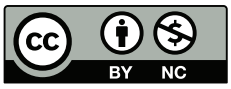
Java: Exercises in Style

Marco Faella

June 1, 2018

Author: Marco Faella, marfaella@gmail.com



Preface to the Online Draft

This draft is a partial preview of what may eventually become a book. The idea is to share it with the public and obtain feedback to improve its quality. So, if you read the whole draft, or maybe just a chapter, or you just glance over it long enough to spot a typo, let me know!

I'm interested in all comments and suggestions for improvement, including (natural) language issues, programming, book structure, etc.

I would like to thank all the students who have attended my Programming Languages class, during the past 11 years. They taught me that there is no limit to the number of ways you may think to have solved a Java assignment.

Naples, Italy
January 2018

Marco Faella

Contents

Contents **iv**

I Preliminaries **1**

1 Introduction **3**
1.1 Software Qualities 5

2 Problem Statement **9**
2.1 Data Model and Representations 10

3 Hello World! **13**

4 Reference Implementation **17**
4.1 Space Complexity 21
4.2 Time Complexity 23

II Software Qualities **29**

5 Need for Speed **31**
5.1 Optimizing “getAmount” and “addWater” 32
5.2 Optimizing “connectTo” and “addWater” 34
5.3 The Best Balance: Union-Find Algorithms 37
5.4 Experiments 43

6 Precious Memory **45**
6.1 Gently Squeezing 45
6.2 Plain Arrays 48
6.3 Forgoing Objects 50
6.4 The Black Hole 54

7 Reliability **57**
7.1 Designing Containers by Contract 59
7.2 Checking Contracts 63

7.3 Checking Invariants 68
7.4 Testing 72

8 Readability **85**
8.1 Readable Containers 86

9 Multi-Threading **93**
9.1 Global Locking 93
9.2 A Failed Attempt 95
9.3 Thread-safe Containers 95

10 Generality **99**
10.1 The General Framework 100
10.2 A Concrete Implementation 101

11 Golf Coding **103**
11.1 The Shortest I Came Up With 103

Index **107**

Part I

Preliminaries

Chapter 1

Introduction

In the modern classic *Exercises in Style*, Raymond Queneau writes the same story in 99 different ways. The point of the book is not the story, which is simple and unremarkable, but rather the whimsical exploration of the virtually endless expressive capabilities of natural languages. Programming is certainly not literature, despite efforts by renowed personalities such as Donald Knuth to bring the two closer together. In fact, beginner programmers may be forgiven if they think that that there is one best way to solve each programming assignment, just like simple math problems have a single solution. In reality, modern day programming can be much more similar to literature than to math. Programming languages have evolved to include ever more abstract constructions, multiplying the ways to achieve the same goal. Even after their introduction, programming languages evolve, often by acquiring new ways of doing the same things. The most used languages, such as Java, have been evolving at a higher and accelerated pace, to keep up with the younger languages trying to take their positions.

In this book I try to give a taste of the wide range of possibilities that one should consider, or at least be aware of, when solving a simple programming task. The task I propose is quite unremarkable: a class representing water containers, that can be connected with pipes and filled with liquid. Clients interact continuously with the containers, adding or removing water, and placing new pipes at any time.

I present and discuss 15 different implementations for this task. My examples are extreme: in each one I strive to focus on a single objective, be it performance, code clarity, or other. As a consequence, in most practical contexts, *none of my examples is right*. Life demands compromises, and programming is no exception. In real-world scenarios, the context dictates which compromise between different program qualities is the most reasonable practical objective to shoot at.

The book is not a dry sequence of code snippets. Whenever the context calls for it, I take the opportunity to introduce a number of specialized topics,

pertaining to computer science (complexity theory and amortized complexity), Java programming (thread synchronization and the Java memory model), or software engineering (the design-by-contract methodology and testing techniques).

Who is this Book for?

The core message of this book is intended for programmers of all trades and levels: programming is a complex creative endeavor, involving many different concurrent objectives. A good programmer is aware of them. A very good programmer can selectively optimize the system with respect to each one of them. A great programmer can effectively choose the best balance among them, based on the context.

The content of the book may not be appropriate for beginner programmers. This is neither a Java manual, nor an introductory programming textbook. To make the most out of this book, you should be familiar with basic object-oriented programming in Java. Even on more advanced topics, such as multi-threading, I will only cover what is needed to understand the specific usage done in the examples. At the end of each chapter, the "Further Reading" sections list various resources to learn or refresh your memory on specific topics.

Regarding general programming, you should be aware of recursion and the general tenets of OOP. Specifically regarding Java, besides the basics, you should have some exposure to the standard collections (sets and lists) and to basic multi-threading and low-level thread synchronization (`synchronized` blocks and methods).

The idea for writing this book came after 12 years of teaching an advanced undergraduate programming class. I hope that the result might be useful for similar classes, as a way to frame in the same context a number of topics that are usually spread out in different programming, algorithm, or software engineering courses.

Book Organization

After this introduction, the book presents the programming task that all subsequent chapters solve, each in its own way. A simple use case establishes a common public interface, that most implementations adhere to. The next chapter describes a reference implementation, that tries to strike a compromise between different qualities. Other versions are then compared to it.

All code snippets are correct, in the sense that they can be compiled and give rise to a program that respects the initial specifications, up to some point. The pros and cons of each version are discussed in each section.

All the code appearing in the book can be found online in a Git repository:

`https://bitbucket.org/mfaella/ExercisesInStyle`

Each version is identified by a tag, which corresponds to the Java file name in the repository. For instance, the reference implementation from Chapter 4 is denoted by REFERENCE, and its file name is `Reference.java`.

This is a *theory box*. In the rest of the book, it is used to include technical information that frames the content in the wider Computer Science picture. You can skip them with no adverse effects, except a missed opportunity to expand your understanding!

1.1 Software Qualities

As for all products, software qualities are ways to define the extent to which the system fulfills its purposes. They are traditionally divided into internal and external qualities. External qualities are those that can be perceived by the end user. Typical external qualities are:

- Efficiency — adequate consumption of resources, mostly time and space (memory).
- Correctness — adherence to stated objectives, a.k.a. requirements or specifications.
- Robustness — resilience to incorrect inputs or adverse/unanticipated external conditions (e.g., lack of some resource). Correctness and robustness are sometimes lumped together as *reliability*.
- Usability — a measure of the effort needed to learn how to use it and to achieve its goals; ease of use.

The most important internal qualities are:

- Readability — Clarity, understandability by fellow programmers.
- Maintainability — Ease of finding and fixing bugs, as well as evolve the software.
- Testability — Ability and ease of writing tests that can trigger all relevant program behaviors and observe their effects.

The boundary between the two categories is not clear-cut. Most internal qualities can be indirectly perceived by the end user. Maintainability is mostly internal, but end users will be made aware of it if a defect is found and it takes programmers a long time to fix it. Conversely, robustness can also be internal, if the piece of software under consideration is not exposed to the end user, but only interacts with other system modules.

readability				
robustness	-			
space efficiency	↓	-		
time efficiency	↓	↓	↓	
	readability	robustness	space efficiency	time efficiency

Table 1.1: Typical interactions between code qualities: ↓ stands for "hurts" and - for "no interaction". Inspired by Figure 20-1 in *Code Complete*.

Some software qualities represent contrasting objectives, while others go hand-in-hand. The result is a balancing act common to all engineering specialties. Table 1.1 summarizes the relationships between four of the qualities that we examine in this book.

Both time and space efficiency usually hinder readability. Seeking maximum performance leads to sacrificing abstraction and writing lower level code. In Java, this may entail using primitive types instead of objects, or in extreme cases writing performance-critical parts in a lower-level language like C and connect them with the main program using the Java Native Interface.

Minimizing memory requirements also favors the use of primitive types, as well as special encodings, where a single value is used as a compact way to represent different things (we'll see an example of this in Section 6.4). All these techniques tend to hurt readability. Conversely, readable code uses more temporary variables and support methods, and shies away from low-level performance hacks mentioned above.

Time and space efficiency also conflict with each other. For instance, a common strategy for improving performance consists in storing extra information in memory, instead of computing it every time it is requested. A prominent example is the difference between singly and doubly linked lists. Even though the "back" link of every node could in principle be computed by scanning the list, storing and maintaining those links allows for constant-time deletion of arbitrary nodes. The class in Section 6.4 trades improved space efficiency for increased running time.

