

Chapter 3

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.

3.1 Distinguishing Data from Overhead

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

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

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

Things To Remember 1 — 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.

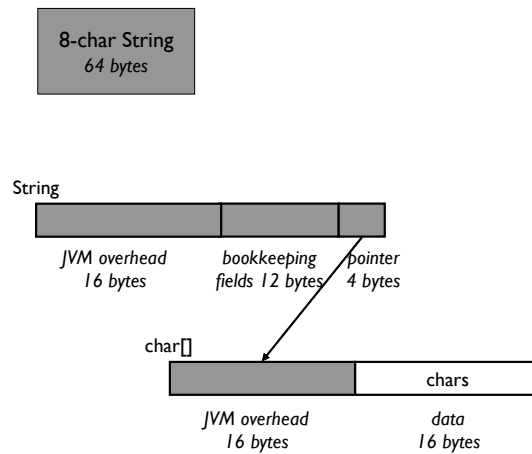


Figure 3.1: An eight character string in Java 6.

Example 1 (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 3.1 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 3.4.

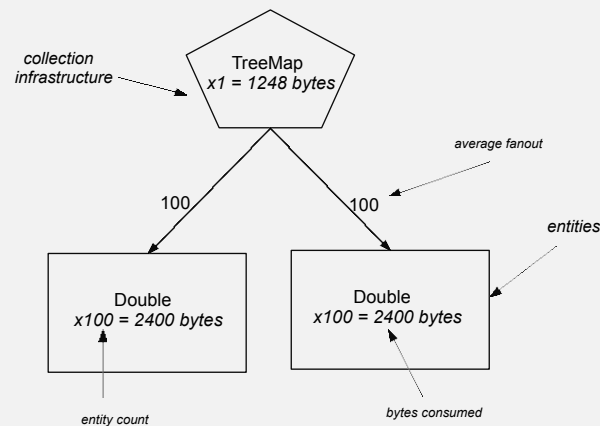
Strings, in particular short strings, are pervasive. It is important to be aware of the overhead of strings when incorporating them into your data design. Given the overhead of Java strings, the choices you make in Java need to be different than in a language like C, where the string overhead is minimal.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

3.2 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.

Things To Remember 2 — 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.

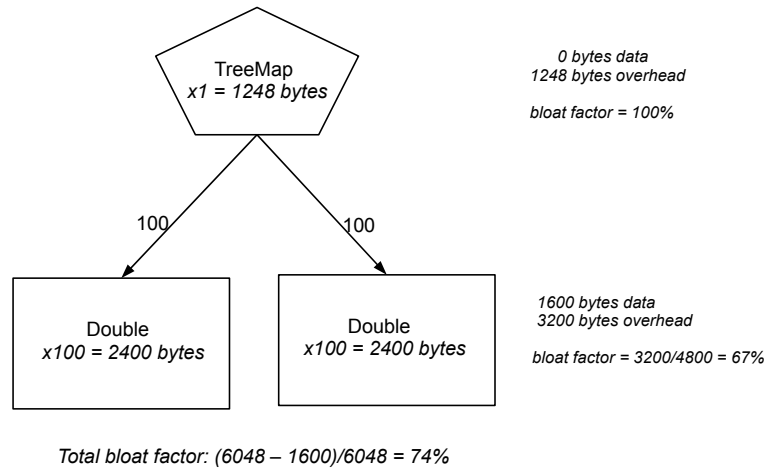


Figure 3.2: EC Diagram for 100 samples stored in a **TreeMap**

Example 2 (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 3.2. This diagram gives a schematic view of the **TreeMap** and the entities it contains, along with their costs. The total cost, obtained by adding up the entity and collection costs, is 6,048 bytes. The real data consumes only 1,600 bytes of this, since a double occupies 8 bytes and there are 200 of them. Therefore, the bloat factor is 74%.

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

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

implementation, which is more memory-efficient for this case, but still expensive.

