Algorithmic framework programming in Java

Rodion “rodde” Efremov

May 11, 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The book

This book focuses on intersection of two areas of technical excellence: algorithms and software development. One might think that the two are the same thing, but, in fact, they do not intersect as much and as often as one would prefer. Researchers focus on getting a job done. Most of the time they have no time to refactor their code in a fancy library: they are **researchers**, their time is expensive so they focus on research. On the other hand, software development professionals may be reluctant to push the limits of their algorithmic software, so some of them confine themselves to available implementations. Without further ado, this book is about bridging the gap between aforementioned areas and groups of professionals.

1.1.1 The target audience

This book atttempts to reach as many readers as possible, yet a reader must be interested in at least one of the two major aspects covered. Also students in computer science and/or software engineering may get useful insight into the topics covered. I tried to organize the text such that the less you know the more you benefit, yet I have to assume that the reader has at least basic familiarity with some modern version of Java programming language.

Note that this is not quite a course book as there is no exercises in the book altogether. Whenever formalism and mathematics are needed, they are presented. The author will attempt to make the reader learn non-trivial techniques without

spending multitude of working hours on exercises. Also, this book may be used as a cookbook of various relevant techniques.

1.1.2 How this book is organized

The chapters (or actually material they cover) are ordered in such a way that each new chapter makes no references to succeeding chapters. If this requirement cannot be satisfied, the amount of "forward references" will be minimzed. The code for each software item we use (such as an algorithm, a data structure, an API item, and so on) will be listed in a well formatted (and colored) way. As you procede, you will fill in more and more components into a larger framework of reusable algorithms and data structures.

1.2 Toolchain

This section will list the tools you need in order to be able to reproduce all the code.

1.2.1 JDK

JDK[1] stands for "Java development kit" and it mainly consists of Java classes usable in common programming tasks. Also, a compiler, a virtual machine among other utilities are included. One important tool is `javadoc`, which generates API documentation from the comments embedded in the source code. We will assume JDK **version 8** here as it is pretty common and mature nowadays.

1.2.2 NetBeans IDE

IDE stands for "integrated development environment" and usually it is a fancy text editor tailored to writing code. Most of them provide different shortcuts for doing common tasks, and by learning 10 or so of them your productivity will boost up significantly. Also, later in this chapter we will introduce you to "fluent" APIs, which lose all their benefit if you don't use an IDE. The current version of NetBeans, as the time of writing, is 8.0.2.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`

1.2.3 Maven

Maven [2] is a tool that automates many project management related tasks such as running tests, compiling, packaging, generating documentation, and so on. If you plan to use NetBeans, the IDE will do most of Maven configuration for you.

1.2.4 git and GitHub

git [3] is a really popular version control system and **GitHub** [4] provides a web hosting service for **git** repositories. The two became de facto standard for sharing open source software, and we will use them too.

1.3 Formalism

Formalism is a scientific paradigm of communicating things precisely. One may say "I will choose only a car with the price tag between 14 and 16 thousands of dollars". Expressed formally, the set of cars that will be considered is

$$\{c \in C : p(c) \in [14000, 16000]\},$$

where C is the set of all cars and $p(c)$ is a function that gives a price of the car c. Naïvely, a **set** is a collection of items $\{i_1, i_2, \ldots, i_N\}$ in which each element appears at most once. The three dots denote continuation: there is N distinct elements in that set. Note that $\{1, 2, 3, 2, 2\} = \{1, 2, 3, 2\} = \{1, 2, 3\}$, since the same element is not allowed to repeat in any set. Now, if an element e is in the set A, we denote that fact by stating $e \in A$; if e does not belong to a set A, we state $e \notin A$.

A **function** $f : A \to B$ is a **mapping**, which assigns each element e from A some (and only one) element $f(e)$ from the set B. It is perfectly allowed that for some function f, $f(a) = f(b)$ when $a \neq b$. A function $f : A \to B$ is called **surjective** if it maps **each** element from A to some element in B. A function f is called **injective** if it assigns to each element of B no more than one element of A. Rephrased, for any injections f, $f(a) \neq f(b)$ if and only if $a \neq b$. A **bijection** is a function that is both a surjection and an injection. They map

[2] `https://maven.apache.org/`
[3] `http://git-scm.com/`
[4] `https://github.com/`

to each element $e \in A$ an unique element from B. Bijections are useful when defining some properties, such as, say, equivalences.

There is a couple of useful operations on sets: Given two sets A and B, the **union** of them is denoted by $A \cup B$, and is defined as $\{x: x \in A$ **or** $x \in B\}$. Namely, the union of two sets is the set of elements that belong **at least** to one of the sets A, B. Given two sets A and B, **intersection** of A and B is denoted by $A \cap B$, and is defined as $\{x: x \in A$ **and** $x \in B\}$: the intesection of two sets is the set of elements that belong to both of the sets A, B. Furthermore, given two sets A and B, the difference $A - B$ (also $A \setminus B$) is the set of elements belonging to A, but **not** belonging to B. Recapping,

Example 1. *If* $A = \{2, 4, 7\}, B = \{1, 2, 3, 4, 6\}$,

- $A \cup B = \{1, 2, 3, 4, 6, 7\}$,
- $A \cap B = \{2, 4\}$,
- $A \setminus B = \{7\}$.

Suppose we choose N distinct persons and measure each persons height and weight. We would end up with **ordered pairs** (or in general **tuples**) of the form $(h_1, w_1), \ldots, (h_N, w_N)$. If H is the set of possible heights and W is the set of possible weights, the **Cartesian product** of H and W is denoted by $H \times W$ and it means $\{(h, w): h \in H$ **and** $w \in W\}$. Since $H \times W$ is also a set, you can extend to, say, $(H \times W) \times A = H \times W \times A$ for some set A and you will get a set of **triples**, and so on. The elements of a tuple is not required to be an element of some set; it can **be a set** or something more abstract. Also, elements within a tuple may be of "different" types: one tuple element may be a complex number, another one a set, the third a function, and so on.

Often we deal with collections where some elements repeat. In our daily shopping, two or more packs of milk may end up in our market basket. Sets can't do this, but **multisets** can: a multiset is just like a set that allows repetition of some elements. Formally, a multiset is defined as an ordered pair (A, f), where A (as a set) contains element "identities", and for each item $a \in A$, $f(a)$ gives the amount of objects wit identity a in the multiset.