Maximizing robustness requires adding code that checks for abnormal circumstances and reacts in the proper way. Such checks incur a performance overhead, albeit usually quite limited. Space efficiency need not be impacted in any way. Similarly, in principle there is no reason why robust code should be less readable.

In this book, we focus on those software qualities that remain meaningful when applied to a small software unit consisting of a single class. Moreover, we will mostly stick to a fixed public interface and only change the internal implementation of the class. In detail, here is a break down of the chapters and their corresponding code qualities.

Chapter 2 The description of the programming task to be solved.

Chapter 3 A naive implementation, as conceived by an inexperienced programmer. To break the ice and have some fun.

Chapter 4 The reference implementation (in the following, REFERENCE).

Chapter 5 Time efficiency. We improve the running time of REFERENCE by up to two orders of magnitude (500x), and discover that different use cases lead to different performance trade-offs.

Chapter 6 Space (memory) efficiency. Compared to REFERENCE, we shrink the memory footprint of containers by more than 50% when using objects and by 90% when forgoing an object for each container.

Chapter 7 Reliability, comprising correctness and robustness. Robustness is intended in its internal meaning: how gracefully a piece of software reacts to being invoked in the wrong way. We discuss the *design by contract* methodology and apply the related coding techniques: runtime checks, assertions, and unit tests.

Chapter 8 Readability. Starting from REFERENCE, we follow the best-practices for clean self-documenting code.

Chapter 9 Thread-safety. This is not a general code quality, but rather the specific ability of a class and its objects to simultaneously interact with different threads.

Chapter 11 Succinctness, i.e., writing the shortest possible code for the given task. This is not a code quality at all. On the contrary, it leads to horrible, obscure code. It is included as a fun exercise that pushes the language to its limits.

Further Reading

- R. Queneau. *Exercises in Style*. New Directions, 2013

  The original "exercises in style" book (actually, the original was written in French in 1947).

- C. Videira Lopes. *Exercises in Programming Style*. CRC, 2014

  Very related to the book you are reading: the author solves a simple programming task in 33 different styles, using Python. Rather than optimizing different code qualities, every style results from obeying certain *constraints*. Among other things, it teaches you a lot about the history of programming languages.

- D.E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1995

- S. McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004

  A precious book on coding style and all-round good software. Among many other things, it discusses about code qualities and their interactions.

- S. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, 2008

  It includes a detailed chapter on software qualities and the related international standards.

Chapter 2

Problem Statement

This section describes the programming problem that all examples solve, each in its own peculiar way. We want to model a set of *water containers*. All containers are identical and have a virtually unlimited capacity. The state of a container is described by the amount of liquid it contains. Morerover, two containers can be connected by *pipes*. Whenever two containers are connected, they become communicating vessels and therefore split equally the liquid contained in them.

Therefore, at the very least we require a Container class, with the following three methods:

1. public double getAmount()
   Returns the amount of water in this container.

2. public void addWater(double amount)
   Pour amount quantity of water into this container. Water is automatically and equally distributed among all containers that are connected, directly or indirectly, to this one.

   This method can also be used with a negative amount, to remove water from this container. In that case, there should be enough water in the group of connected containers to satisfy the request (we wouldn't want to leave a negative amount of water in a container).

3. public void connectTo(Container other)
   Connects this container with other, using a (permanent) pipe.

The next code fragment shows a simple use case for the Container class.

UseCase: Part 1

```
Container a = new Container();
Container b = new Container();
Container c = new Container();
Container d = new Container();
```

```
a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println (a.getAmount()+" "+b.getAmount()+" "+
                    c.getAmount()+" "+d.getAmount());
```

At the end of this fragment, containers a and b are connected, so they share the water that was put into a, whereas containers c and d are isolated. So, the desired output from the println is:

```
6.0 6.0 0.0 8.0
```

Then, let's continue the use case with the following lines.

```
b.connectTo(c);                                          UseCase: Part 2
System.out.println (a.getAmount()+" "+b.getAmount()+" "+
                    c.getAmount()+" "+d.getAmount());
```

The first line above connects c to b and, indirectly, to a. So, a, b, and c are now communicating vessels and the total amount of water contained in all of them distributes equally among them. Container d is unaffected, leading to the output:

```
4.0 4.0 4.0 8.0
```

This point in the use case is what we will use to show the memory layout of all the implementations in the following chapters.

Finally, we connect d to b, and all containers form a single connected group:

```
b.connectTo(d);                                          UseCase: Part 3
System.out.println (a.getAmount()+" "+b.getAmount()+" "+
                    c.getAmount()+" "+d.getAmount());
```

In the final output the water level is equal in all containers.

```
5.0 5.0 5.0 5.0
```

In this use case, we first inserted water and then we started connecting containers. In general, those two operations may be freely interleaved. What's more, new containers can be created at any time.

2.1 Data Model and Representations

Before we start discussing various possible implementations for our problem, let us pause and make some preliminary observations. What fields should our Container class include? This is often the most important decision when

designing any class. The examples in this book show that many different answers can be given, based on the specific objectives we aim at.

Irrespective of the objectives, we should include enough information to offer the services required by our contracts. This much is clear. Once this basic criterion is met, we still have two types of decision to make:

1. Do we store any extra information, even if not strictly necessary?

2. How do we encode all the information we want to store? Which data types or structures shall we use?

Regarding question 1, we may want to store unnecessary information for two possible reasons. First, we may do it for performance; this is the case of information that could be derived from other fields, but we prefer to have it ready. Secondly, we sometimes store extra information to make room for future extensions.

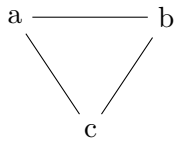
In our case, it seems clear that containers should know the amount of water in them. Moreover, when adding water to a container, the liquid must be distributed equally over all containers that are connected (directly or indirectly) to this one. So, each container must be able to identify all the containers that are connected to it. An important decision is whether to distinguish direct from indirect connections. A direct connection between `a` and `b` can only be established via the call `a.connectTo(b)` or `b.connectTo(a)`, whereas an indirect connection may arise as a consequence of direct ones. In mathematical terms, indirect connections correspond to the *transitive closure* of direct ones.

The operations required by our use case do not distinguish these two types of connections, so we could do without the distinction and just store the more general type: indirect connections. However, suppose that at some point in the future we want to add a "disconnectFrom" operation, whose intent is to undo a previous "connectTo" operation. If we mix up direct and indirect connections, we cannot hope to correctly implement "disconnectFrom".

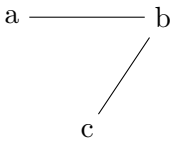
Indeed, consider the two scenarios represented in Figure 2.1, where direct connections are drawn as lines between containers, and water levels are not shown. If only indirect connections are stored in memory, the two scenarios are indistinguishable: in both cases, all containers are mutually connected. Hence, if the same sequence of operations is applied to both scenarios, they are bound to react in the exact same way. On the other hand, consider what *should* happen if the client issues the following operations:

```
a.disconnectFrom(b);
a.addWater(1);
```

In the first scenario (Fig. 2.1A) the three containers are still mutually connected, so the extra water must be split equally in all of them. Conversely, in the second scenario (Fig. 2.1B) container `a` becomes isolated, so the extra



(A) Three mutually connected containers



(B) Containers a and c are only indirectly connected

Figure 2.1: Two 3-container scenarios. Lines between containers represent direct connections.

water must be added to a only. Clearly, storing indirect connections only is incompatible with a "disconnectFrom" operation.

So, if we think that the future addition of a "disconnectFrom" operation is likely, we may have reason to store direct connections explicitly and separately from indirect ones. However, if we do not have specific information about the future direction of our software, we should be wary of such temptations. Programmers are known to be prone to over-generalization, and tend to weigh the hypothetical benefits more than the certain costs that come with it. Notice that costs are not limited to development time, as each unnecessary feature needs to be tested, documented, and maintained just like the necessary ones.

Also, there is no limit to the amount of extra information one may want to include. What if we want to remove all connections older than 1 hour? We should store the time when each connection was made! What if we want to know how many threads have created connections? We should store the set of threads that have ever created a connection, and so on.

In the following chapters, we will generally stick to only storing the information that is necessary for our present purposes, with a few clearly marked exceptions.