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

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

3.3 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());
    }
}
```

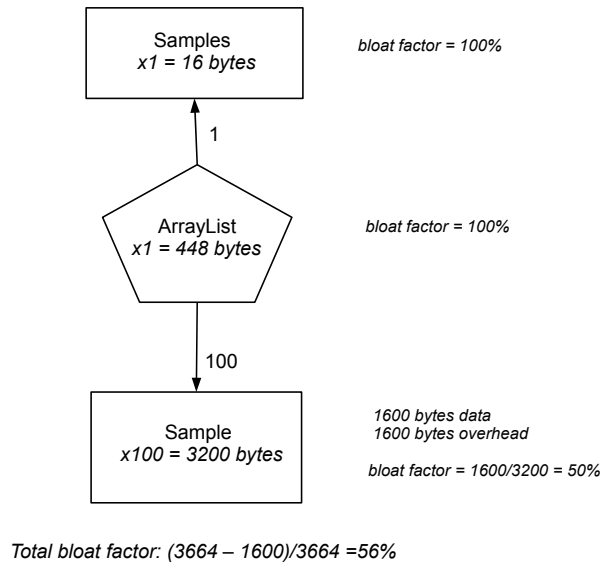


Figure 3.3: EC Diagram for 100 samples stored in an ArrayList of Samples

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();
    }

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

The EC diagram for 100 entries is shown in Figure 3.3. This design uses less memory and has better

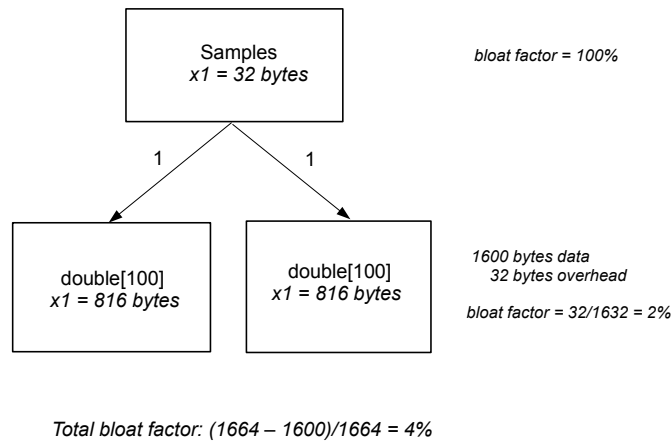


Figure 3.4: EC Diagram for 100 samples stored in two parallel arrays

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 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 3.4. The overhead is only 4%.

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

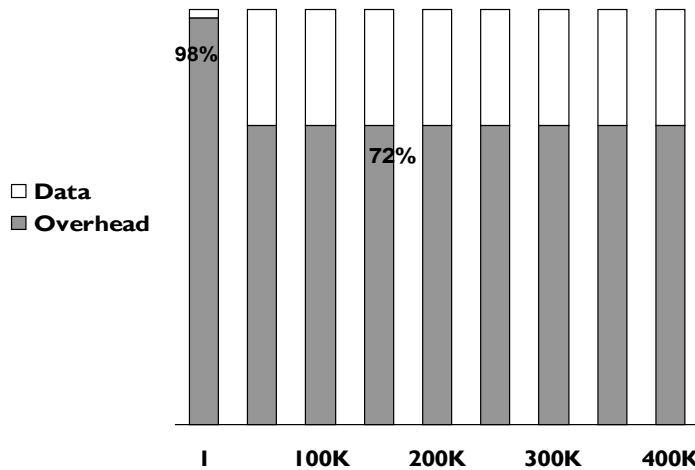


Figure 3.5: Health Measure for the **TreeMap** Design Shows Poor Scalability

3.4 Scalability

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

The basic question for predicting scalability is what happens to the bloat factor as a data structure grows. The **TreeMap** design has 74% overhead with 100 samples. With 100,000 samples, maybe the high overhead will be amortized away, that is, maybe this design will scale well, even if it is inefficient for small data sizes. The bar graph in Figure 3.5 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 **TreeMaps**, the fixed cost of the initial **TreeMap** infrastructure is relatively big. The **TreeMap** wrapper object alone is 48 bytes. This initial fixed cost is quickly amortized away as samples are added.

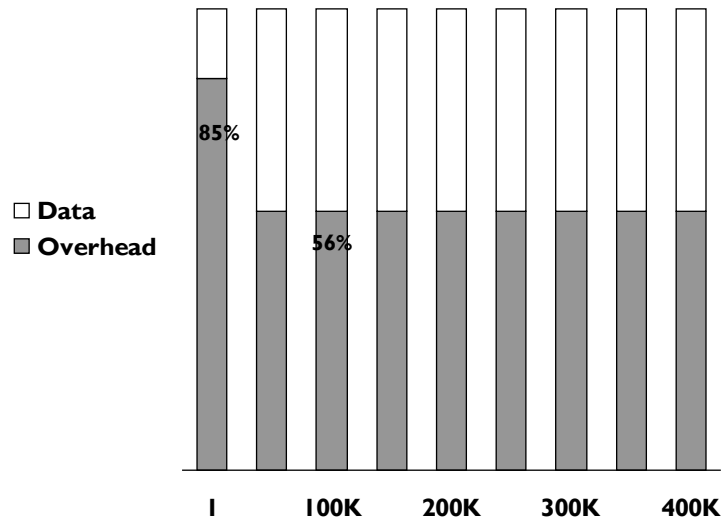


Figure 3.6: Health Measure for the `ArrayList` Design

Things To Remember 3 — 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 3.6 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 3.7 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 difference.

3.5 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

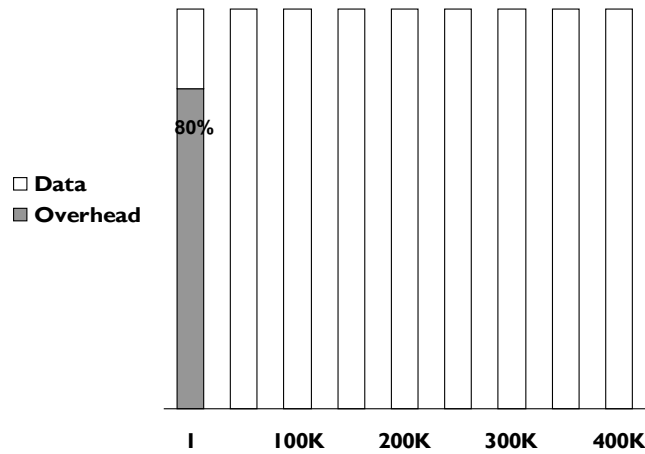


Figure 3.7: Health Measure for the Array-Based Design Shows Perfect Scalability

can pinpoint problems early on. This chapter described three conceptual tools to help evaluate the memory-efficiency of a design.

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

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

Chapter 4

Delegation

An important design decision when modeling data is how many Java classes to use to represent an entity. At one extreme, all of the attributes of an entity are represented by a single class, resulting in a coarse-grained design. At the other extreme, a main entity class delegates functionality to other classes, resulting in a fine-grained design. Delegation is a very popular pattern because of the flexibility it provides.

Programmers rarely think about memory costs when deciding how to model entities. Yet, the choices made at this design stage impact memory costs significantly. Coarse-grained designs may result in many objects with unused fields. While delegation is sometimes a way to avoid this problem, overly fine-grained data models can result in poor memory health from excessive object header overhead. This chapter explains how to evaluate object granularity design choices from a memory perspective. It begins with the costs of basic objects, and works up to examples from real applications.

