: When, or even whether, an object exits limbo depends upon how your program references it. If these references aren't explicitly overwritten, e.g. by your code expliclty assigning the reference to `null`, then an object only exits limbo under certain restricted circumstances.

9.1 Basic Management Mechanisms

The point at which an object exits its limbo depends upon how it is referenced by other objects. One can always assure that an object exits limbo by modifying your code to overwrite all references to that variable. If an object is no longer referenced at all, then it will exit limbo immediately.[1] For example, a common way to do this is by assigning references to the value `null`. This is tricky in many cases, because it may not be easy to know where all those references emanate from. Who is to say that, when one calls the `parse` method of a `SimpleDateFormat` object, that it does not squirrel away a reference to the `ParsePosition` passed as a parameter? The API contract for `parse` makes no such claims, one way or the other. This is certainly calls to mind the worst of the days of explicitly managing memory in a language like C.

Still, one can't always rely on automatic mechanisms to guide an object out of limbo in a timely fashion. Figure 9.2 and Table 9.1 illustrate how an object may exit limbo. A garbage collector only knows that an object is ready to be collected based on *reachability*: how the objects point to eachother. If, as in the `ParsePosition` or `SimpleDateFormat` objects from our example, the object is referenced only by a local variable of a method, the JRE will not consider reclaming its storage until the variable's scope exits; e.g. when the loop continues to the next iteration, or when the method returns, depending on the scope of the variable that references the object. If the object is referenced only by a field of another object, then it must wait for that other object to exit limbo before it can do so. An objects pointed to be only by a static field has a good chance of never being collected. A class only exits limbo when it is unloaded by the JREs class loading mechanism, which is unlikely to happen if it has static fields that reference other objects. Therefore, unless that field is

[1] Talk about reference cycles?

overwritten, objects pointed to by static fields are likely never to exit limbo.

9.2 More Complex Management Mechanisms

There are important lifetime management policies that are not expressible via the normal mechanisms. When referenced by a local variable, an object lives or dies with the scope of the variable; when referenced by another object, it lives or dies along with that object (both, of course, in the absence of overwriting a reference). The Java specification provides three other mechanisms that let you guide the JRE to the right time for an object to exit limbo: weak references, soft references, and thread-local storage.

9.2.1 Weak References

Java provides a low-level mechanism that one can use to implement a correlated lifetime memory management policy, in the form of weak references. The standard library exposes this feature in the class `java.util.WeakReference`. Using this class correctly is difficult, because the semantics of weak references does not directly map to any important application-level use cases.

Definition 1 *When calling the constructor* `new WeakReference(obj)`*, this new instance will maintain a reference to* `obj`*, however* `obj` *will exit limbo, and become a candidate for cleanup processing by the JRE, as if that reference did not exist.*

When used in this way, weak references don't keep an object alive longer than it otherwise would have, in the absence of weak references. This seems pretty far from anything an application might need. Still, you can use this low-level feature to implemented correlated lifetime, as long as you're careful. When used to implement correlated lifetime policies, weak references may indeed delay the time till an object exits limbo. Improper use of a `WeakReference` will render your code worse off than before. It is quite possible that you will not have achieved the correlated lifetime that you need, but in a way that is hard to tell. Even worse, when using weak references, you can introduce memory leaks. Be very cautious when using them, and follow these rules.

Rules for Using `java.util.WeakReference`

In order to assure that you use of weak references works properly, you must follow three rules:

- Your instances of `WeakReference` must not, directly or indirectly, maintain a non-weak reference to the object you wish to annotate. It is best to maintain a collection of non-weak references to the annotated objects, and use a local variable, or a subclass of `WeakReference` for any other ways you refer to the annotated object.
- You must ensure that the `WeakReference` objects (or subclasses thereof) that you create will exit limbo no sooner than the annotated objects. Otherwise, these objects themselves, following the rules of Table 9.1, will exit limbo too early. So, if the annotated objects are referenced by a collection that is in turn referenced by a static field, then the same must be true for the weak reference objects as well.
- Since you have to maintain two, parallel, collections to maintain references to the annotated objects, and to the annotations, you must ensure that exit limbo in lockstep. It is best to create your instances of `WeakReference` with a `ReferenceQueue` parameter. You must periodically call `poll` on this queue, and remove the `WeakReference` instances from the parallel registry.

You can see that, despite the benefit of some support from the JRE, there is quite a bit of memory management that you are left with. Luckly, the standard library ships with a `WeakHashMap` which deals with some, but not all, of the legwork of managing weak references. It will handle the second and third items, but not the first. It is still up to you to ensure that none of your annotations, directly or indirectly, reference the annotated object.

Timestamp Annotation: How can you associate a timestamp with an object in a way that avoids memory leaks and that scales well to a highly concurrent workload?

We can start with the following code:

```
class TimestampAnnotation<T> {
    T t;
    long timestamp;
}
List annotations;
```

```
for (String string : inputList) {
    ...
    annotations.add(new WeakReference(new Wrapper<String
        >(string)));
    ...
}
```

Despite your use of `WeakReference`, you would find that neither the main object (the strings), nor the annotations, would ever be collected. This code has two memory leaks. One of the leaks is due to a violation of the first principal of the use of weak references: the annotations strongly reference the objects being annotated. It is not always this easy to debug problems in using weak references. Your application will hold on to objects that you didn't expect. Quite often, it is difficult to even know that there is a problem in the first place! The application may behave normally, except that it will consume more memory than necessary; if this extra memory consumption pushes it over your maximum heap size, then your application will crash — you will know something is wrong, but diagnosing this type of problem, a memory leak, is quite difficult. It is better to keep the three principles of weak references in mind, and design in a way that avoids memory leaks in the first place. Your annotations can be modified to use a `WeakReference` to the main object:

```
class TimestampAnnotation<T> {
    WeakReference<T> t; // annotation only weakly refs
        main object
    long timestamp;

    TimestampAnnotation(T t) {
        this.t = new WeakReference(t);
    }
}
```

In this case, the annotation has no normal references to the annotated object, and so it obides by the first rule of weak references. If you remember from Chapter 3, the code can be improved further to avoid the cost of delegation. This version of the annotation class extends `WeakReference`:

```
class TimestampAnnotation<T> extends WeakReference<T> {
    long timestamp;

    TimestampAnnotation(T t) {
        super(t);
    }
}
```

Unfortunately, both of these updated versions h

9.2.2 Soft References

9.2.3 Properly Draining a Reference Queue

9.2.4 Thread-local Storage

Chapter 10

Outside the Java Box

Nodes and Edges:

10.1 The Bulk Sharing Pool

The bulk storage that backs a set of data items, each of the same type. Objects share the data by indexing into the pool.

10.2 Column-oriented Storage

10.3 Representing Relationships

10.4 Memory Mapping

Chapter 11

Lifetime Management in Other Languages

11.1 C++: Smart Pointers

auto_ptr, single-owner notion, auto-free

11.2 C#: Value Types

11.3 Ada95: Storage Pools

Appendix A

Tools to Help with Memory Analysis

Bibliography

[1] James Gosling, Bill Joy, Guy Steele, and Gilad 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*, 36(1), 2006.

Index