In graph-theoretic terms, our containers can be seen as nodes in an *undirected graph*, whose edges are the connections established by the connectTo method. The graph is undirected because our connections are symmetric. Proper water distribution requires connected components to be known. Summarizing, the overall setting can be seen as an example of maintenance of transitive closure with edge insertions, a type of *dynamic graph connectivity problem*.

Chapter 3

Hello World!

This chapter presents a simple implementation, that could be authored by a novice programmer, who's just picked up Java after some exposure to a structured language like C. This implementation compiles and works as expected, but it contains several shortcomings, that I will comment right after each code snippet.

Let's start with the class fields and constructor.

NOVICE

```
public class Container {

    Container[] g;
    int n;
    double x;

    public Container() {
        g = new Container[1000];
        g[0] = this;
        n = 1;
        x = 0;
    }
```

The intent is the following: g is the array of all containers connected to this one, including this one (as clear from the constructor); n is the number of containers in g; x is the amount of liquid in this container.

These few lines contain a wealth of small and not-so-small defects. Let's focus on the ones that are superficial and simple to fix, as the others will become apparent when we move to better versions in the next chapters.

The single quirk that immediately marks the code as amateurish is the choice of variable names. They are too short and completely uninformative. A pro wouldn't call the group g if a mobster gave him 60 seconds to hack into a super-secure system of water containers. Meaningful naming is the first rule of readable code, as we'll see in Chapter 8.

Then, we have the visibility issue. Fields should be private instead of *default*. Recall that default visibility is more open than private: it allows access

from other classes that reside in the same package. Information hiding (a.k.a. encapsulation) is a fundamental OO principle, enabling classes to (a) ignore the internals of other classes and interact with them via a well-defined public interface (a form of separation of concerns), and (b) change their internal representation without affecting existing clients.

Encapsulation provides footing to this book too. The many implementations presented in the following chapters comply with the same public API, and therefore can in principle be used interchangeably by clients.

Moving along, the array size is defined by a so-called *magic number*, i.e., a constant that is not given any name. Best practices dictate that all constants should be assigned to some `final` variable, so that: (a) the variable name can document the meaning of the constant, and (b) there is a single point where the value of that constant is set, which is especially useful if the constant is used multiple times.

The very choice of using a plain array is not very appropriate, as it puts an *a-priori* bound to the maximum number of connected containers: too small a bound and the program is bound to fail; too large is just wasted space.

Moreover, using an array forces us to manually keep track of the number of containers actually in the group (field `n` here). Better options exist in the Java API and are discussed in Chapter 4. Nevertheless, plain arrays will come handy in Chapter 6, where our primary objective is to save space.

Let's proceed to the source for the first two methods.

NOVICE

```
public double getAmount() { return x; }

public void addWater(double x) {
    double y = x / n;
    for (int i=0; i<n; i++)
        g[i].x = g[i].x + y;
}
```

`getAmount` is a trivial getter and `addWater` shows the usual naming problems with variables `x` and `y`, whereas `i` is acceptable as the traditional name for an array index. If the last line used the += operator we wouldn't have to look back and forth to make sure that it is actually incrementing the same variable.

Notice that `addWater` does not check whether its argument is negative and, in that case, whether there is enough water in the group to satisfy the request. Robustness issues like this one will be dealt with specifically in Chapter 7.

The `connectTo` method is where the naming issues hurt the most.

NOVICE

```
public void connectTo(Container c) {
    if (g==c.g) return;
    double z = (x*n + c.x*c.n) / (n + c.n);

    for (int i=0; i<c.n; i++)
```

```
            g[n+i] = c.g[i];
        n += c.n;

        // Each container that is connected with c belongs to the new group
        for (int i=0; i<c.n; i++) {
            c.g[i].g = g;
            c.g[i].n = n;
        }

        // Update amount
        for (int i=0; i<n; i++) {
            g[i].n = n;
            g[i].x = z;
        }
    }
```

All those single letter names make it really hard to understand what's going on. Besides, the method is too long and should be split. For a dramatic comparison, you may want to compare it with the readability-optimized version on page 87.

In the next chapter, we are going to present a reference implementation, that solves most of the superficial issues noted here, while striking a balance between different code qualities.

Chapter 4

Reference Implementation

This chapter presents a version of the Container class that tries to strike a good balance between different qualities, such as clarity, efficiency, and memory usage.

As discussed in Section 2.1, we store and maintain the set of indirect connections between containers. In practice, we equip each container with a reference to the set of containers directly or indirectly connected to it, called its *group*. We try to exploit the Java Collection Framework as much as possible. For starters, the container group is represented by a Set<Container>. Set is an interface used for unordered collections with no duplicates, which fits our purposes nicely. Initially, the group consists of this object only. Additionally, each container is aware of the amount of water in it, encoded by a double value.

REFERENCE

```
import java.util.*;

/* A water container.
 *
 * by Marco Faella
 */
public class Container {

    private Set<Container> group;
    private double amount;

    /* Creates an empty container. */
    public Container() {
        group = new HashSet<Container>();
        group.add(this);
    }
```

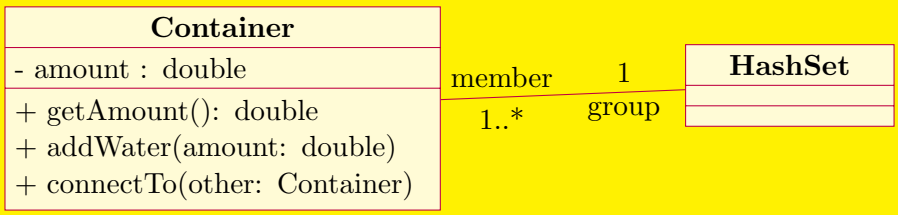
Figure 4.1 shows the memory layout of REFERENCE, after executing USE-CASE, Part 2. At that point, three of the four containers are connected in a group, and the fourth one is isolated. The memory layout diagram is a

simplified scheme of how the objects are arranged in memory, similar to UML object diagrams. Both display static snapshots of a set of objects, including the value of their fields and their relationships.

Unified Modeling Language (UML) is a standard providing a rich collection of diagrams, intended to describe various aspects of a software system. Class diagrams and sequence diagrams are two of the most commonly used parts of the standard.

A class diagram is a description of the static properties of a set of classes, particularly regarding their mutual relationships, such as inheritance or containment. The above mentioned object diagrams are closely related to class diagrams, except that they depict individual instances of those classes.

For instance, the class diagram for REFERENCE may look like this:



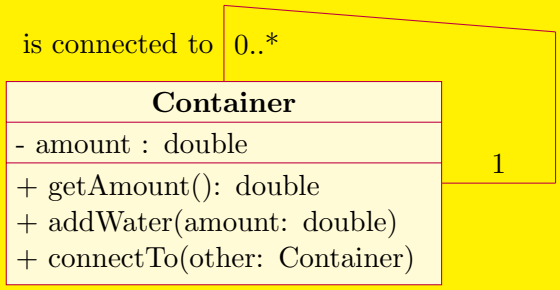
The Container box is quite self-explanatory, listing fields and methods, whose visibility is denoted by a plus (public) or minus (private) sign. The HashSet box does not specify any field or method, and that is perfectly fine for such diagrams: they can be as abstract or as detailed as you wish.

The line between the two boxes is called an association and represents a relation between two classes. At each end of the line, we can describe the role of each class in the association (member, group) and the so-called cardinality of the association. The latter specifies how many instances of that class are in relation to each instance of the other class. In our case, each Container belongs to a single group and each group includes one or more members, denoted in UML by " 1..* ".

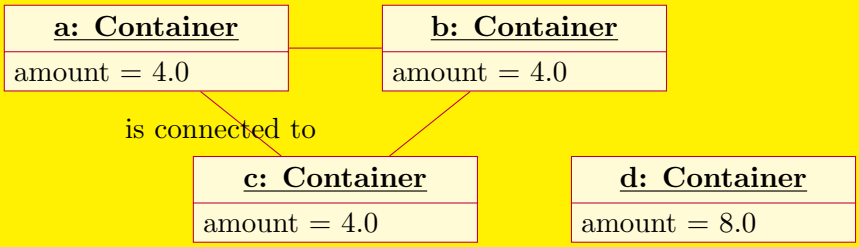
Although formally correct, the class diagram above is too detailed for most purposes. UML diagrams are intended to describe a *model* of the system, not the system itself. If a diagram becomes too detailed, it may as well be replaced by the actual source code. Hence, standard collections such as HashSet are normally not explicitly mentioned. Rather, they are interpreted as just one possible implementation of an association between classes.