4.1 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 JVM, not the programmer, manages storage, and the JVM has freedom to implement its own 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 is usually sufficient for making intelligent choices among design alternatives. You need to know just a few basics, starting with the size of the Java primitive types. These are given in Table 4.1.

Objects are much bigger. First, the JVM and the garbage collector require an object header that stores information such as the object's class, a lock, and an identity hashCode. For array objects, the header has an additional integer to store the number of array elements. Secondly, the hardware imposes alignment costs. The hardware may require 2-byte, 4-byte, or 8-byte alignment, depending on the type of the data. For example, integers are usually aligned on a 4-byte boundary, and some hardware might require a double to be aligned on an 8-byte boundary. A JVM may impose additional alignment requirements. This overhead is significant, especially if the object does not have much data in it.

The object header sizes and alignment costs vary depending on the JVM. To illustrate this variability, Table 4.2 gives the costs for both the SUN Java 6 (update 14) JVM and the IBM Java 6 (J9 SR4) JVM. These costs are for 32-bit architectures. It is important to keep in mind that these costs are for specific JVM releases only, and are subject to change in future releases.

Boxed scalars, which are objects with a single primitive data type field, are the simplest kind of object. For both JVMs, a boxed scalar is at least 16 bytes. Since both JVMs allocate objects on 8-byte boundaries, the size of an object must be a multiple of 8. Table 4.3 gives the sizes of boxed scalars.

There is a simple rule that holds for boxed scalars. The size of the object is obtained by summing the size of the header and the data, and then rounding it up to the nearest multiple of 8. The generalization of

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

Table 4.1: The sizes of Java primitive types

	Sun Java 6 (u14)	IBM Java 6 (J9 SR)
Object Header size	8 bytes	12 bytes
Array header size	12 bytes	16 bytes
Alignment	8 bytes	8 bytes

Table 4.2: The sizes of boxed scalar objects.

JVM		Sun Java 6 (u14)			IBM Java 6 (J9 SR4)		
Type	Data Size	Header	Alignment	Total bytes	Header	Alignment	Total bytes
Boolean	1	8	7	16	12	3	16
Byte	1	8	7	16	12	3	16
Character	2	8	6	16	12	1	16
Short	2	8	6	16	12	1	16
Integer	4	8	4	16	12	0	16
Float	4	8	4	16	12	0	16
Long	8	8	0	16	12	4	24
Double	8	8	0	16	12	4	24

Table 4.3: The sizes of boxed scalar objects.

this rule gives you a good way to estimate the size of any object. Often, this estimate turns out to be the real object size, but not always.

Things To Remember 4 — Minimum-Size Estimate Rule

Let *Header* be the size of an object header required by the JVM, and *Alignment* be the object alignment. That is, every object must be allocated at an address which is a multiple of *Alignment*. To estimate the minimum size of an object 1) sum up the sizes of all of its fields, including fields from superclasses, 2) add *Header* to this sum, and 3) round the result up to the next multiple of *Alignment*. The estimate is exact if the JVM packs and rearranges fields to fit into the smallest space. Otherwise, the actual size can be bigger than the estimated size.

Example 3 (Simple employee class.). Consider an `Employee` class with all primitive fields:

```
class Employee {
    int hoursPerWeek;
    boolean exempt;
    double salary;
    char jobCode;
    int yearsOfService;
}
```

Assume both *Header* and *Alignment* are 8 bytes. To apply the minimum-size estimate rule to `Employee` first add the primitive field sizes $4+1+8+2+4$ ($=19$), then add in the header size 8 ($=27$), and then round to the next multiple of 8. The result is 32 bytes. If the header is 12 bytes, the estimated size is also 32 bytes, since $19+12$ is 31, which again rounds up to 32.

How accurate are these estimates for the two JVMs? For the Sun JVM, an `Employee` object is exactly 32 bytes. The minimum-size estimate rule works well here, since the Sun JVM does a good job packing fields. For the IBM JVM, the real size of an `Employee` object is 40 bytes, not 32. The IBM JVM does not pack fields that are less than 4 bytes. In other words, the IBM JVM aligns all fields on 4 byte boundaries. When an object has several fields that are less than 4 bytes, such as booleans and chars, the minimum-size estimate rule can underestimate its size. For the Sun JVM, the bloat factor is 46%. For the IBM JVM, the bloat factor is 53%.

A modified version of the minimum-size estimate rule, where fields are never less than 4 bytes, works better for the IBM JVM. To apply this modified rule to `Employee`, first add the field sizes $4+4+8+4+4$ ($=24$), add in the IBM JVM header size 12 ($=36$), and then round up the result to 40 bytes.

Things To Remember 5 — Word-Aligned Estimate Rule

Let *Header* be the size of an object header required by the JVM, and *Alignment* be the object alignment. To estimate the size of an object 1) sum up the sizes of all of its fields including superclass fields, *assuming each field is aligned on a 4-byte boundary* 2) add *Header* to this sum, and 3) round the result up to the next multiple of *Alignment*.

Understanding the memory costs and scalability of a data model starts with understanding the cost of each object type. For this purpose, there are simple rules for obtaining pretty accurate estimates. The `Employee` example shows that object sizes may be different with different JVMs and different versions of the same JVM.

If you need the exact sizes of an objects, you can get them from HPROF heap snapshots. The `jmap` utility that comes with the Sun JVM distribution writes an HPROF heap dump:

```
jmap -dump:format=a,file=outputfile <pid of jvm process>
```

If you do not have `jmap`, you can enable the HPROF agent using the command line option:

```
-agentlib:hprof=heap=dump,format=a,file=outputfile
```

Sending a signal to the JVM process produces a heap snapshot in *outputfile*.

4.2 The Cost of Delegation

The `Employee` class is not a very realistic example, since it has only primitive fields. Usually, a class has fields that are objects. In Java, an object field is implemented as a *delegated* object, that is, as a separate object pointed to by the owner object. On 32-bit architectures, a pointer is 4 bytes. You can estimate the size of an object using the estimate rules in Section 4.1, plugging in 4 bytes for each object field. Unless otherwise stated, the following examples assume the Sun JVM, and use the minimum-size estimate rule.

Example 4 (Employee class with object fields). Here is a more realistic `Employee` class, with several object 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 Employee {
    int hoursPerWeek;
    String name;
    BigDecimal salary;
    Date startDate;
    boolean exempt;
    char jobCode;
    int yearsOfService;
}
```

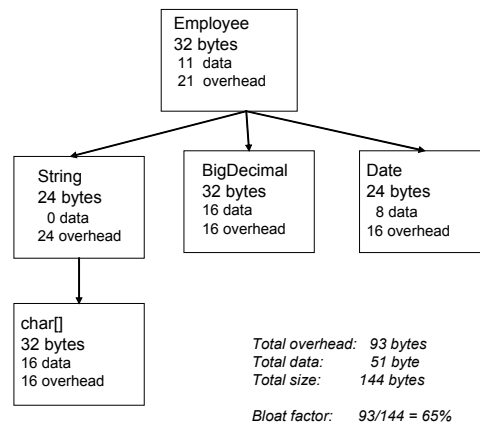


Figure 4.1: The memory layout for an employee “John Doe”

To estimate the size of this **Employee** object, add the field sizes $4+4+4+4+1+2+4$ ($=23$), add in 8 bytes for the header ($=31$), and then round this up to 32 bytes.

While an instance of the **Employee** class, is a single 32 byte object, an instance of an entire employee entity, including name, salary, and start date consists of five objects. Two of these objects are used to store the name **String**. Recall from Section ?? 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 ??.