In our case, the HashSet can be replaced by a more abstract association linking the Container class with itself. In this way, rather than describing the implementation, we are conveying the idea that each con-

tainer may be connected to zero or more other containers. This can be represented graphically as follows.



UML object diagrams appear very similar to class diagrams. For instance, here is the object diagram for REFERENCE, after executing USE-CASE up to its Part 2.



The object diagram above follows the second class diagram, where the HashSet's are not explicitly modeled, but rather hidden within the association between containers. Objects are distinguished from classes by having their names and types underlined.

In the following of this book, instead, I'm going to use a custom, more intuitive form of object diagram.

In the memory layout diagrams in this book, many details, such as the object header, are omitted. The internal composition of the HashSet is completely hidden, as the focus is on which object contains each piece of information, and which object points to which other object.

Going back to the code, compared to NOVICE, this version uses proper encapsulation and field naming. The getAmount method is a trivial getter.

REFERENCE

```
/* Returns the amount of water held in this container */
public double getAmount() { return amount; }
```

Next, let us analyze the connectTo method, displayed below. Start by observing that connecting two containers essentially entails merging their two groups. So, the method initially computes the total amount of water in the two groups and the amount of water in each container after the merge. Then,

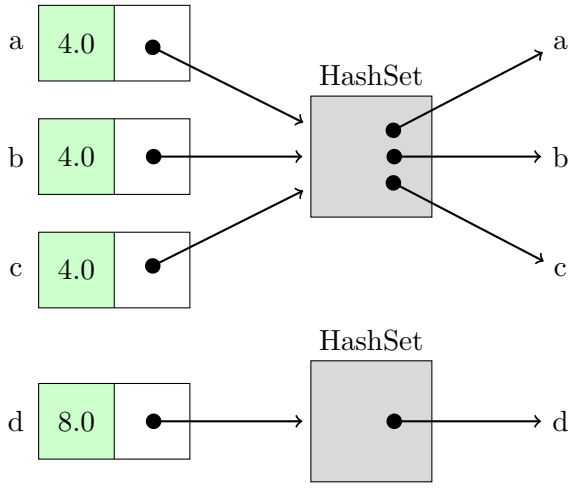


Figure 4.1: Memory layout of REFERENCE after executing USECASE, Part 2. To avoid the clutter, the references from the HashSet's back to the containers are pictured as reaching the name of the container.

the group of this container is modified to absorb the second group, and all containers of the second group are assigned the new, larger group. Finally, the amount of water in each container is updated with the pre-computed new amount.

REFERENCE

```
/* Connects this container with other. */
public void connectTo(Container other) {

    // If they are already connected, do nothing
    if (group==other.group) return;

    int size1 = group.size(),
        size2 = other.group.size();
    double tot1 = amount * size1,
           tot2 = other.amount * size2,
           newAmount = (tot1 + tot2) / (size1 + size2);

    // Merge the two groups
    group.addAll(other.group);
    // Update group of containers connected with other
    for (Container c: other.group) c.group = group;
    // Update amount of all newly connected containers
    for (Container c: group) c.amount = newAmount;
}
```

The method above is heavily commented in an attempt to improve its readability. The modern trend, instead, would be to split it in smaller methods and use descriptive names, as we show in Chapter 8.

The addWater method simply distributes an equal amount of water to each container in the group.

REFERENCE

```
/* Adds water to this container. */
public void addWater(double amount) {
    double amountPerContainer = amount / group.size();
    for (Container c: group) c.amount += amountPerContainer;
}
```

As in NOVICE, the method above accepts all negative values, thus running the risk of leaving a negative amount of water in one or more containers. This sort of robustness issues are addressed in Chapter 7. In the next two sections, we are going to analyze the memory and time consumption of the above implementation, so as to compare it with those of the following chapters.

4.1 Space Complexity

Despite the fact that primitive types have a fixed size, estimating the memory size of a Java object is not trivial. Three factors render the exact size of an object architecture and implementation-dependent: reference size, object headers, and padding.

How these factors influence the size of an object depends on the specific JVM used to run your program. Recall that the Java framework is based on two official specifications: one for the Java language and one for the virtual machine. Different vendors are free to implement their own compiler and/or VM, and indeed as of today Wikipedia lists 17 actively developed JVMs. In the following VM-dependent arguments, we are going to refer to Oracle's standard JVM, called *HotSpot*.

Let's consider each of the above factors in more detail. First, the size of a reference is not fixed by the language. Whereas this size is 32 bits on 32-bit hardware, on modern 64-bit processors it can be either 32 or 64 bits, due to a technology called *Compressed ordinary object pointers (OOPs)*. Compressed OOPs allow the program to store references as 32-bit values, even when the hardware supports 64-bit addresses, at the cost of using up to 32GB of the total available heap space. In our estimates, we assume a fixed reference size of 32 bits.

Compressed OOPs work by implicitly adding 3 zeros at the end of each 32-bit address, so that a stored address of, say, `0x1` is interpreted as the machine address `0x1000`. In this way, machine addresses effectively span 35 bits, and the program can access up to 32GB of memory. The JVM must also take steps to align all variables to 8-byte boundaries, as the program can only refer to addresses that are multiples of 8.

Summarizing, this technology saves space for each reference but in-

creases padding space and incurs a time overhead when mapping stored addresses to machine addresses (a quick left shift operation). Compressed OOPs are turned on by default, and can be turned off with a JVM command-line option.

Second, the memory layout of all objects starts with a header containing some standard information needed by the JVM. As a consequence, even an object with no fields (a.k.a. a *stateless* object) takes up some memory.

The detailed composition of the object header goes beyond the scope of this book [1], but three features of the Java language are mainly responsible for it: reflection, multi-threading, and garbage collection.

1. Reflection requires objects to know their type. Hence, each object must store a reference to its class, or a numeric identifier referring to a table of loaded classes. This mechanism allows the instanceof operator to check the dynamic type of an object and the getClass method of the Object class to return a reference to the (dynamic) class of this object.

2. Multi-threading support assigns a *monitor* to each object (accessible via the synchronized keyword). Hence, the header must accommodate a reference to a monitor object. Modern virtual machines create such monitor on demand, only when multiple threads actually compete for exclusive access to that object.

3. Garbage collection needs to store some information on each object, such as a *reference count*. In fact, modern garbage collection algorithms assign objects to different *generations*, based on the time since they were created. In that case, the header also contains an *age* field.

In this book, we will assume a fixed 12-byte per-object overhead, which is typical of modern 64-bit JVMs.

Finally, hardware architectures require or prefer data to be aligned to certain boundaries, i.e., they work more efficiently if memory accesses employ addresses that are multiples of some power of 2 (usually 4 or 8). This circumstance leads to compilers and virtual machines to employ *padding*: inflating the memory layout of an object with empty space, so that each field is properly aligned and the whole object fits exactly into an integer number of words. For simplicity, in this book we will ignore such architecture-dependent padding issues.

We can now turn our attention to the actual memory occupancy of the reference implementation. For starters, each Container object requires 12 bytes for overhead, 8 bytes for the amount field, 4 bytes for the reference to the set, plus the size of the set itself.

[1] If you're curious for details, you can browse the source code for HotSpot. The header's content is described in the file `src/share/vm/oops/markOop.hpp`.

A HashSet is typically implemented by an array of linked lists (called *buckets*), plus a couple of extra fields for bookkeeping. The size of the array is proportional to the number of elements in the set, and the lists tend to be very short, ideally at most one element long. Summarizing, an empty HashSet takes up approximately 64 bytes, and each extra element in the set requires one reference (to the list) and a list with one element: approximately 28 more bytes.

To get actual numbers and ease comparisons with other implementations, we will consider two scenarios: first, 1000 isolated containers; second, 1000 containers connected in 100 groups of ten containers each. In those two scenarios, our reference implementation performs as reported in Table 4.1.

scenario		bytes
1000 isolated	$1000 \cdot (12 + 8 + 4 + 64 + 28)$	116 000
100 groups of 10	$1000 \cdot (12 + 8 + 4) + 100 \cdot (64 + 10 \cdot 28)$	58 400

Table 4.1: Memory requirements of Reference.

4.2 Time Complexity

When measuring the memory footprint of a program, we can use bytes as the standard basic unit. If we ignore certain low-level details, such as word size and memory alignment issues, a given Java program will take the same amount of memory on all computers.

The situation for time measurements is more fishy. The same program performs in vastly different ways on different computers. Rather than measuring actual running time, we can then count the number of *basic steps* performed by the program. Roughly speaking, we can define as basic step any operation that requires a constant amount of time. For instance, any arithmetical or comparison operation can be considered a basic step [2].

The second issue we need to face is the fact that the same function can execute a different number of basic steps when given different inputs. For instance, the connectTo method above contains two for loops, whose length (i.e., number of iterations) depends on the size of the two container groups being merged. Incidentally, notice how in OOP the current state of this object is part of the input to an instance method. This is not surprising, considering that this is an effective (albeit hidden) parameter for these methods.

In such cases, we summarize with one or more numeric parameters what it is in the input that influences the running time of our algorithm. Usually,

[2]The formal definition of basic step must be based on a formal model of computation, such as Turing machines. A basic step may then be defined as any operation that requires a constant number of steps of a Turing machine.

the summary consists in measuring the *size* of the input in some way. If the number of basic steps of our algorithm varies even for same-sized inputs, we just consider the worst case, i.e., the maximum number of steps performed on any input of a given size.

Going back to our `connectTo` method, as a first attempt we may consider two parameters `size1` and `size2`: the sizes of the two groups of containers being merged. Using these parameters, we may analyse the `connectTo` method as follows:

```
public void connectTo(Container other) {                    | REFERENCE
    // If they are already connected, do nothing
    if (group==other.group) return;                          | 1 step
    int size1 = group.size(),
        size2 = other.group.size();
    double tot1 = amount * size1,
           tot2 = other.amount * size2,
           newAmount = (tot1 + tot2) / (size1 + size2);      | 5 steps
    // Merge the two groups
    group.addAll(other.group);                               | size2 steps
    // Update group of containers connected with other
    for (Container c: other.group) c.group = group;          | size2 steps
    // Update amount of all newly connected containers
    for (Container c: group) c.amount = newAmount;           | size1 + size2 steps
}
```

I'm counting as one step anything that does not involve a loop, because its running time is going to be essentially constant, and in particular independent of the parameters `size1` and `size2`. According to the above reasoning, the number of basic steps performed by connectTo is

$$6 + 2 * \texttt{size2} + (\texttt{size1} + \texttt{size2}) = 6 + \texttt{size1} + 3 * \texttt{size2}. \tag{4.1}$$

However, we should recognize that the number 6 in the above expression is somewhat arbitrary. If we counted assembly lines instead of Java lines, we might get 6 thousand instead of 6, and 6 million steps if we counted steps of a Turing machine. For the same reason the 3 multiplier in front of `size2` is essentially arbitrary. In other words, constants 3 and 6 depend on the *granularity* we choose for our basic steps.

A more interesting way to count steps, that elegantly sidesteps the granularity issue, is to only focus on how quickly the number of steps grows when the size parameters grow. This is called the *order of growth* and it is the basic tenet of complexity theory, a branch of Computer Science. The order of growth frees us from the burden of establishing a specific granularity for the basic steps, thus providing performance estimates that are more abstract but easier to compare to one another. At the same time, the order of growth preserves the *asymptotic* behavior of our function: i.e., the trend for large values of its parameter(s).

In practice, the most common way to indicate the order of growth is the so-called *big-O notation*. For instance, the expression (4.1) above in big-O notation becomes $O(\texttt{size1} + \texttt{size2})$, effectively hiding all arbitrary additive

and multiplicative constants. In so doing, it highlights the fact that the number of steps is lineraly proportional to `size1` and `size2`. More precisely, the big-O notation establishes an *upper bound* to the growth of a function. So, $O(\texttt{size1} + \texttt{size2})$ asserts that our running time grows *at most* linearly with respect to `size1` and `size2`.

Before we delve a little deeper into the asymptotic notation, let us further simplify our complexity analysis, by switching from two size parameters to a single one. Call n the total number of containers ever created, clearly $\texttt{size1} + \texttt{size2}$ is at most equal to n (distinct groups are disjoint by definition). Since the upper bound $O(\texttt{size1} + \texttt{size2})$ holds for our function, so does $O(n)$, which is greater than the first. In words, we are saying that the time required by a connectTo operation grows at most linearly with the total number of containers around. This may seem like a brutal approximation, and it is. After all, `size1` and `size2` are likely to be much smaller than n. However, rough as it is, this type of upper bound is going to be accurate enough to distinguish the efficiency of the various implementations presented in the following chapters.

When someone says that an algorithm has complexity $O(f(n))$, for some function f, they mean that $f(n)$ is an upper bound to the number of basic steps performed by the algorithm on inputs of size n. Clearly, this makes sense if we agree on how to measure the size of the input with a single parameter n. The following table presents some common big-O bounds, their names, and examples of array algorithms matching that bound. For algorithms running on arrays, the parameter n is generally understood to refer to the size of the array.

Notation	Name	Example
$O(1)$	constant time	checking whether the first element in an array is zero
$O(\log n)$	logarithmic time	binary search: the smart way to look for a specific value in a sorted array
$O(n)$	linear time	finding the maximum value in an unsorted array
$O(n \log n)$	quasilinear time	sorting an array using merge sort
$O(n^2)$	quadratic time	sorting an array using bubble sort

method	time complexity
getAmount	$O(1)$
connect	$O(n)$
addWater	$O(n)$

Table 4.2: Time complexities for REFERENCE.

More formally, the big-O notation can be applied to any function $f(n)$, representing the number of steps required by some algorithm when run on an input of size n. Consider an algorithm and let $g(n)$ be the actual number of "steps", however these may be defined, performed by the algorithm on an input of size n. Then, writing that the algorithm has time complexity $O(f(n))$ means that there exist two numbers m and c such that, for all $n \geq m$,

$$g(n) \leq c \cdot f(n) .$$

In other words, for sufficiently large inputs, the actual number of steps is at most equal to a constant times the value of the f function.

Complexity theory includes several other notations, denoting lower bounds, simultaneous lower and upper bounds, and so on.

We can now precisely state the time complexity of all the methods from REFERENCE. The getAmount method takes constant time, while connectTo and addWater need to cycle over all containers in a group. Since a group can be as large as the whole set of all containers, their complexity in the worst case is linear in the total number n of containers. Table 4.2 summarizes these observations. Chapter 5 is devoted to trying to improve these time complexities.

Further Reading

There are a thousand introductory books on Java programming. My favorites are:

- C.S. Horstmann. *Core Java.* Prentice Hall, 2015

  A two-volume behemoth, which covers many parts of the API in detail, with a strong teaching emphasis.

- P. Sestoft. *Java Precisely.* MIT Press, 3rd edition, 2016

Not an actual introductory book, but rather a concise and comprehensive reference guide to the language and a limited selection of the API (including collections and Java 8 streams).

Regarding time complexity and the big-O notation, any introductory book on algorithms features comprehensive explanations on the topic. The classical one is the following.

- T.H. Cormen, C.E. Leicerson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009

Finally, for UML and related software engineering techniques:

- Martin Fowler. *UML Distilled*. Addison-Wesley, 2003

  As its name suggests, this book condenses in fewer than 200 pages a solid introduction to UML notation, with special focus on class and sequence diagrams.

- Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2004

  Much wider in scope and page count than Fowler's book, this volume goes way beyond UML and serves as a systematic introduction to OO analysis and design. The second edition is also available as a free download.

Part II

Software Qualities

Chapter 5

Need for Speed

Achieving the maximum possible speed for a given computational task has fascinated programmers since the ancient times of punch-card programming. Indeed, one may say that a large part of computer science itself was born to meet this urge. In this chapter, I will present three different implementations, that optimize speed in different ways. Why three? Can't I just present you the *best* one? The thing is, there is no single best version, and that is perhaps the main takeaway from this chapter.

If you mostly studied computer science in school and have little programming experience, this may come as a surprise. Computer science curricula deal extensively with time efficiency, particularly in algorithm and data structure classes. Those classes and their textbooks focus on a single problem at a time, be it visiting a graph or balancing a tree. When you consider a single algorithmic problem, with given inputs and desired outputs, you can compare any two algorithms for performance, and possibly find the fastest possible procedure, as the one with the least asymptotic worst-case time complexity. This is indeed how research makes progress on single computational questions.