A comparison of this employee representation with the original **Employee** object in Section [?] shows that the size has increased from 32 to 144 bytes, and the bloat factor, the percentage of memory overhead, has increased from 46% to 65%. There is more information in the new version, so it is no surprise that it is bigger. The increase in the bloat factor is more significant. Delegation increases memory bloat. Delegation introduces additional object headers, a pointer for each delegated object, and empty pointer slots for uninitialized object fields. Delegation may also force additional alignment costs, since each new delegated object has to be aligned to an 8-byte boundary.

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

In Java, delegation is one of the costs you pay for object-oriented programming. 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 this add up quickly, and contribute significantly to large bloat factors in real applications.

4.3 Fine-Grained Data Models

The Java language makes it hard to avoid delegation. Programmer choices also impact delegation costs. The current 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 a lot of extra baggage that the subclass does not need. In languages with single inheritance, once you have used up your inheritance in your single inheritance hierarchy, it becomes hard to refactor your code. In this case, delegation is more flexible than inheritance for specializing types. However, it is possible to overuse delegation, resulting in an overly fine-grained data model with many small objects. Fine-grained data models can be expensive both in execution time and memory space.

There is no simple rule that can always be applied. Each situation has to be evaluated in context, and there may be tradeoffs among different goals. To make an informed decision, you need to understand the impact of delegation on memory requirements, by measuring the object sizes and overhead.

Example 5 (Employee emergency contact). An emergency contact is needed for each employee. An emergency contact is a person and a preferred method to reach her. The preferred method can be email, cell phone, work phone, or home phone. All contact information for the contact person must be stored, just in case the preferred way does not work in an actual emergency.

Here are class definitions for an emergency contact. Delegation is used heavily. This programming style is common in real applications. Often programmers are not aware of the overhead that delegation introduces.

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

class EmergencyContact {
    ContactPerson contact;
    Contact preferredContact;
}

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

class Contact {
    ContactPerson owner;
}

class PhoneNumber extends Contact {
    char[] phone = new char[10];
}

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

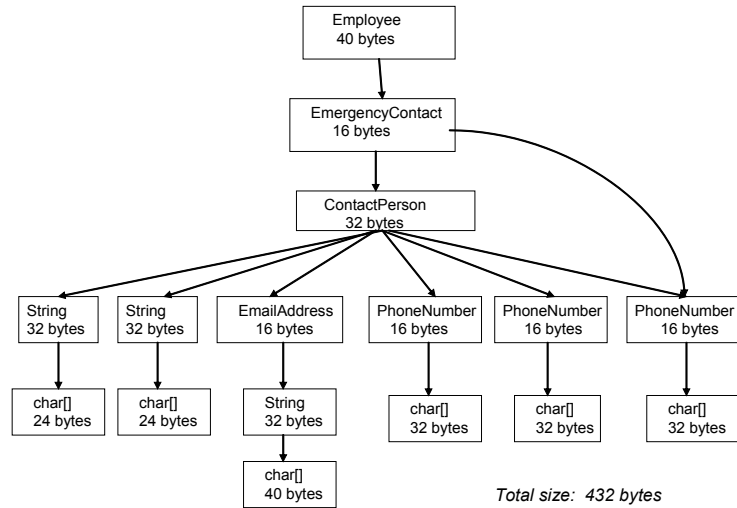



Figure 4.2: The memory layout for an employee with an emergency contact.

The memory layout for a sample employee is shown in Figure 4.2. There are 15 objects used to store emergency contact information. This seems excessive. The objects are all small, containing one or two meaningful fields, which is another sign of an overly fine grained data model. One object that looks superfluous is **EmergencyContact**, which encapsulates the contact person and the preferred contact method. Reversing this delegation involves simply inlining the two **EmergencyContact** fields into the parent **Employee** object, and then removing the **EmergencyContact** object altogether. Alternatively, it makes more sense to move the **ContactPerson** field to the **Employee** object, and the **preferredContact** field to the **ContactPerson**, since the preferred contact method is really an attribute of the contact person. Here is the modified code:

```

class Employee {
    ...
    ContactPerson contact;
}

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