On the other hand, real-world programming and our little container example are not like that. They do not accept an input, compute an output and then terminate. They ask of us to design a number of interacting methods or functionalities, that may be used repeatedly any number of times. Different data structures may favor one method over another, improving the complexity of the first and slowing down the latter. For this reason, often there is no all-the-way best solution, just different trade-offs.

In a multi-method context like ours, worst-case time complexity induces a *partial order* between implementations. A partial order is a binary relation such that not every pair of items is comparable. For instance, consider the binary relation "being descendant from", applied to pairs of people. A pair like (Mike, Anna) belongs to the relation if Mike descends from Anna. If two people a and b are unrelated, neither the pair (a, b) nor

the pair (b, a) belongs to the relation, which means that the relation is a partial order. In a partial order, there might be items that are *not smaller* than any other. They are top items. Economists call these items Pareto optimal, and Pareto front the set of all those items. If we interpret "being descendant from" as "being smaller", the mythical Adam and Eve would be the only top elements, as they are not smaller than (i.e., descendant from) any other person.

If we don't have specific information on how many times and in what sequence each one of our methods will be invoked, the best we can do is to pick a Pareto optimal implementation. In such an implementation, no method can be improved without degrading the performance of another method. This chapter presents three Pareto optimal implementations for our problem.

5.1 Optimizing "getAmount" and "addWater" [SPEED1]

First, we are going to optimize the addWater method, bringing down its time complexity from linear to constant. The good news is that we can do this without increasing the complexity of the other two methods in the class.

In REFERENCE, the problem with addWater is that it needs to visit all containers that are connected to this one, and update their water amount. This is a waste, especially because *all connected containers share the same amount*. So, we move the amount field from the Container class to a new Group class. All containers belonging to the same group will point to the same Group object, containing the amount of water present in each of those containers.

In practice, our new Container class has a single field:

```
private Group group = new Group(this);
```

and Group is the following nested class.

SPEED1

```
private static class Group {
    double amount;
    Set<Container> elems;

    Group(Container c) {
        elems = new HashSet<>();
        elems.add(c);
    }
}
```

The Group class is static because we do not need each group to be permanently linked to the container that created it. It is private because it should not be

exposed to the clients. Being private, there is no point in applying visibility modifiers to its constructor and fields. The resulting memory layout is pictured in Figure 5.1.

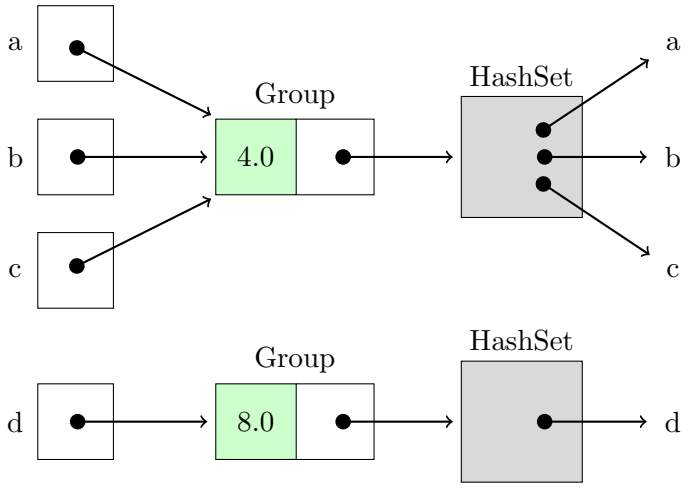


Figure 5.1: Memory layout of SPEED1 after executing USECASE, Part 2.

Then, the read and write methods of Container operate straightforwardly on the Group object.

SPEED1

```
public double getAmount() { return group.amount; }

public void addWater(double amount) {
    double amountPerContainer = amount / group.elems.size();
    group.amount += amountPerContainer;
}
```

The connectTo method is very similar to the one in REFERENCE and can be found in the repository.

Time complexity. Similarly to REFERENCE, the connectTo method still needs to iterate over all containers in a group, leading to the time complexities in Table 5.1.

method	time complexity
getAmount	$O(1)$
connect	$O(n)$
addWater	$O(1)$

Table 5.1: Time complexities of SPEED1.

There are two steps in the connectTo method that require linear time to complete:

1. merging the elements of the two groups with addAll;
2. informing the elements of one of the groups being merged that their group has changed.

The first step above is easy to replace with a faster alternative. Switch from sets to linked lists and *voilà*: merging two collections becomes a constant-time operation. Step 2, instead, is much more complicated to avoid. In fact, it is impossible to make connectTo run in constant time without raising the time complexity of getAmount. However, if for some reason we really need a constant-time connectTo, we can employ the implementation from the next section.

5.2 Optimizing "connectTo" and "addWater" [Speed2]

In this section we are going to use a radically different way to represent a group of connected containers: a manually implemented circular linked list. A circular linked list is a sequence of nodes where each node points to the next one, in a circular fashion. There is no first or last node, no head or tail.

In our application, each container is a node in a list, featuring an amount field and a single "next" reference, resulting in a singly linked list.

Speed2

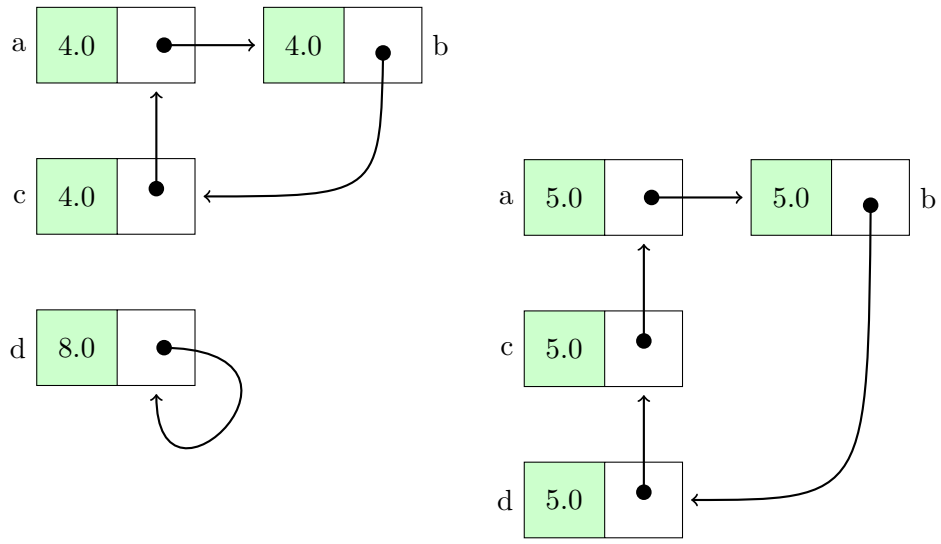
```
public class Container {

    private double amount;
    private Container next = this;
```

A nice property of circular linked lists and the very reason we are using them here is that, if you are given any two nodes from two such lists, you can merge the lists in constant time, even if the lists are singly linked. The merge is accomplished by swapping the "next" references of the two nodes, as shown in Figure 5.2.

That figure represents the memory layout in two moments during the execution of UseCase, with the implementation of containers from this section (that is, Speed2). In the left-hand side of the figure, containers a, b, and c have been connected into a single group, so they are linked to each other in a circular fashion. Container d is still isolated, so its "next" reference points to itself.

The right-hand side shows the effect of running the b.connectTo(d) instruction. Swapping the "next" references of b and d is sufficient to merge the two lists into a single one. Such swapping is in fact the only content of the following connectTo implementation.



(A) Before b.connectTo(d).

(B) After b.connectTo(d). Swapping the next pointers of b and d leads to merging the two lists into a single one.

Figure 5.2: Memory layout of SPEED2 during the execution of USECASE, before and after Part 3.

SPEED2

```
public void connectTo(Container other) {
    Container oldNext = next;
    next = other.next;
    other.next = oldNext;
}
```

To keep connectTo running in constant time, it does not update the water amounts in any way. After all, water amounts are only visible when getAmount is called. So, we *delay* the update until the next call to getAmount. This approach is a standard trick in the programmer's toolbox, called *lazy evaluation*, a staple of functional programming.

We use the same laziness with addWater, so that it only updates the current container, without actually distributing water among the group.

SPEED2

```
public void addWater(double amount) {
    this.amount += amount;
}
```

Unfortunately, sooner or later getAmount will be called, and we are going to pay for all our laziness with a costly update operation, which distributes water amounts equally within a group.

For clarity, let's delegate the update to a separate private method.

SPEED2

```
public double getAmount() {
```

```
        updateGroup();
        return amount;
    }
    private void updateGroup() {
        Container cur = this;
        double totalAmount = 0;
        int groupSize = 0;
        // First pass: collect amounts and count
        do {
            totalAmount += cur.amount;
            groupSize++;
            cur = cur.next;
        } while (cur != this);
        double newAmount = totalAmount / groupSize;
        cur = this;
        // Second pass: update amounts
        do {
            cur.amount = newAmount;
            cur = cur.next;
        } while (cur != this);
    }
```

The update method makes two passes over the circular list representing this group: in the first pass, it computes the total amount of water in the group and it counts the number of containers in it; in the second pass, it actually updates the amount of water in each container to the appropriate value.

It is easy to visit each node in a circular list. You can start from any node, follow the "next" references and just stop whenever you go back to your initial node.

A couple of questions come to mind:

1. Do we really need to invoke updateGroup every time? Perhaps we could use a boolean flag to remember whether this container is already updated and avoid unnecessary calls to updateGroup.

2. Can we move the updateGroup call from getAmount to addWater? It would be more reasonable to pay the price when writing, rather than reading.

Unfortunately, neither of these potential improvements is feasible. That is, assuming we want to keep the connection operation constant-time.

First, suppose we add an "updated" flag to all containers. Whenever a group is updated, its containers are flagged as updated. Subsequent calls to getAmount on those containers do not need to invoke updateGroup. So far so good. Now, suppose that two groups are merged with connectTo. The "updated" flags of their containers need to be reset, but this cannot be done in constant time. There goes out first improvement attempt.

Second, moving the updateGroup call from getAmount to addWater is fine, but only if a similar call is introduced in connectTo as well. Otherwise, reading the amount right after a group merge would give a stale result. Clearly, this change also puts connectTo in linear-time complexity, which is against the objectives of this section.

Time complexity. The worst-case time complexity of this implementation is summarized in Table 5.2.

method	time complexity
getAmount	$O(n)$
connect	$O(1)$
addWater	$O(1)$

Table 5.2: Time complexities of SPEED2.

5.3 The Best Balance: Union-Find Algorithms [SPEED3]

It turns out that our little container problem is similar to the classical *union-find* setting. In that scenario, one wants to maintain sets of elements, and a distinguished element for each set, called the set *representative*. The following two operations should be supported:

- merge two sets (union operation);
- given an element, find the representative from its set (find operation).

In our case, the sets would be groups of containers. The representative for a group can be any container, and we are going to use it to store the "official" water amount for that group. So, when a container receives a getAmount call, it invokes a find operation to get the value from its group representative.

Many smart computer scientists have tackled this type of problems, eventually developing the following, provably optimal, algorithm. It suggests to represent a group as a tree of containers, where each container needs only to know its parent in the tree. The root of each tree is the representative for the group. Roots should also store the size of their tree.

In computer science, a (parent pointer) tree is a linked data structure in which each node points to exactly another node, called its parent, except one special node, called the root, that points to no other node. Moreover,

all nodes can reach the root following the pointers.

The constraints above also ensure that the pointers form no cycle, so trees are a special type of directed acyclic graphs (DAGs). The height of a tree is the length of the longest path from any node to the root.

We end up with the following fields in each container:

SPEED3

```
public class Container {

    private double amount;
    private Container parent = this;
    private int size = 1;
```

The root of a tree is identified by having parent==this. You can see above that each new container is initially the root of its tree, and the only node in it. The fields amount and size are only used for the root containers. For the others, they are just wasting space. A memory-optimized implementation may want to do something about that.

A private support method called findRootAndCompress returns the representative for this container (i.e., it is the standard find operation). As its name suggests, it also modifies the tree in such a way that future calls to it are more efficient. Specifically, the method navigates the tree from this container up to the root of its tree, following the parent links. Along the way, it updates the parent reference of all encountered elements to point directly at the root. In other words, it flattens the path it travels by turning each container into a direct child of the root. As a consequence, whenever findRootAndCompress is called again on any of those objects, it will terminate in constant time, because it will immediately find the root. Here is a simple, recursive implementation of this method.

SPEED3

```
    private Container findRootAndCompress() {
        if (parent != this)
            parent = parent.findRootAndCompress();
        return parent;
    }
```

We can now easily understand the following two methods.

SPEED3

```
    public double getAmount() {
        Container root = findRootAndCompress();
        return root.amount;
    }
    public void addWater(double amount) {
        Container root = findRootAndCompress();
        root.amount += amount / root.size;
    }
```

The tree structure allows for a straightforward connection algorithm. We find the roots of the two groups being merged and we let one of the roots become a child of the other root. To get a more balanced tree, we apply the following rule: we attach the smaller tree (the one with fewer nodes) to the root of the larger tree. This is called *link-by-size* policy and it is an important ingredient to obtain the desired performance, as explained in the following section.

Figure 5.3 shows the memory layout of this implementation after executing UseCase, Part 2. At that point, b is the representative for the group comprising containers a, b, and c, whereas d is its own representative.

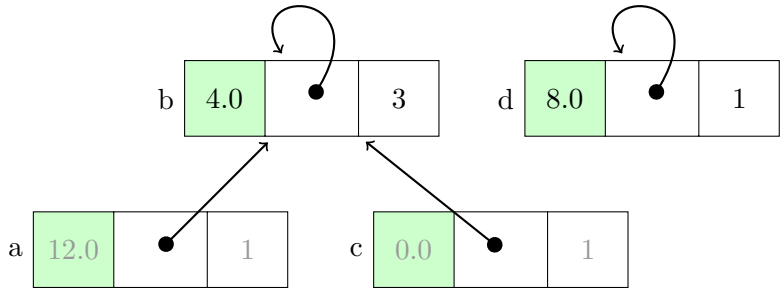


Figure 5.3: Memory layout of Speed3 after executing UseCase, Part 2. The amount and size fields of a and c are greyed out because they contain stale values that are irrelevant to the behavior of their objects. Only fields of group representatives are relevant and up-to-date.

Speed3

```
    public void connectTo(Container other) {
        Container root1 = findRootAndCompress(),
                  root2 = other.findRootAndCompress();
        int size1 = root1.size, size2 = root2.size;
        double newAmount = ((root1.amount * size1) +
                            (root2.amount * size2)) / (size1 + size2);

        if (size1 <= size2) {
            root1.parent = root2;
            root2.amount = newAmount;
            root2.size  += size1;
        } else {
            root2.parent = root1;
            root1.amount = newAmount;
            root1.size  += size2;
        }
    }
}
```

Time complexity. The first time findRootAndCompress is called on a given container, it may have to perform a number of iterations that is proportional to the current height of that tree. Under our link-by-size policy, the height of a tree is at most logarithmic w.r.t. its size.

Figure 5.4 shows a sequence of union operations that build a tree with logarithmic height. The trick is to always merge trees with the same size. For every such merge, the height of the resulting tree increases by one, but the number of nodes doubles. Hence, the height is constantly equal to the logarithm of the size.

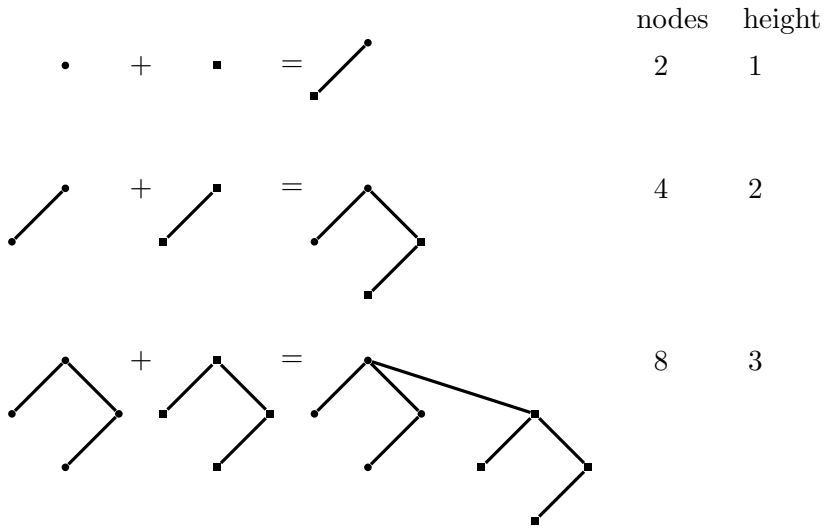


Figure 5.4: A sequence of union operations building a tree whose height is logarithmic in its size.