```

This change eliminates an object, but not much space. You can save considerably more space by inlining the four **Contact** objects into the **ContactPerson** object, which results in a 15% space reduction. In a

system with many in-memory `Employees`, this reduction is significant. In order to make this change, the preferred contact method must be encoded somehow in `ContactPerson`. A simple way to do this is to use an enumeration type 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;
    char[] cellPhone;
    char[] homePhone;
    char[] workPhone;
}
```

Figure ?? shows the memory layout after these changes.

When many objects are used to represent a single entity, 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.

4.4 Large Base Classes

For cross-cutting features that are common to many objects, inheritance is an obvious implementation choice. A base class implements a feature, and all objects that require the feature are instances of subclasses of this base class.

Example 6 (Keeping track of updates). 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 that stores create and update information, taken from a real application.

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

It is now possible to track changes to any object by subclassing from `ModifyInfo`.

Returning to Example 3 in Section 4.3, suppose that updates to employees' emergency contacts need to be tracked. You have to be a bit careful here. If you decide to use the fine-grained data model, and decide to track changes to every contact phone number or email address, you may run out of memory. As shown in Figure ??, this adds an extra 16 bytes to each `Contact` object, and up to four additional `Date` and `Party` objects for each of the four `Contact` object. A far more scalable solution is to track changes to each `ContactPerson`, whether or not you stay with the fine-grained data model.

When highly-delegated objects are combined with large base classes, memory costs multiply quickly. you may end up with many big objects

The moral of this example is that if you have both a fine-grained design and large base classes, memory costs can multiply quickly. The resulting of fine-grained functionality may not be necessary, or even intended. For example, it is overkill to attach update information to every contact point for every emergency contact for every employee. It is probably sufficient to keep update information only for the `ContactPerson`.

It is very easy for a programmer to subclass without looking closely at the memory size of a superclass, especially if the inheritance chain is long. Also, programmers may not think about how many instances of an entity will be created at runtime. The only thing worse than too many small objects is too many big objects.

4.5 64-bit Architectures

In the 64-bit world, that string is now 96 bytes, and the reason is 50% larger than it was in 32-bit JVM. The reason is first of all so in this case, just the delegation costs is now 32 bytes, which is basically 1/3 of the cost, and the total cost of the string is up by 50%. In general, this is a real problem for fine-grained designs. Unfortunately, a lot of designs out there are fine-grained, and we'll see the reasons for that.

So just let's talk a little about what this means for 64-bit JVMs. People are busting out of this 32-bit address space now, and they are saying let's go to 64-bit, and that will solve everything. Unfortunately, when you go to 64-bit, things blow up pretty quickly. , the object headers are double, pretty sure that's the same thing in sun. arrays are not quite double. 24 bytes. The other thing is that all the pointers instead of being 4 bytes, are now 8 bytes; and finally, there's still that same alignment cost in J9. In SUN you will have a higher alignment cost, since in SUN now it's aligned to 8-byte boundary, where it was aligned to 4-byte boundary before. The studies I've read, both outside and inside IBM, are showing that going to a 64-bit JVM will increase your memory usage by 40-50%, and I've heard arguments that say that your application will run faster, because of the native architecture and all that, but in some cases there is evidence that it may run slower, because of the worse cache locality. It's another example where it's super important to measure, and not make assumptions that this will fix my speed problem, or it will fix my memory problem. There are a lot of surprises in this area. So J9, and some of the other JVMs, I don't think SUN has this yet, I could be wrong, I know JRocket has it , they have a mode have called compress addressing, where they can squeeze a few extra bits out of the 32-bit addresses, and in J9 6SR2 is going to allow addressing of up to 28 gig with a 32 bit address still, so without all of these additional footprint problems, if you use this compressing flag. From what I've read, it's only a few percentage degradation in performance, it's pretty minor, so it can certainly help people in this transition area.

4.6 Summary