So, some findRootAndCompress calls require logarithmic time. Since that method is called by all of the three public methods, we obtain the worst-case time complexities in Table 5.3.

method	time complexity
getAmount	$O(\log n)$
connect	$O(\log n)$
addWater	$O(\log n)$

Table 5.3: Time complexities of Speed3.

We will see shortly how the complexities reported in Table 5.3 are in fact quite misleading, albeit formally correct. For the moment, let us use them

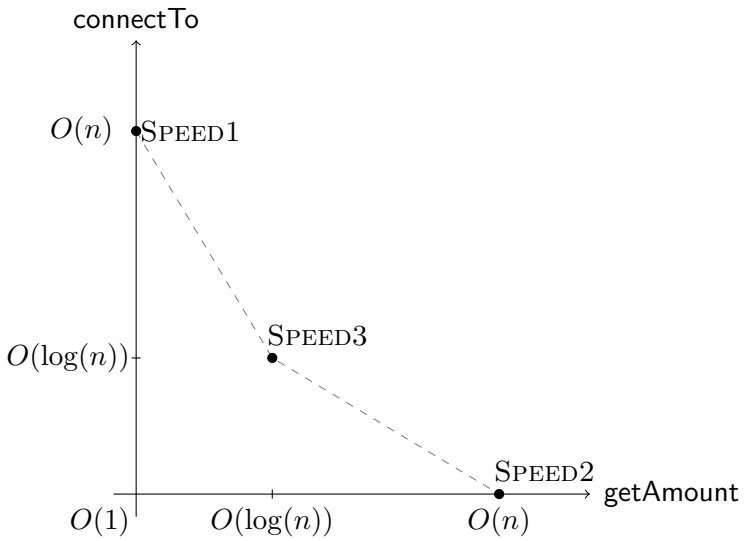


Figure 5.5: Graphical representation of the time complexity of methods getAmount and connectTo in implementations Speed1, Speed2, and Speed3. The dashed line connecting the three implementations represents a virtual Pareto front.

to compare the performance of the three implementations presented in this chapter.

Figure 5.5 puts into a graphical representation the complexity of methods getAmount and connectTo in the three versions. As anticipated at the beginning of this chapter, none of them is always better than another. Speed1 is the only one with guaranteed constant time for getAmount. Symmetrically, Speed2 features the best performance for connectTo. Speed3 strikes a balance between the two methods, attributing the same logarithmic complexity to both. When comparing any pair of implementations, one method improves its performance and the other method worsens it. To choose an implementation, we should analyse the application context and figure out how often each method is going to be called. If most calls are made to getAmount, we should prefer Speed1. Conversely, if clients are more likely to invoke connectTo, Speed2 should be picked.

However, the discussion above is unfair to Speed3, whose performance really shines if standard complexity analysis is replaced by *amortized* complexity analysis. While standard analysis focuses on a single run of an algorithm, amortized analysis takes into account *sequences* of runs. This kind of analysis is the most appropriate for algorithms that perform some extra operations, so that future calls may be more efficient. Those extra operations work as an *investment*: they are an immediate cost for a future benefit. Standard analysis would account for the cost, but not for the benefit. By considering

sequences of operations, amortized analysis manages to measure both the cost and its future benefit.

In our case, the "compress" part of findRootAndCompress is the extra cost. It is not needed to find the root, but it makes future calls faster.

To perform amortized analysis, we have to decide on a sequence of operations of arbitrary length m, performed on a set of n containers. We are interested in the long-run cost, so we assume that m can be bigger than n. Next, we have to choose how many of those m operations are connectTo, getAmount, or addWater. Notice that only $n-1$ calls to connectTo are significant: after that, all containers will be connected in a single group. So, the sequences we are going to analyze are composed as follows:

1. they are generally longer than n;
2. they contain $n-1$ calls to connectTo;
3. all the other operations are either getAmount or addWater.

Now, we can ask the asymptotic number of basic steps performed by any such sequence (i.e., the worst case among all the sequences satisfying our assumptions). The actual analysis is way beyond the scope of this book, and you are referred to the "Further Reading" section for details. In fact, even *stating* the complexity upper bound is somewhat complex! The most accurate upper bound for a sequence of m operations is not one of the "easy" functions, being slightly more than constant, but way less than quasilinear $(m \log m)$. As reported in Table 5.4, it can be written as $O\big(m \cdot \alpha(m,n)\big)$, where α is the *inverse Ackermann function*. If you are in a rush, just know that $\alpha(m,n)$ is at most 4 for all inputs m and n up to $2^{2^{2^{2^{16}}}}$, which is more than the number of atoms in the known universe. In other words, the upper bound is essentially $O(m)$ for all practical purposes. Since we are discussing the complexity of a sequence of m operations, an $O(m)$ upper bound means that, in the long run, each operation takes constant time. We couldn't hope for anything better. In fact, the experiments presented in the next section show that in this case the amortized analysis is much more relevant than the standard worst-case analysis, putting Speed3 way ahead of the competition in a typical scenario.

If you want a little more background on the function α, continue with the following theory box.

The Ackermann function $A(m,n)$ can be defined recursively as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

scenario	amortized time complexity
a sequence of m operations on n containers	$O\big(m \cdot \alpha(m,n)\big)$

Table 5.4: Amortized time complexity for SPEED3.

This function was originally proposed in 1928 by Wilhelm Ackermann, a student of renowed mathematician David Hilbert, and an accomplished researcher himself. It was the first known example of a function that is algorithmically computable, but not computable through a limited set of operations called *primitive recursive*. The key property of this function is that it grows extremely fast, even for small values of its arguments. For instance, $A(3,3)$ is a modest 61, whereas $A(4,4)$ is the humongous $2^{2^{2^{2^{2}}}} - 3$. Intuitively, the inverse Ackermann function α grows as slowly as the Ackermann function is fast.

5.4 Experiments

Skeptics may want to experimentally check that our complexity measures correspond to actual running times. To this aim, I submit you a simple experiment, where the three implementations from this chapter challenge REFERENCE on the following test case:

1. create 20,000 containers and add some water to each (20k constructor calls and 20k addWater calls);
2. connect containers in 10,000 pairs, add some water to each pair, query the amount in each pair (10k connectTo, 10k addWater, and 10k getAmount);
3. connect pairs of containers until all containers are connected into a single group; after each connection add some water and query the resulting amount (10k connectTo, 10k addWater, and 10k getAmount).

Table 5.5 summarizes the results. As expected, all classes from this chapter greatly outperform REFERENCE, by as much as two orders of magnitude. In particular, our best attempt SPEED3 is 500 times faster. On the other hand, SPEED2 is an order of magnitude slower than SPEED1 and SPEED3 (but still significantly faster than REFERENCE). As noted before, SPEED2 is a rather odd implementation, that only makes sense if getAmount queries are rare compared to the other operations. That is not the case for the tested scenario.

version	time (msec)
Reference	2 300
Speed1	26
Speed2	505
Speed3	6

Table 5.5: Running times for a balanced use case involving 20,000 containers.

version	time (msec)
Reference	2 300
Speed1	25
Speed2	4
Speed3	5

Table 5.6: Running times for a use case involving 20,000 containers and a single call to getAmount.

To verify this analysis, let us run a modified use case, where we remove all calls to getAmount except for one, at the end. We get the running times from Table 5.6. As you can see, Speed2 now matches the performance of Speed3, whereas the other three implementations are essentially unaffected by the change, demonstrating that the amount query is a very cheap operation in all other versions. The second experiment confirms that Speed3 is in practice the best choice overall.

Further Reading

Several standard algorithm books cover union-find algorithms and amortized complexity.

- T.H. Cormen, C.E. Leicerson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2009
- J. Kleinberg and E. Tardos. *Algorithm Design.* Pearson, 2005
- For a quick overview of union-find algorithms, Kevin Wayne from Princeton maintains high-quality slides that summarize their history and properties, based on the Algorithm Design book above.